INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Padrões de Desenho para Contratos Inteligentes em Ethereum

JOÃO MIGUEL RIBEIRO DA COSTA
Outubro de 2019

POLITÉCNICO
DO PORTO

Design Patterns for Ethereum Smart Contracts

João Miguel Ribeiro da Costa

A dissertation submitted in fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Software Engineering

Supervisor: Isabel de Fátima Silva Azevedo

Porto, October 13, 2019

Abstract

The invention of Bitcoin in 2008 offered a solution for a digital currency that could be used without a trusted third-party settling disputes over transactions. Bitcoin relied on a technology known as the blockchain, which can be described as a distributed database that relies on a consensus mechanism (generally Proof Of Work is employed) to be resilient against tampering.

Ethereum, launched in 2015, leveraged the blockchain technology to augment the initial proposal of Bitcoin, enabling computational statements to be executed as part of each block validation. The platform offers a Turing-complete runtime environment (the Ethereum Virtual Machine), which can run smart contracts - scripts that verify and enforce the execution of predefined legal contracts.

The technical development of smart contracts present significant challenges that are not well modeled by the current body of knowledge and practices of software engineering. In fact, some of the characteristic of blockchain make the contract execution uncontrollable by the programmer and immutable after deployment. Also, the potential security risks are considerable, since there is a large incentive to exploit vulnerabilities in a smart contract for financial gain.

Considering the concerns presented above, the establishment of well understood and well-defined design patterns for the development of smart contracts is of paramount importance. In the realm of software engineering, design patterns are defined as generic and reusable solutions to common problems in software design.

In the context of this work, a survey of design patterns that target the Ethereum framework was performed, with an extensive analysis regarding the context in which they can be employed, as well as implementations, examples and consequences of their use. A total of 11 design patterns were analysed. The design patterns identified for the Ethereum framework focus on several concerns specific to this platform – most of these concerns revolve around safety, upgradeability, and the limitations inherent to the sandboxed approach of the Ethereum Virtual Machine.

A Decentralized Application (dApp) was created to showcase the employment of several of the identified contracts, and to highlight the value they can provide. This dApp offers a framework for decentralized betting in a *trustless* environment, where neither the user needs to trust the owner nor vice-versa. The dApp implements several use cases that are reliant on the identified design patterns.

Keywords: blockchain,ethereum,design patterns,smart contracts,blockchain software engineering

Resumo

A invenção da Bitcoin em 2018 disponibilizou uma solução para uma moeda digital que poderia ser usada sem a necessidade de envolver terceiros para a mediação de transações. A Bitcoin recorre a uma tecnologia conhecida como blockchain, que consiste numa base de dados distribuída, assente num mecanismo de consenso resistente a alterações não acordadas.

Ethereum, lançada em 2015, utiliza a mesma tecnologia da blockchain para oferecer uma plataforma que se baseia na Bitcoin, mas que também permite a execução de instruções como parte do processo de validação de cada bloco. A plataforma permite correr contratos inteligentes (smart contracts) - scripts que verificam e garantem a correta execução de um contrato predefinido.

O desenvolvimento técnico de contratos inteligentes apresenta desafios significativos que não são atualmente modelados pela área de engenharia de software. De facto, algumas das caraterísticas da blockchain fazem com que a execução de contratos não seja controlável pelo programador e também com que estes contratos sejam imutáveis após serem colocados na rede principal de Ethereum.

Tendo em conta os pontos anteriores, é importante o estabelecimento de padrões de desenho (design patterns) bem definidos para serem usados em contratos inteligentes.

No contexto deste trabalho, foi realizada uma análise dos padrões de desenho usados em Ethereum, tendo em conta o contexto em que são utilizados, as suas implementações e exemplos da sua utilização em contratos existentes. Um total de 11 padrões de desenho foram identificados e analisados.

Uma Decentralized Application (dApp) foi desenvolvida para demonstrar o emprego dos padrões de desenho identificados. Esta dApp disponibiliza uma framework para se efetuar apostas de uma forma descentralizada, em que nem o utilizador necessita de confiar no dono do contrato, nem vice-versa.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter aims to explain the context and problem that will constitute the body of work in this dissertation. The methodology that will serve as the base for the work to be developed is introduced. Finally, the structure of this document is also presented.

1.1 Context

The emergence of Bitcoin in 2008 [1] offered a compelling solution for a digital currency that could be efficiently used without a trusted third-party settling disputes over transactions. The technology that enables this groundbreaking achievement is blockchain, which can be described as a distributed database that relies on proof of work methods to be resilient against tampering by malicious actors.

As a technology, blockchain was quickly adopted beyond the original purposes of serving as a ledger for currency transactions. Ethereum, launched in 2015, rose to preeminence by proposing a decentralized platform that enables computational statements to be executed as part of each block validation [2]. In fact, the platform offers a Turing-complete runtime environment (the Ethereum Virtual Machine), which can be leveraged to run smart contracts - scripts that verify and enforce the execution of predefined legal contracts [3]. The blockchain, due to its own properties, can provide guarantees that the transactions initiated by the smart contract are truthfully executed.

1.2 Problem

While the use of smart contracts has been increasingly popular, particularly in the Ethereum ecosystem, they present significant technical challenges that are not well modelled by the current body of knowledge and practices of software engineering [4]. In fact, some of the characteristic of blockchain make the contract execution uncontrollable by the programmer and immutable after deployment [5].

The potential security risks are very large, as there is a large incentive to exploit vulnerabilities in the smart contract for financial gain [4, 5]. This is compounded by the fact that the source code for the smart contracts is public, since it is part of the blockchain once deployment occurs.

Moreover, current development tools do not offer adequate support, which, conjugated with poor architectural design decisions, augments the probability of existence of security issues

[6]. Inadequate software development practices have been considered the main cause of successful attacks [7], and to cement the field of blockchain based software engineering is, in this context, a critical endeavour.

Considering the concerns presented above, the establishment of well understood and well-defined design patterns for the development of smart contracts is of paramount importance.

1.3 Objectives

The main objectives of this work are:

- Identify common patterns in publicly available smart contracts targeting the Ethereum framework.
- Compare available patterns and identify the advantages and drawbacks of specific design patterns, according to their aim, as well as their costs.
- Implement proof of concepts for various design patterns.

1.4 Research Methodology

With the high growth and increasing interest on smart contracts, addressing the specific challenges that they pose in terms of security as well as in the broader sense of software engineering practices is fundamental.

The Ethereum platform, as the primary framework for smart contracts deployment and execution, has been the target of high profile attacks, which resulted in the loss of large quantities of Ether, valued in hundreds of millions dollars. Inefficient computations in the currently deployed smart contracts are also a source of major losses, since these inefficiencies are levied onto the users in the form of unnecessarily high transaction costs.

An extensive knowledge of these drivers is required to base the work developed in this project. As such, in an initial phase, a survey of the Ethereum smart contract landscape was done, in order to identify high profile examples of lack (or wrongful) of use of design patterns.

Based on the available open sourced contracts, an identification of implemented design patterns was performed. In both cases, priority was given to the most relevant contracts.

Following this initial stage, with a strong investigative nature, a second phase corresponded to the implementation of smart contracts that showcase the use of the identified design patterns.

A Gantt chart illustrating the proposed schedule for this work is represented in figure 1.1.

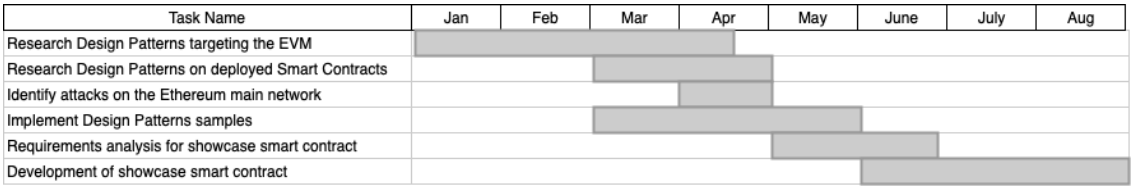

Figure 1.1: Gantt chart representing the research work schedule.

1.4.1 Systematic Mapping Study

In order to accomplish the goal of identifying relevant design patterns in the Ethereum smart contract ecosystem, a systematic approach was followed, ensuring proper review of the relevant literature. The employment of systematic reviews allows to identify and summarize empirical evidence of the field, as well as to identify eventual gaps in the state of the art.

While there are several approaches to perform systematic reviews of the literature, the chosen approach was the systematic mapping review. In this approach, the definition of one or several research questions is the starting point for the analysis. The research questions used were:

1. What are the most effective design patterns for smart contract development in the Ethereum Framework?
2. What are the specific software engineering concerns in the development of smart contracts?

Search strings and acceptance criteria

In order to conduct a search, a set of search strings was defined:

1. ethereum design patterns
2. design patterns blockchain
3. design patterns smart contracts
4. blockchain software engineering

A set of acceptance criteria, for which all considered papers should comply with, was also set:

1. only papers written in english
2. only include papers published after 2015
3. only consider papers with at least 5 citations
4. only consider papers with focus on Ethereum and Solidity

Search

The search was performed independently for each search string. For the search string "ethereum design patterns", the following papers were found:

- A Pattern Collection for Blockchain-based Applications [8]
- A Semantic Framework for the Security Analysis of Ethereum Smart Contracts [9]
- A Taxonomy of Blockchain-Based Systems for Architecture Design [10]
- An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns [11]

- Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps [12]
- Design Patterns for Smart Contracts in the Ethereum Ecosystem [13]
- Design of Blockchain-Based Apps Using Familiar Software Patterns to Address Interoperability Challenges in Healthcare [14]
- Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach [15]
- Detecting Token Systems on Ethereum [16]
- Ethereum Smart Contract Development [17]
- Ethereum Smart Contracts: Security Vulnerabilities and Security Tools [18]
- Ethereum transaction graph analysis [19]
- Ethereum: State of Knowledge and Research Perspectives [20]
- Foundations and Tools for the Static Analysis of Ethereum Smart Contracts [21]
- Mastering Ethereum: Building Smart Contracts and dApps [22]
- Security Vulnerabilities in Ethereum Smart Contracts [23]
- Smart contracts: security patterns in the ethereum ecosystem and solidity [24]
- Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts [25]
- VeriSolid: Correct-by-Design Smart Contracts for Ethereum [26]

For the search string "design patterns blockchain":

- A Pattern Collection for Blockchain-based Applications [8]
- A typology of blockchain recordkeeping solutions and some reflections on their implications for the future of archival preservation [27]
- Applying Design Patterns in Smart Contracts [28]
- Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps [12]
- BlockSci: Design and applications of a blockchain analysis platform [29]
- Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends [30]
- Chapter One - Blockchain Technology Use Cases in Healthcare [31]
- Comparative Requirements Analysis for the Feasibility of Blockchain for Secure Cloud [32]
- Design Patterns for Smart Contracts in the Ethereum Ecosystem [13]
- Design of Blockchain-Based Apps Using Familiar Software Patterns to Address Interoperability Challenges in Healthcare [14]
- FHIRChain: Applying Blockchain to Securely and Scalably Share Clinical Data [33]

- How Much Blockchain Do You Need? Towards a Concept for Building Hybrid dApp Architectures [34]
- On the design of communication and transaction anonymity in blockchain-based transactive microgrids [35]
- Toward an ontology-driven blockchain design for supply-chain provenance [36]
- Virtual Resources& Blockchain for Configuration Management in IoT [37]

For the search string "blockchain software engineering":

- A Novel Method for Decentralised Peer-to-peer Software License Validation Using Cryptocurrency Blockchain Technology [38]
- Adaptable Blockchain-Based Systems: A Case Study for Product Traceability [39]
- An Agile Software Engineering Method to Design Blockchain Applications [40]
- An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns [11]
- Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps [12]
- Block-VN: A Distributed Blockchain Based Vehicular Network Architecture in Smart City [41]
- Blockchain Platform for Industrial Internet of Things [42]
- Blockchain and Artificial Intelligence [43]
- Blockchain and Business Process Improvement [44]
- Blockchain challenges and opportunities: a survey [45]
- Blockchain technology innovations [46]
- Blockchain-Oriented Software Engineering: Challenges and New Directions [47]
- CitySense: blockchain-oriented smart cities [48]
- Comparing Blockchain and Cloud Services for Business Process Execution [49]
- How digital identity on blockchain can contribute in a smart city environment [50]
- Obsidian: a safer blockchain programming language [51]
- Smart contracts vulnerabilities: a call for blockchain software engineering? [52]
- Toward an ontology-driven blockchain design for supply-chain provenance [36]

For the search string "design patterns smart contracts":

- A Concurrent Perspective on Smart Contracts [53]
- A Semantic Framework for the Security Analysis of Ethereum Smart Contracts [9]
- A Taxonomy of Blockchain-Based Systems for Architecture Design [10]
- An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns [11]

- An architecture pattern for trusted orchestration in IoT edge clouds [54]
- Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps [12]
- Auditing with Smart Contracts [55]
- BitML: A Calculus for Bitcoin Smart Contracts [56]
- Blockchain Disruption and Smart Contracts [57]
- Chainspace: A Sharded Smart Contracts Platform [58]
- ContractFuzzer: fuzzing smart contracts for vulnerability detection [59]
- Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach [15]
- Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts [60]
- Erays: Reverse Engineering Ethereum's Opaque Smart Contracts [61]
- Ethereum Smart Contracts: Security Vulnerabilities and Security Tools [18]
- Formal Verification of Smart Contracts: Short Paper [62]
- MadMax: surviving out-of-gas conditions in Ethereum smart contracts [63]
- Mastering Ethereum: Building Smart Contracts and dApps [22]
- ReGuard: finding reentrancy bugs in smart contracts [64]
- Safer smart contracts through type-driven development [65]
- Scilla: a Smart Contract Intermediate-Level LAnguage [66]
- Securify: Practical Security Analysis of Smart Contracts [67]
- Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security [68]
- Smart Contracts Software Metrics: a First Study [69]
- Smart contract applications within blockchain technology: A systematic mapping study [70]
- Smart contracts vulnerabilities: a call for blockchain software engineering? [52]
- Smart contracts: security patterns in the ethereum ecosystem and solidity [24]
- SmartCheck: Static Analysis of Ethereum Smart Contracts [71]
- Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts [25]
- ZEUS: analysing Safety of Smart Contracts [72]

After removing duplicates, a total of 65 papers were found as the result of this search.

Screening

The screening phase corresponds to an analysis of all the relevant papers identified in the search phase. This analysis takes into account the pre defined acceptance criteria (in section 1.4.1). Only papers that comply with all acceptance criteria are accepted.

From the original 65 papers, a total of 27 were considered, after the screening phase.

Those papers are:

1. A Pattern Collection for Blockchain-based Applications [8]
2. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts [9]
3. An Agile Software Engineering Method to Design Blockchain Applications [40]
4. An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns [11]
5. Applying Design Patterns in Smart Contracts [28]
6. Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps [12]
7. Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends [30]
8. Blockchain-Oriented Software Engineering: Challenges and New Directions [47]
9. ContractFuzzer: fuzzing smart contracts for vulnerability detection [59]
10. Design Patterns for Smart Contracts in the Ethereum Ecosystem [13]
11. Design of Blockchain-Based Apps Using Familiar Software Patterns to Address Interoperability Challenges in Healthcare [14]
12. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach [15]
13. Detecting Token Systems on Ethereum [16]
14. Erays: Reverse Engineering Ethereum's Opaque Smart Contracts [61]
15. Ethereum Smart Contracts: Security Vulnerabilities and Security Tools [18]
16. Foundations and Tools for the Static Analysis of Ethereum Smart Contracts [21]
17. How Much Blockchain Do You Need? Towards a Concept for Building Hybrid dApp Architectures [34]
18. MadMax: surviving out-of-gas conditions in Ethereum smart contracts [63]
19. Mastering Ethereum: Building Smart Contracts and dApps [22]
20. ReGuard: finding reentrancy bugs in smart contracts [64]
21. Safer smart contracts through type-driven development [65]
22. Securify: Practical Security Analysis of Smart Contracts [67]
23. Security Vulnerabilities in Ethereum Smart Contracts [23]

24. Smart contracts vulnerabilities: a call for blockchain software engineering? [52]
25. Smart contracts: security patterns in the ethereum ecosystem and solidity [24]
26. SmartCheck: Static Analysis of Ethereum Smart Contracts [71]
27. VeriSolid: Correct-by-Design Smart Contracts for Ethereum [26]

These screened papers served as the basis for the investigation performed in this work.

1.5 Structure

This document is structured as follows:

Introduction in this chapter the context of this work is presented, along with some important concepts about the work to be developed, the objectives of the work and its methodology.

State of the Art this chapter focus on providing a sound background on the concepts involved in the development of this work: the fundamental concerns of contracts, the technical details behind the blockchain technology and the inner work of the Ethereum platform.

Value Analysis on this chapter, the value that this work can bring to a customer is analysed.

Design Patterns this chapter provides a detailed analysis of several design patterns targeting the Ethereum platform.

Betting On The Block dApp this chapter details the implementation of a decentralized application employing several design patterns.

Conclusion this chapter provides an overview of the achievements of this work, as well as some considerations for future improvements.

Chapter 2

State of the Art

In this chapter, an introduction to the conceptual topics that constitute the background to the work that will be developed are presented, as well as the current state of some of the technologies that will be researched.

2.1 Contracts

A contract can be broadly defined as a binding agreement between two or more parties that is legally enforceable. This definition, though not complete, offers a starting point for the introduction of smart contracts. Contracts allow for a formalization of a relationship, and can apply to both business and personal relationships (marriage being a classical example). They have a strong legal character and their study is an important focus of Economics.

Governmental institutions are often used as the third party responsible for enforcement and arbitration of a contract, although private arbitration can also be employed.

A free market economy, where private parties enter in voluntary exchange of services and goods for mutual benefit, is extremely dependent on contracts. The lack of enforceability of these can be a relevant hindrance for economic development, given that it could discourage mutual beneficial relationships from occurring. The importance of this is reflected in indices such as the "Enforcing Contracts", where significant correlation is found between contract enforceability scores and a country economic development [73].

In [74], four basic principles of contract design are defined:

Observability the involved parties should be able to observe the performance of the contract for all other parties, and to prove their performance.

Verifiability a party should be capable of proving to a third party (arbitrator) that a contract has been performed or has been breached.

Privity the knowledge over the contents (and performance) of a contract should be distributed only to the parties involved and limited third parties (arbitrators, for instance).

Enforceability incentives included in the contract should guarantee that the contract is enforceable.

The concept of smart contracts was coined by Szabo [74], and refers to the idea that several aspects of contractual clauses can be well modelled by software or even specialized hardware. The vending machine is given in [74] as a primitive example of a smart contract: a simple automata will dispense some product if a certain amount of coins is provided and will also

provide change accordingly. More advanced examples include Point Of Sale software which process electronic transactions.

Advantages of smart contracts include inherent observability and verifiability, as well improved enforceability. They can also help circumvent legal barriers, which constitute a large cost when developing business across different jurisdictions. Another important advantage is the low transaction costs typically associated with smart contracts - this however is not guaranteed for cryptocurrencies, as large valuations as well as increasing rate of transactions have led to very high costs.

2.2 Blockchain

Smart contracts, as explained in [74], typically require one or several trusted third parties to audit, arbitrate and mediate disputes. This however does bring some of the drawbacks of standard contracts - the principals need to agree on the third parties to be trusted, and any attack or malicious actions by this third party could risk the reliability of the contract. Although it is not the topic of this work, this problem is especially relevant in online commerce and transactions - a third party always needs to be involved to mediate the transactions, raising those transactions costs.

Digital signatures (explained in section 2.2.2) can provide a means for electronic transactions to be performed. A certain party, holder of a private key for an account with funds, could perform a transaction by digitally signing it. This could ensure that this party intended to perform the transaction, but no guarantee could be made that he would not reuse the same funds for another transaction - this is commonly known as the double spending problem. In order to avoid this, a third party is needed. The mechanism for this type of transactions is illustrated in figure 2.1.

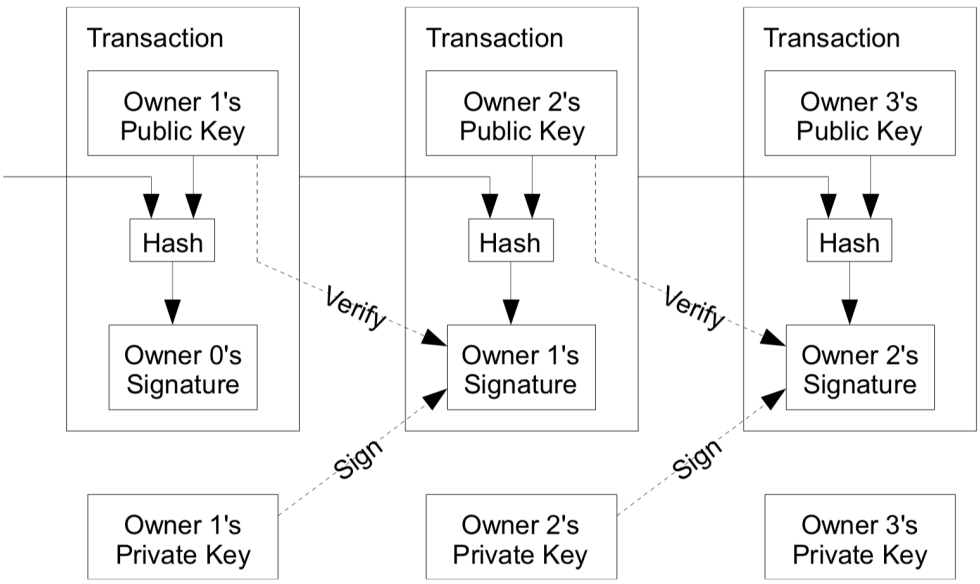

Figure 2.1: Representation of a set of digitally signed transactions. From [1].

In [1], it is argued that an electronic payment system should rely on cryptographic proof instead of trust, thus allowing direct transactions between two parties without a middleman. This would obviously require solving the double-spending problem. In [1], it is proposed a distributed approach based on cryptographic proof of work and digital signatures to allow this, a technology now commonly referred as blockchain. The initial proposed implementation of this technology was in an electronic coin, Bitcoin.

The solution considers a timestamp server, where various blocks (represented by a hash of its contents) are chained / linked to each other in a linear fashion. Each new block (represented by its hash) contains the hash of the previous block. Due to this, each new block reinforces the ones that precede it. In the implementation of Bitcoin, each block contains a set of transactions, whose structure is similar to the one represented in figure 2.2.

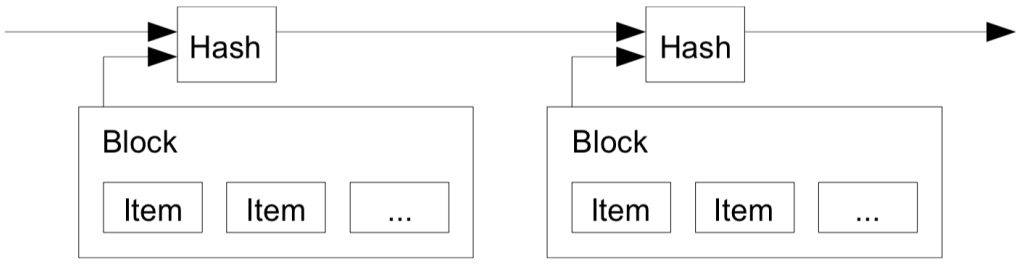

Figure 2.2: Representation of a chain of blocks. From [1].

2.2.1 Consensus Mechanism

To be secure, a distributed blockchain requires a distributed consensus mechanism, enabling synchronization and acceptance of the individual chains that each node has access to. In Bitcoin, this is addressed by a Proof Of Work (PoW) mechanism - each node of the distributed network must be able to prove the consumption of resources/time to add a new block to the current chain.

The proof of work used is based on Hashcash [75]. Each node will retrieve a set of transactions from the network to include in the next block and, from there, will start by validating each of these transactions. The new block to mint will include this set of transactions as well as the hash of the previous block and nonce - this last item is computed by the proof of work system. The proof of work will scan for a nonce value that, when hashed with a cryptographic hashing function, will produce a hash that start with a configurable number of zero bits - the larger the number of zero bits, the harder the proof.

Mining nodes will compete for being the fastest to mint a new block and since finding the right nonce is a random process, any malicious actor not in control of the majority of the nodes would not be able to revert a transaction after it was integrated in the blockchain. Verifying that the work was done is relatively cheap, which allows the node that computed the block to broadcast it and other nodes in the system can easily verify that the work was effectively expended, and would consider the now longer blockchain as the correct one and would start mining the next block from the new block hash of this chain.

Bitcoin introduces two incentives for mining new blocks - each transaction can include small fees to be paid to the mining node and, for a limited time, each new mining node can create a new bitcoin for itself.

Proof Of Work is the consensus mechanism used by most cryptocurrencies. However, there are two significant disadvantages associated with this protocol:

- Extremely high energy consumption to maintain the network, at a global scale.
- Low transaction thoroughput.

Alternative Consensus Mechanisms

Due to the limitations of Proof Of Work as a consensus mechanism, research on alternatives has seen a rise in interest. The most common alternatives are Proof Of Stake (PoS) and Delegated Proof Of Stake (DPoS), although others were proposed such as Proof Of Capacity and Proof Of Elapsed Time.

Proof Of Stake consensus relies on the assumption that the holders of large quantities of the currency would have an incentive to positively participate in the network maintenance and in increasing the value of their coins. In PoS, any holder of the coin could stake their property to be allowed to participate in a randomized process to generate a new block - in case it is selected, the proposing holder would earn transaction fees and, possibly, newly generated coins.

Delegated Proof Of Stake is another alternative, where the holders of coins can vote for delegates, in proportion to the amount of coins that they hold. The elected delegates are then eligible to create new blocks, in the same way as for the PoS method.

Both methods address the main concerns of PoW, enabling higher transaction thoroughputs and much lower overheads for operating the networks. However, both present their own type of limitations. In the case of PoS, implementing it has been particularly difficult and there are no coins with high market capitalizations currently using it. In the case of DPoS, decentralization is sacrificed, opening up new attack vectors that do not exist for networks using PoW.

2.2.2 Asymmetric cryptography

The use of cryptography is ubiquitous in any blockchain, as it is used for authenticating any sender of a transaction. In cryptography, two categories of algorithms are considered: symmetric and asymmetric cryptography [76].

In symmetric (or conventional) cryptography, the sender and the receiver of a message hold a shared secret, the key, that can be used to both encrypt and decrypt messages. In opposition, asymmetric cryptography (or public key cryptography) uses a more convoluted approach to avoid the problem of having the sender and receiver communicating the shared secret. In this system, a pair of keys is used - a public and a private key - and if one key is used to encrypt the message, the other can be used to decrypt it. The public key is generally published and is available to anyone while the private key is a secret.

The properties of asymmetric cryptography allows several important use cases. One of those is digital signatures - as traditional signatures, these guarantee that a given message was sent by some party with access to a private key, thus ensuring authentication. These digital signatures are created by encrypting a hash of the message with the private key. A party with access to the public key can then use that key to decrypt the hash and verify that it really is the correct hash of the sent message. Since tampering with the message would change its hash, it is guaranteed that the private key holder was the sender of the message if this decrypted hash matches the message.

2.3 Ethereum

As previously referred the first implementation of the blockchain technology was focused in allowing decentralised transactions for an electronic currency, the Bitcoin. However, other applications were also possible, such as tracking the ownership of physical devices and assets such as domain names, decentralized financial exchange platforms, peer to peer gambling, etc. Blockchain also enables the implementation and deployment of decentralized smart contracts.

The Ethereum platform was proposed as a means to provide a "fully-fledged Turing-complete programming language that can be used to create "contracts" that can be used to encode arbitrary state transition functions" [77].

Ethereum expands upon the previous original proposal of blockchain to include a complete Virtual Machine type of environment running on each node, and code can run as part of the validation of the creation of each block. This code can alter the state of the blockchain, by transferring some amount of an ether token. This ether token is the corresponding currency in the Ethereum platform. Each computational step requires a specific amount of "gas", thus creating the incentive for mining nodes to run the code in each smart contract, as well as for smart contract developers to ensure that the execution of the code is as performant as possible.

Although Ethereum is built on top of the Bitcoin platform Turing incomplete scripting, it offers significant features:

- It is Turing complete (it allows, for instance, loop constructs).
- Value-awareness, meaning that the code has access to the value and parties involved in a transaction.
- Blockchain awareness.

In [77], the blockchain is conceptualized as being the history of changes to the state of a decentralized database. As such, each block will consist of the set of changes to the previous state. The state of the Ethereum blockchain is considered to be formed by a set of objects, the accounts. Each state transition will consist of transfers of value (Ether) and information between the accounts. An account contains the following information:

- Nonce, a simple counter
- Current Ether balance
- Contract code

- Account storage

Two types of account can be considered: externally owned account and contract accounts. Externally owned accounts are controlled through their private keys and have no associated code, being able to send transactions. Contract accounts have associated code whose execution can be triggered by transactions or messages. When activated, the execution code can in turn read and write to the internal storage and send other messages (to trigger other contracts code execution) or even create new contracts.

Transactions in Ethereum are special types of messages, which are signed by an externally owned account. A transaction contains the recipient of the message, a signature if the sender, the data and ether to send, as well the values STARTGAS and GASPRICE. These last two items are used to limit the amount of computation that can be triggered by a transaction. If a transaction runs out of gas all the computational steps executed are reverted. The state transition associated with a transaction is represented in figure 2.3.

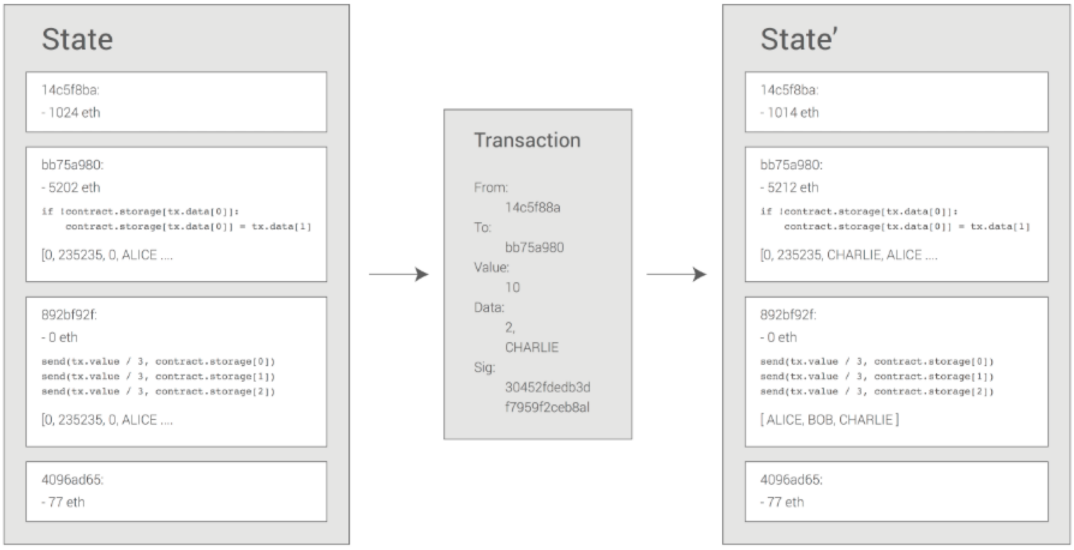

Figure 2.3: Representation of a state transition in Ethereum. From [77].

2.3.1 Ethereum Virtual Machine

All the code that runs as part of contract accounts, uses the Ethereum Virtual Machine runtime (EVM). The EVM is sandboxed from the user system and, due to this, any code that is running will have no access to the network or the file system, although it has access to some information in the blockchain and can interact with other accounts through message sending.

There are three types of memory in the EVM:

- Stack, where all the computations are performed. The maximum size is 1024 and each word has 256 bits. Arbitrary access is limited to the top 16 elements in the stack.
- Memory, which is reset at each execution and is theoretically infinite in size - however, due to gas costs scaling quadratically, it becomes extremely expensive to hold an unnecessary amount of memory.

- Storage, the only persistent type of memory. This is a key value store, where each key and each word has a size of 256 bits.

The EVM executes a stack based bytecode, consisting of a series of bytes, each one representing an operation. A program counter is used to keep track of the current operation and is incremented whenever an operation is performed.

Due to the properties of the EVM, its full computational state is represented by the set of *block_state*, *transaction*, *message*, *code*, *memory*, *stack*, *program_counter*, *gas*.

It is possible for contracts to interact with other contracts or sending ether by employing the concept of message calls. The calling contract can specify how much of its gas will be sent and the payload. If the execution of the called contract runs out of gas, this will cause an error to be placed in the top of the stack. The called contract may return data and its execution is synchronous. A special type of message sending is the *delegatecall* - in this case, the code of the called contract will be executed with the current context (*msg.sender* and *msg.value*), allowing to use contracts as common software libraries.

Smart contracts are also able to perform other operations such as creating other contracts or self destructing, consequently sending all the funds to a pre-designated address.

2.3.2 Consensus Mechanisms

As discussed in 2.2.1, Bitcoin uses Proof Of Work as its consensus mechanism. Since Ethereum also leverages much of the technical achievements of Bitcoin, it also uses PoW as the consensus mechanism. However, the limitations of this approach have been recognized early in the development of Ethereum and a transition to a Proof Of Stake has been identified as a priority.

A PoS implementation has been in development for several years, with the eventual release suffering several delays. The current PoS mechanism roll-out date is currently estimated between 2019 and 2021.

Along with the implementation of the new consensus approach, several scalability features are also being implemented, such as sharding - where instead of having a single main blockchain, there would be several ones linked together, but independent.

2.3.3 Coding for the EVM

The Ethereum Virtual Machine executes byte code for which multiple high level programming languages were designed. A brief overview of the main programming languages for the EVM will be given in this section, with special emphasys on Solidity.

Serpent

Serpent is object oriented programming language deeply inspired by Python, with a simple syntax and dynamic typing. Despite its roots in a very high level language, Serpent provides low level operations on the EVM.

Even though it is one of the earliest languages for the EVM, its development has been practically stalled since October 2015 [78] and significant issues were found in an audit, mostly focused around the low code quality (lack of tests and documentation) and poor error handling [79].

LLL

LLL, standing for Low-level Lisp-like Language is based, as its name suggests, on Lisp. It offers low level access to the EVM, including direct access to memory and storage and direct control over the EVM opcodes.

The low level nature of this language provides some advantages such as very small binaries and highly optimized bytecode. These advantages can yield significant cost savings depending on the contract being developed.

Mutan

Mutan is another language targeting the EVM, offering a C like syntax. Mutan is deprecated since March 2015.

Vyper

Vyper is another language based on Python and is distinguishable from others by its lack of some common features such as modifiers, class inheritance, function and operator overloading, recursion, etc. One of the main reasons for avoiding some of the constructs is the strict requirements for security and readability.

Solidity

Solidity is the most used language for smart contract implementation. It is object oriented and is based on C++, Python and Javascript. Solidity uses static typing and fully supports inheritance. In this section, the structure and syntax of the language will be presented.

```
pragma solidity >=0.4.0 <0.6.0;

contract SimpleStorage {
    uint storedData;

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

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

In line 1, a *pragma* instruction indicates that this contract should only be compiled with version 0.4.0 to 0.6.0 (not inclusive) of the compiler.

In line 4, an unsigned integer variable *storedData* is declared as part of the (permanent) storage. The contract contains two functions, *get* and *set*, that allow the caller to retrieve and update the stored value.

The following example showcases a voting contract, a simplified version of an example provided in [80]. In this contract, every voter is able to cast a vote in a single proposal, that are predefined when the contract is created. Due to the advantages of the blockchain, this contract solves one of the most important issues in the design of voting system - proving that the result cannot be tampered.

```
1  pragma solidity >=0.4.22 <0.6.0;
2
3  contract Ballot {
4      struct Voter {
5          uint weight; // weight of the vote
6          bool voted;  // if true, that person already voted
7          uint vote;   // index of the voted proposal
8      }
9
10     // This is a type for a single proposal.
11     struct Proposal {
12         bytes32 name;   // short name (up to 32 bytes)
13         uint voteCount; // number of accumulated votes
14     }
15
16     address public chairperson;
17
18     mapping(address => Voter) public voters;
19
20     Proposal[] public proposals;
21
22     constructor(bytes32[] memory proposalNames) public {
23         chairperson = msg.sender;
24         voters[chairperson].weight = 1;
25
26         for (uint i = 0; i < proposalNames.length; i++) {.
27             proposals.push(Proposal({
28                 name: proposalNames[i],
29                 voteCount: 0
30             }));
31         }
32     }
33
34     // May only be called by `chairperson`.
35     function giveRightToVote(address voter) public {
36         require(
37             msg.sender == chairperson,
38             "Only chairperson can give right to vote."
39         );
40         require(
41             !voters[voter].voted,
42             "The voter already voted."
43         );
44         voters[voter].weight = 1;
45     }
46
```

```
47      function vote(uint proposal) public {
48          Voter storage sender = voters[msg.sender];
49          require(sender.weight != 0, "Has no right to vote");
50          require(!sender.voted, "Already voted.");
51          sender.voted = true;
52          sender.vote = proposal;
53
54          proposals[proposal].voteCount += sender.weight;
55      }
56
57      function winningProposal() public view
58              returns (uint winningProposal_)
59      {
60          uint winningVoteCount = 0;
61          for (uint p = 0; p < proposals.length; p++) {
62              if (proposals[p].voteCount > winningVoteCount) {
63                  winningVoteCount = proposals[p].voteCount;
64                  winningProposal_ = p;
65              }
66          }
67      }
68
69      function winnerName() public view
70              returns (bytes32 winnerName_)
71      {
72          winnerName_ = proposals[winningProposal()].name;
73      }
74  }
```

As explained, this contract includes the concepts of a *Voter* and a *Proposal*, which are defined as *structs*. An associative array is used to map between Ethereum addresses and voter information. Upon the creation of the contract (and only then), the constructor declared in line 22 is invoked, and the list of proposals is populated. A *chairperson* is also defined - this corresponds to an Ethereum address that is able to give right to vote to other addresses. This power is attributed to the Ethereum address that created the contract.

In line 35, the function for giving a right to vote is declared. This function includes a provision to ensure that its caller is the *chairperson*.

The function invoked to cast a vote is declared in line 47. This function ensures that the sender is in fact able to vote, by having a non zero weight attribute, as well as not having already voted. The proposal for which the sender voted will have its vote count incremented.

2.4 DAO Attack

The DAO is a type of Decentralized Autonomous Organization deployed in the Ethereum blockchain in May 2016. A Decentralized Autonomous Organization is an organization whose processes and rules are encoded by a set of smart contracts, allowing shareholders control through a transparent mechanism, not subjected to the regulations imposed by a single country or governing body.

The goal of *The DAO*, in particular, was to allow direct venture capital funding by the Ethereum community into projects related with this platform. Its structure follows several

patterns of typical public traded companies, with decisions made by shareholders through voting. Since the source code was publicly available, anyone could audit the operation of *The DAO*.

Investors initially deposited funds and were given DAO tokens in return, which could be used to vote on proposed projects. The projects that were selected for funding would receive a certain amount of Ether, from the deposits of the investors. If those venture were successful, the investors would receive the profits according to the terms defined in the initial proposal for funding.

The DAO received a large amount of Ether from the Ethereum community - in May 2016, 14% of all Ether was held by *The DAO*. However, in June 16 2016, a bug was exploited, which led to the loss of a large portion of these funds held in *The DAO*. In total, more than $50 million dollars were stolen from the DAO - considering the relatively low Ether price of the time. The magnitude of the attack led to a relentless search for solutions, which eventually resulted in the proposal of a controversial hard fork to revert the attack.

This attack was possible due to a reentrancy bug in the DAO code - in simplified terms, the DAO smart contract performed a movement of funds which included sending a message to the sender smart contract, and only after this was performed, the actual balance of the sender funds were updated in the DAO. Since receiving a message triggers the fallback function in a smart contract, the sender (attacker) used this fallback function to call the original DAO function, leading to the movement of funds to be performed multiple times without an associated (and expected) removal of funds.

2.5 ERC-20

One of the main criticisms of design patterns, not necessarily related to those specific of Ethereum smart contracts, is that they tend to introduce complexity in the code base. Another common criticism, which compounds this first observation, is that the raison d'être of some design patterns is actually shortcomings in the targeted programming language/framework.

The security and financial concerns of code that is run in the Ethereum platform make smart contracts particularly sensitive to the correct use of design patterns, and the aforementioned concerns are being taken into consideration by the Ethereum Core development team, to enable a more consistent development experience.

ERC-20 is a proposal for a standard to the Ethereum platform, initially proposed in 19-11-2015. The goal of the ERC-20 proposal was to provide a standard API for tokens in the platform, thus guaranteeing a more consistent implementation of a common use case for smart contracts.

The standard API consists of six methods and two events (**Transfer** and **Approval**).

```
function name() public view returns (string)
(OPTIONAL): Returns the name of the token.

function symbol() public view returns (string)
(OPTIONAL): Returns the symbol of the token.

function decimals() public view returns (uint8)
```

```
8  (OPTIONAL): Returns the number of decimals the token uses.
9
10 function totalSupply() public view returns (uint256)
11 Returns the total token supply.
12
13 function balanceOf(address _owner) public view returns (uint256 balance)
14 Returns the total account balance of an account.
15
16 function transfer(address _to, uint256 _value) public returns (bool
       success)
17 Transfers an amount of tokens to an address.
18
19 function transferFrom(address _from, address _to, uint256 _value) public
       returns (bool success)
20 Transfers an amount of token from an address to another. This is used in
       an withdrawal workflow.
21
22 function approve(address _spender, uint256 _value) public returns (bool
       success)
23 Allows an address to withdraw from an account.
24
25 function allowance(address _owner, address _spender) public view returns
       (uint256 remaining)
26 Returns the amount of tokens of which an address can still withdraw from
       another address.
```

Chapter 3

Value Analysis

Value analysis is a fundamental tool to comprehend the impact of blockchain based applications and, in particular, how the development and systematization of design patterns can bring value to the customers. In this chapter, this value analysis will be presented, with emphasis on the concepts of value, perceived value and customer value.

As reviewed in chapter 2, the emergence of Bitcoin (and, in particular its innovative technical solution, the blockchain) revolutionized the concepts of online transactions by avoiding the need for a trusted peer to mediate these transactions. The incremental developments that originated other projects, such as the Ethereum framework, expanded on that application to enable full Turing-complete smart contracts, which can be enforced also without requiring a third party - other than the miners, for which the economic incentives makes the prospect of collusion for sabotaging the participants an irrational decision.

Even if the design of the blockchain provides some safeguards against exploitation, the same is not true for faults in the smart contracts code - these faults can be exploited in what is called an incentive compatible fashion (profitably), thus harming the interested parties in those contracts. The appropriate employment of design patterns can enable a more robust implementation of those contracts, thus avoiding possible attacks and inefficient execution resulting in high gas expenditure.

As proposed in [81], value can be defined as "need, desire, interest, standard criteria, beliefs, attitudes, and preferences".

The costumer for this solution are the Ethereum software developers and the users of the smart contracts, since these groups can derive value from a solution that enables more robust and efficient contracts.

3.1 Perceived value

The concept of perceived value is related to the user assessment of the utility of a product, based on the perception of what he receives and gives [82].

One of the goals of this dissertation is a smart contract which showcases the employment of multiple design patterns to achieve the goals related with security, upgradeability, user trust and transparency. This can be analysed as the "product" - the following advantages are granted:

Improved security Ethereum, which serves as the platform for the smart contracts that will be studied in this work, offers, by design, solid guarantees on the security of

transactions on this platform: in order to be considered valid, there has to be a wide agreement by the mining nodes; after approval, the transaction is forever stored in the Ethereum blockchain, on widely distributed nodes, and cryptographically linked to other subsequent transactions. Having established the security features provided by the Ethereum framework, the other remaining source of security issues is application bugs/failures. This will be addressed by appropriate use of smart contracts that will help to mitigate any potential bug in the developed smart contracts.

Enhanced Transparency due to the distributed nature of Ethereum, transaction information is distributed among a very large number of nodes. This characteristic enables a greater transparency than in traditional systems, reliant on a central authority to keep track of this same information - all the participants are able to fully consult the details of the transactions in the system.

Immutability once a transaction is commited to the Ethereum blockchain and the block is agreed upon a majority of nodes, the transaction is effectively immutable - while theoretically it is possible to revert it, the amount of resources that would need to be allocated for that endeavor should surpass the ones entire computational power of the current Ethereum network. Furthermore, the greater the amount of time passed since the transaction acceptance, the harder it will be to revert it.

Low Costs the removal of intermediaries/middleman from the process of mediating transactions, causes fees and overhead to be reduced which results in potential lower costs. Some caveats should be made in relation to this topic: transaction fees (in the form of gas) may be substantial, depending on the Ether value in the market and other factors that affect the mining economy. An eventual transition to a more efficient consensus method than Proof Of Work (such as Proof Of Stake) may alleviate this issue significantly.

A smart contract can be analysed as essentially providing a service. The characteristics of these services developed in the blockchain offer their own suite of strengths:

Always on Since the Ethereum framework supports the smart contracts, the service provided by these contracts is, by definition, always available - issues such as server downtime and deprecation/loss of support (unless it is intentional) are not concerns.

Reliable by employing the power of robust design patterns to their best effect, a robust solution is achieved.

Extensively Tested the service will feature extensive automated testing, which will allow increasing the user confidence.

Open Sourced by open sourcing the smart contract, the previous points are reinforced: user confidence in the contract can be backed by their own analysis of the source code or, alternatively, by analysis of reputed third parties.

The relationship between the user and developing party is also relevant for analysing perceived value. The following attributes can be identified:

Transparency by relying on the blockchain, all the concerns related with the service are fully auditable: the transactions that are processed, the business logic that handles those transactions and the parties that have administrative control (if any).

Anonymity while transactions in the blockchain are publicly available for anyone to check, the transaction trail only reveals Ethereum addresses, which can not be linked to anyone in particular and are randomised, since they are generated by cryptographic algorithms. No one except the owner of the addresses (and their funds) could identify a person by analysing the blockchain.

Security by employing well researched patterns, the user can be safe in using the developed contract.

Trustless no central authority (nor the developer) will be able to interfere with the contract logic, so not trust needs to be put on the developer nor any other third party.

Despite the large benefits identified in the last sections, some sacrifices are also present and worthy of consideration. These are:

High variability in Ether value while transactions costs are tendentiously low, and a contract may not be necessarily handling financial transactions, gas is still paid in Ether, whose underlying value (as measured against traditional currencies) is wildly volatile. This concern can be partly addressed (and with some cost) by hedging the held Ether against a currency such as USD, using available contracts in the Ethereum ecosystem for this end.

Technical Nature the technical nature of smart contracts limits the User Experience and, despite the proliferation of tools that foster adoption by less technically savvy users, some roadblocks still exist and some benefits are limited for this class of users.

Limited Scope due to the ambitious scientific goals of this work, the developed smart contract may have a limited feature scope, which could fail to provide enough value to attract users.

3.2 Fuzzy Front End, New Product Development (NPD) and Commercialization

As proposed by [83], the innovation process can be divided into three distinctive segments: fuzzy front end (FFE), new product development (NPD) and commercialization.

This is represented in figure 3.1.

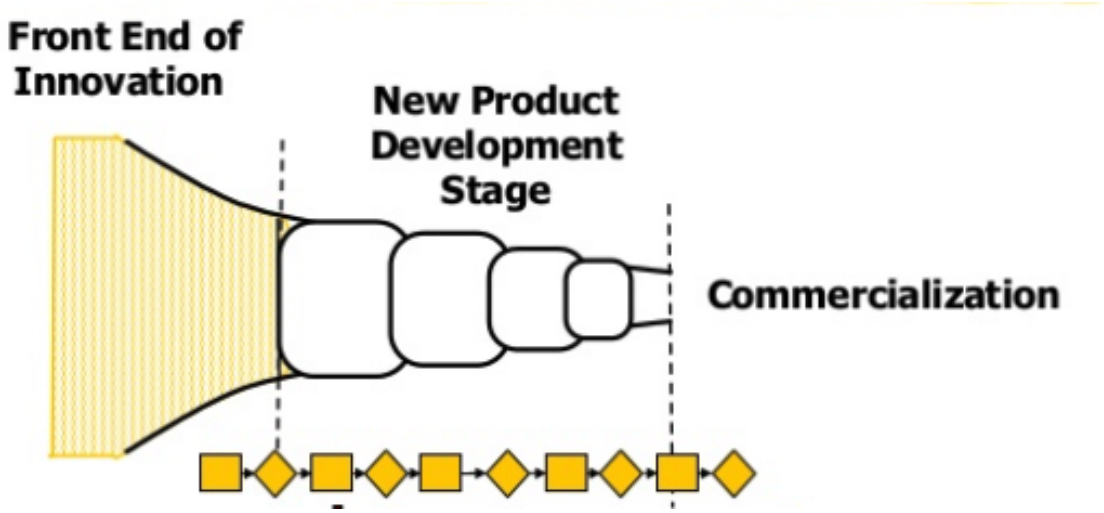

Figure 3.1: Representation of the three phases of the innovation process.

The differences between the fuzzy front end and new product development processes can be difficult to pinpoint. In table 3.1, these differences are summarized. In a basic definition, the FFE comprehends less formal activities, with a high focus on experimental work and unpredictable commercialization potential.

Table 3.1: Comparison between Fuzzy Front End and New Product Development phases.

Topic	**FFE**	**NPD**
Type of Work	Experimental	Goal-oriented, geared towards incrementalism
Commercialization	Uncertain	Expected
Activity	Heavy focus on research	Process development team
Funding	Highly unpredictable	Well modelled in a project plan

3.2.1 New Concept Development Model

The New Concept Development Model (NCD) supports the Fuzzy Front End phase of the innovation process, by providing nomenclature for defining the key components of the FFE. An overview of this model is represented in figure 3.2.

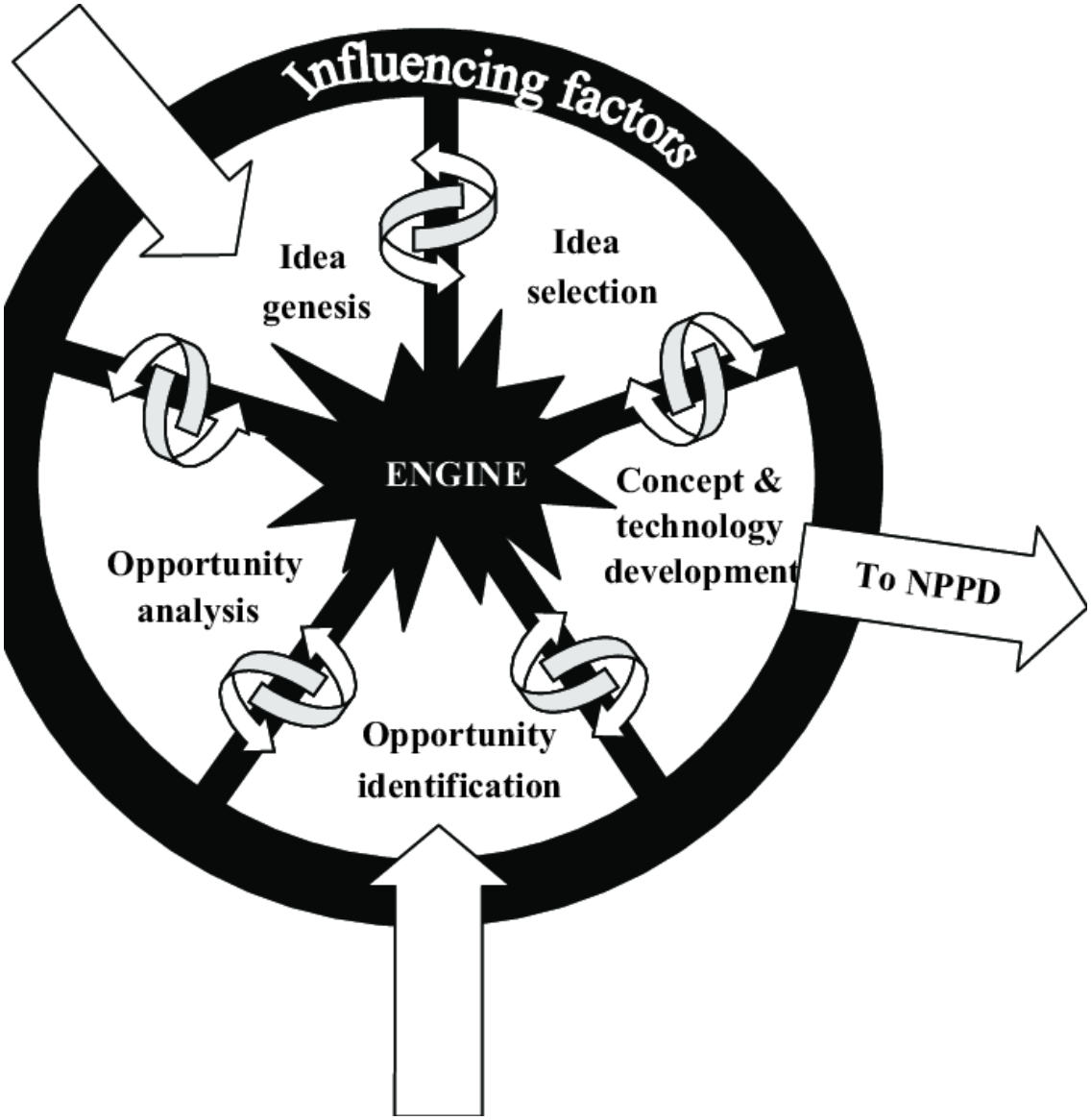

Figure 3.2: Representation of the New Concept Development Model.

A brief explanation of the model:

Engine the engine represents support by management and the organization where the project takes place.

Influencing Factors influencing factors can be be varied in nature, but they are typically associated with uncontrollable factors, such as the business environment and

Five Elements

The Five Elements of the Front End of Innovation are explained in more depth below:

Opportunity Identification in this element, opportunities that the organization wants to pursue are identified. There are several techniques that can be used to nurture the identification of new opportunities. These include ad hoc sessions, informal workplace discussions, individual contributions, etc.

Opportunity Analysis an opportunity is studied with the goal of understanding if it is worthwhile to pursue it. This involves research to guide the final decision. Techniques for these element include focus groups, market studies and scientific studies.

Idea Genesis this element is related with the birth, development and maturation of an idea. This process is intrinsically iterative and includes the involves direct contact with the customers and other stakeholders. Techniques for this activity include, for instance, brainstorming sessions.

Idea Selection selecting the appropriate ideas is a crucial endeavor. Formal decision making processes are generally difficult to employ due to the limited information available on the early stages of the innovation process.

Concept and Technology Development this is the final element, and comprehends the exit to the next stages of the innovation process - the New Product Development.

The application of the five elements to this work:

Opportunity Identification the existence of high profile exploits to Ethereum Smart Contracts, such as the DAO attack, as well as the environmental and economic consequences of ineffective Smart Contracts constitute an excellent opportunity.

Opportunity Analysis an analysis of the causes of most of the exploits points to bad software engineering practices and, often, to wrong use of design patterns.

Idea Genesis in order to promote the development of the Smart Contract ecosystem in Ethereum, currently valued in the tenths of billions of dollars, the adoption of better software development practices is fundamental. An alternative solution is to just bypass decentralized solutions and rely on traditional trust based transactions.

Idea Selection to improve upon the current standards in blockchain based software engineering is the favored approach. The most important factor for this decision is the great benefits that are uncovered by avoiding trusted middleman for peer to peer transaction and contracts.

Concept and Techonology Development comprehends the work to be developed in the context of this dissertation.

3.3 Choosing a Smart Contract Framework

In the early stages of this work, one of the main decisions to carry out was the choice of the Smart Contract framework to use. The choice is not simple, as it depends on multiple factors. Furthermore, this decision will have important implications in the work to be done.

Formal methods exist for decision making, which can be employed for this task. Analytic Hierarchy Process is one of such methods: this method admits the division of the decision problem in hierarchical levels, and allows the use of both quantitative and qualitative metrics in the evaluation process.

To use this method, the following phases should be followed:

1. Define the problem and structure it as an hierarchical diagram. This hierarchy should include, at least, an objective, criteria and alternatives.

2. Compare the alternatives and criteria, establishing priorities, by using a comparison matrix.
3. Obtain the relative priority of each criteria. From the priority of each alternative, relative to the different criteria, normalize these values and obtain the priority vector - by calculating the arithmetic mean of the values in each of normalized matrix rows.

The following criteria were identified as being meaningful for deciding on a framework:

Market Capitalization market capitalization can be used as a proxy for interest and potential of the framework.

Decentralization while all frameworks allow a degree of decentralization, some technical concerns could lead to different degrees of decentralization.

Transactions Per Second the maximum value of Transactions Per Second (TPS) that are capable of being processed is fundamental to understand the scalability of the solution. A very low TPS places a hard cap on the potential of a framework, especially if no solutions are envisioned for surpassing the limitation.

Smart Contract Features the capability associated with a smart contract - a framework that provides more features would enable more powerful smart contracts, which in turn could allow the development of applications for intricate use cases.

The alternative frameworks are:

- Bitcoin
- Ethereum
- EOS

The hierarchical diagram is represented in figure 3.3.

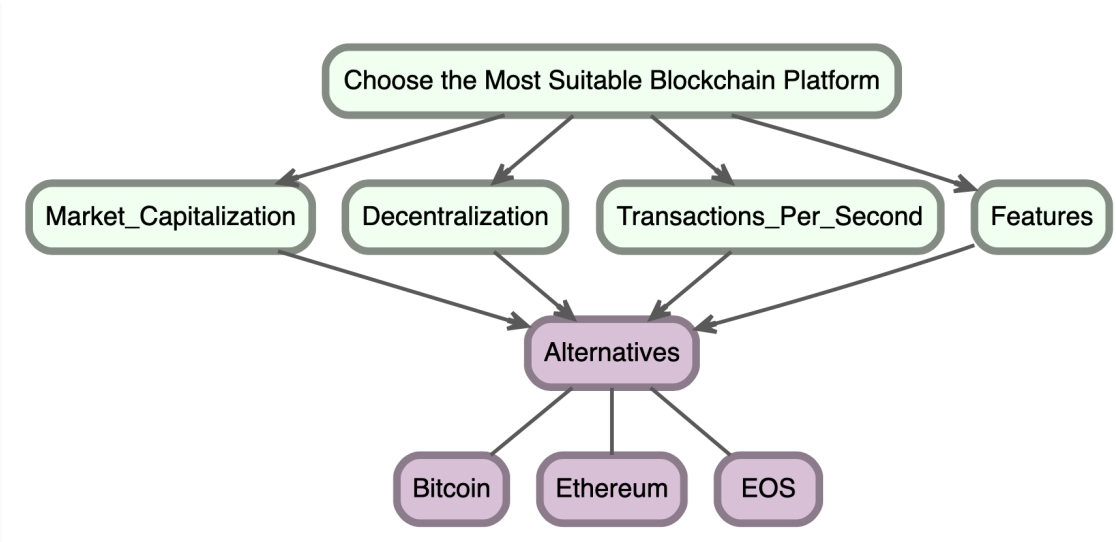

Figure 3.3: AHP hierarchic diagram.

3.3.1 Pairwise Comparisons

Having established the alternatives and the criteria, the next step is to perform pairwise comparisons between the alternatives, for each criteria.

Market Capitalization

Currently, by analysing the market capitalization of the alternatives, it is clear that Bitcoin has the highest, with around $70 billion dollars. Ethereum has around $15.5 billion dollars and EOS has $3.5 billion dollars of market capitalization.

The pairwise comparison for the different alternatives, and respective weights matrix, are shown in tables 3.2 and 3.3.

Table 3.2: Pairwise comparison of alternatives with respect to the Market Capitalization criteria.

Pairwise Comparison for Market Capitalization			
Bitcoin	5	Ethereum	1
Bitcoin	9	EOS	1
Ethereum	5	EOS	1

Table 3.3: Weights matrix for the Market capitalization criteria.

Market Capitalization	Bitcoin	Ethereum	EOS
Bitcoin	1	5	9
Ethereum	1/5	1	5
EOS	1/9	1/5	1

Decentralization

The analysis for decentralization is relative to the publicly deployed networks of Bitcoin, Ethereum and EOS. All of them allow anyone to join, but technical aspects create some limitations to the decentralization.

In the case of Bitcoin, the type of algorithm used for the proof of work (SHA-256) strongly favors the use of specialized hardware - Application Specific Integrated Circuits (ASICs) - that are orders of magnitude more effective than CPUs at this task. Since it is not expected that normal users will have access to this type of hardware, it creates some centralization in the mining power, which could make it easier to exploit for a 51% attack.

EOS uses a Delegated Proof Of Stake approach for its consensus mechanism. Each token holder will vote for a set of validators - a total of 21 will be chosen and these validators will be able to produce new blockchain blocks. Since only 21 validators do exist, collusion is a more threatening possibility in this framework.

Ethereum uses a hashing algorithm that is memory intensive (KECCAK-256), with the declared purpose of mitigating the use of specialized hardware. Its consensus is also proof of work, as with Bitcoin - this helps achieve a high degree of decentralization.

The pairwise comparison for the different alternatives, and respective weights matrix, are shown in tables 3.4 and 3.5.

Table 3.4: Pairwise comparison of alternatives with respect to the decentralization criteria.

Pairwise Comparison for Decentralization			
Bitcoin	1	Ethereum	3
Bitcoin	5	EOS	1
Ethereum	7	EOS	1

Table 3.5: Weights matrix for the Decentralization criteria.

Decentralization	Bitcoin	Ethereum	EOS
Bitcoin	1	1/3	5
Ethereum	3	1	7
EOS	1/5	1/7	1

Transactions Per Second

In relations to transactions per second, Bitcoin has a maximum of around 7 - this is related with the low frequency of block creation and the maximum size of each block (1 megabyte). Ethereum has a maximum transaction rate of around 15 transactions per second. EOS, by virtue of using a Delegated Proof Of Stake consensus mechanism, is able to achieve much higher transaction rates - sustained operation is possible in the thousands per second, and peaks were registered for TPS above 13000.

The pairwise comparison for the different alternatives, and respective weights matrix, are shown in tables 3.6 and 3.7.

Table 3.6: Pairwise comparison of alternatives with respect to the Transaction Per Second (TPS) criteria.

Pairwise Comparison for TPS			
Bitcoin	1	Ethereum	3
Bitcoin	1	EOS	9
Ethereum	1	EOS	9

Table 3.7: Weights matrix for the Transaction Per Second (TPS) criteria.

TPS	Bitcoin	Ethereum	EOS
Bitcoin	1	1/3	1/9
Ethereum	3	1	1/9
EOS	9	9	1

Smart Contract Features

The Bitcoin ecosystem, being developed primarily as a currency, offers a low number of features in respect to possible contracts being developed - the Turing Incomplete nature of the scripting that is possible to use in Bitcoin leads to very limited capabilities.

Ethereum and EOS both offer a Turing complete platform for smart contract deployment. EOS contracts are developed in C++ while Ethereum has several languages that can be used, with a comprehensive range of features.

The pairwise comparison for the different alternatives, and respective weights matrix, are shown in tables 3.8 and 3.9.

Table 3.8: Pairwise comparison of alternatives with respect to the Smart Contract Features criteria.

Pairwise Comparison for Features			
Bitcoin	1	Ethereum	9
Bitcoin	1	EOS	7
Ethereum	3	EOS	1

Table 3.9: Weights matrix for the Smart Contract Features criteria.

Features	Bitcoin	Ethereum	EOS
Bitcoin	1	1/9	1/7
Ethereum	9	1	3
EOS	7	1/3	1

Results

By using the results of the pairwise comparisons between the different alternatives, it is possible to make a decision.

The weights matrix for the different criteria are represented in table 3.10.

Table 3.10: Weights matrix for the criteria.

	Market Capitalization	Decentralization	Transactions Per Second	Features
Market Capitalization	1	1/2	1	2
Decentralization	2	1	2	1
Transactions Per Second	1	1/2	1	1/3
Features	2	1	3	1

The final result is represented in figure 3.4 - Ethereum reached the highest score, 45.9%, with Bitcoin score set at 30.2% and EOS score at 23.9%.

	Weight	Ethereum	Bitcoin	EOS	Consistency
Choose the Most Suitable Blockchain Platform	100.0%	45.9%	30.2%	23.9%	12.9%
Decentralization	32.3%	21.0%	9.0%	2.3%	5.6%
Features	27.0%	17.7%	1.5%	7.8%	6.9%
Market_Capitalization	25.5%	5.3%	18.7%	1.5%	10.1%
Transactions_Per_Second	15.1%	2.0%	0.9%	12.2%	11.7%

Figure 3.4: Results of the decision on the framework to use, using AHP.

Chapter 4

Design Patterns

In the realm of software engineering, design patterns are defined as generic and reusable solutions to common problems in software design. In this context, the work presented in [84] was particularly influential, introducing patterns such as the Iterator, the Abstract Factory, the Strategy, the Builder, the Singleton, etc. The work in [84] allowed to systematise the knowledge surrounding those design patterns - most of them were already being used at the time of publication, but by identifying the patterns by name, and by framing those design patterns in a problem / solution dichotomy, this work enabled the dissemination of their use, as well as facilitating communication in the cases involving the use of those patterns.

In this chapter, an analysis of some common design patterns for developing smart contracts in Ethereum is presented. In table 4.1, a summary of the different properties of those design patterns is represented.

Table 4.1: Overview of several design patterns.

Design Pattern	Problem	Solution	Applicable for	Disadvantages
Checks-Effects-Interaction	The transfer of control to an external contract has the potential to maliciously alter the state of the caller contract.	Limit the attack surface for external contracts after an external call (transfer for instance).	When a function will render execution control to another contract.	No impact on gas expenditure, but it is counter intuitive to perform the effects before the interaction.

Table 4.1: Overview of several design patterns.

Design Pattern	Problem	Solution	Applicable for	Disadvantages
Emergency Stop	The discovery of a critical bug in a contract could cause the loss of the held Ether, or other type of malicious exploitation.	Allow to disable a contract in an emergency situation.	Critical contracts (any financial contract could be in this category) where the ability to avoid losing funds in case of an uncovered vulnerability.	The employment of this design pattern in a contract could be abused by the owner (or another entity authorised to perform this stop), creating distrust for users of the contract - since, for instance, they could be deprived of withdrawing funds held in a stopped contract.
Speed Bump	Some actions can involve values or effects so serious that the consequences of a malicious attack can be devastating.	Delay the execution of irreversible or critical tasks.	To enable recovery after an attack, since the effects will not be consummated until the delay has passed.	The main disadvantage is related with the major impact on user experience - a benign user would have no alternative but to withstand the delay.
Rate Limit	Certain flows can be susceptible to being executed with a very high rate.	Limit the amount of times a task can be executed in a certain period.	Issuing tokens over a time period or limiting very high withdrawal frequency.	User experience can be impacted by the unrealistic setup of the maximum rate.
Balance Limit	Large amounts of Ether in a contract can pose liability issues.	Limit the amount of Ether that a contract can hold.	Alpha or beta version of some contracts are good candidates for this pattern.	By limiting the total balance a contract can hold, the utility that it can bring will also be limited.

Table 4.1: Overview of several design patterns.

Design Pattern	Problem	Solution	Applicable for	Disadvantages
Pull Payment	Sending ether to an address involves a call to that address, which can fail for multiple reasons.	Move the risks of transferring Ether to the user of the contract.	Handling multiple transactions in a single call or avoid the risks associated with ether transfer, with the rollback of all executed successful operations.	This pattern causes the addition of a step for some operations. Besides the harmful effects on user experience, this also can lead to increased costs for the user.
Oracle	Smart contracts have access to very little data outside the blockchain and are not able to query external sources for information relevant for their operations.	Query an external contract for the relevant information.	Any contract that requires access to information outside the blockchain, such as betting or hedging contracts.	The Oracle contract must be trusted to provide accurate information.
Automatic Deprecation	Once deployed, all contracts are forever in the blockchain.	Define an expiration time after which an operation will not be executed.	Contracts that have a time limited horizon, such as alpha or beta versions.	Litters the blockchain with unneeded contracts.
Data Segregation	Updating a contract is often impossible since the data is coupled with the operational logic.	Separate the operational logic and the data into separate contracts.	Updateable contracts.	A penalty in gas expenditure will generally be paid due to the increase in external contract calls.

Table 4.1: Overview of several design patterns.

Design Pattern	Problem	Solution	Applicable for	Disadvantages
Contract Relay	In case of updating a contract, users would not be able to immediately change to the updated version, especially if these users are other contracts.	Users should interact with the contract through a proxy, which will relays all requests to the most updated version.	Contracts which by their nature or complexity are expected to receive updates	This approach will cause some overhead in the number of contract calls, which will cause a cost in spent gas.

4.1 Checks Effects Interaction

4.1.1 Intent

Avoid the possibility of a called contract changing the state of the caller.

4.1.2 Motivation

Contracts in Ethereum can invoke other contracts, and send them funds. As a side effect, when a third party contract is called, the control of the execution is handed over to that contract. This contract, then in charge of the control flow, is able to execute any logic, including calling any other contract in the blockchain. This opens the possibility to perform a so called reentrancy attack, where the caller calls back the first contract. This can lead to undesired effects on the contract and change of state, all of this before the the first invocation is finished. For instance, by re-executing methods that are supposed to be called in a certain order and/or only once, the state of the original contract can be changed in ways that could create an advantage to the third party contract.

4.1.3 Applicability

This pattern can be used for the following cases:

- Protect a contract against reentrancy attacks.
- Avoiding calling a third party contract is difficult or not possible.

4.1.4 Participants

There are two participants in this pattern: the target contract (the one where the design pattern should be applied) and the caller (third party) contract.

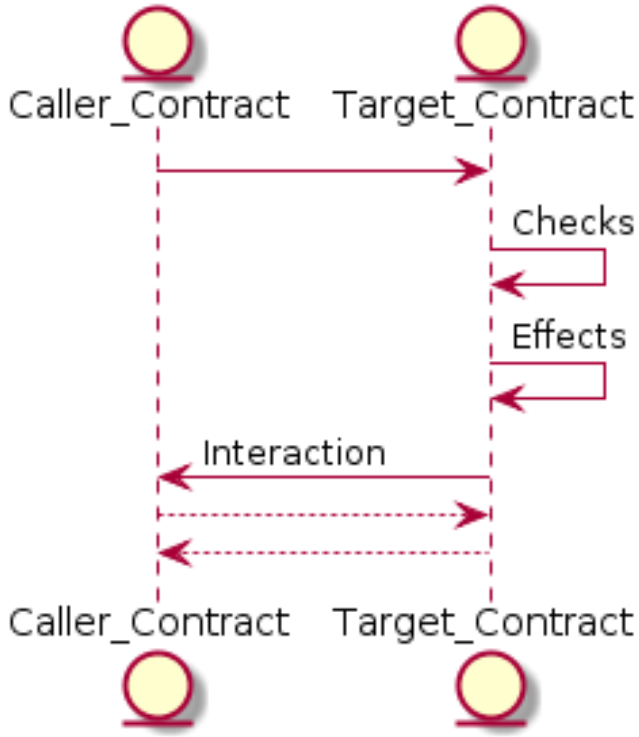

Figure 4.1: Sequence Diagram for Checks Effects Interaction design pattern.

4.1.5 Consequences

Implementing this pattern on a smart contract does not generally entail an overhead in terms of gas costs. However, it should be noted that the style of programming can be counter intuitive to programmers from a traditional background, where effects are generally only applied once the interaction occurs.

While sometimes implementing this pattern can be achieved by only altering the order of the code, it may also not be possible to achieve the desired effects in this way. It may also happen that using Checks Effects Interaction at the function level is not enough, as if the targeted function precedes other state changes, the contract may still be susceptible to reentrancy attacks. Other patterns, such as Mutex and Pull payment may be used for these cases.

4.1.6 Sample Code

```
1 pragma solidity >=0.5.0 <0.6.0;
2
3 contract ChecksEffectsInteraction {
4
5     address public owner = msg.sender;
6     uint256 public balance = 0;
7
8     function withdraw(uint256 amountToWithdraw) public {
9         // Checks
```

```
10          require(msg.sender == owner, "This operation can only be
    performed by the owner");
11          require(balance >= amountToWithdraw, "Amount to withdraw is
    greater than the available balance");
12
13          //Effects
14          balance -= amountToWithdraw;
15
16          // Interaction
17          msg.sender.transfer(amountToWithdraw);
18      }
19
20      function deposit() public payable {
21          balance += msg.value;
22      }
23  }
```

4.1.7 Known Uses

This pattern is widely used in several smart contracts. An example is the CriptoCountries contract [1]. This contract allows its users to "buy" countries in the world map, by doubling the current amount paid for each owner. The function *buy()* showcases the use of this pattern - all checks, logic and state changes occur prior to the transfer of Ether to the previous owner account - this transfer of Ether could pass the control of the flow to the called address, if it is a contract.

A notable lack of use of this pattern occurred in the DAO smart contract, whose attack is described in section 2.4.

4.1.8 Related Patterns

There are some patterns that can be used to address shortcomings of the Checks Effects Interaction pattern. Mutex (analysed in section 4.11) is a pattern that, when used, guarantees that reentrancy can not occur (or limit it, if designed to do so). Pull payment (analysed in section 4.6) is another pattern that can limit the attack surface by avoiding transferring the control flow to a third party contract when transferring funds. Instead, these funds would be assigned to the beneficiary, in the original contract and that entity should then proceed to withdraw them in a separate transaction.

4.2 Emergency Stop

4.2.1 Intent

Allow disabling contract functionality in case of an emergency.

[1] https://etherscan.io/address/0x17df117bb806a622d841bd5166a23b5d8746232f/#code

4.2.2 Also Known As

Circuit breaker.

4.2.3 Motivation

Deploying a smart contract to the main Ethereum network should only occur after extensive auditing to ensure that eventual bugs are not deployed, causing loss of users (or own) funds and loss of trust. However, given the evidence that bugs may still occur even with a large amount of resources and time dedicated to testing, steps should be taken to attenuate the effects of eventual flaws that are discovered. In the case of Ethereum, a quick fix to a smart contract is often impossible due to the inherent immutability of the smart contract code.

Using the Emergency Stop pattern, it is possible to pause the execution of critical flows of the smart contract, thus preventing possible exploits of a flaw affecting this critical flow.

4.2.4 Applicability

The following cases are suitable for the application of the pattern:

- ability to temporarily or indefinitely stop the execution of a smart contract.
- protect critical flows against possible exploits.

4.2.5 Participants

There are three participants that can be identified in this pattern. One of those is the contract where the pattern is applied (target contract), and whose functions should or should not be active depending on its state. The second are the regular users of the contract, which can use all the non administrative functionalities of the contract. The third entity is the administrator of the contract, which can trigger the emergency stop and also resume the contract.

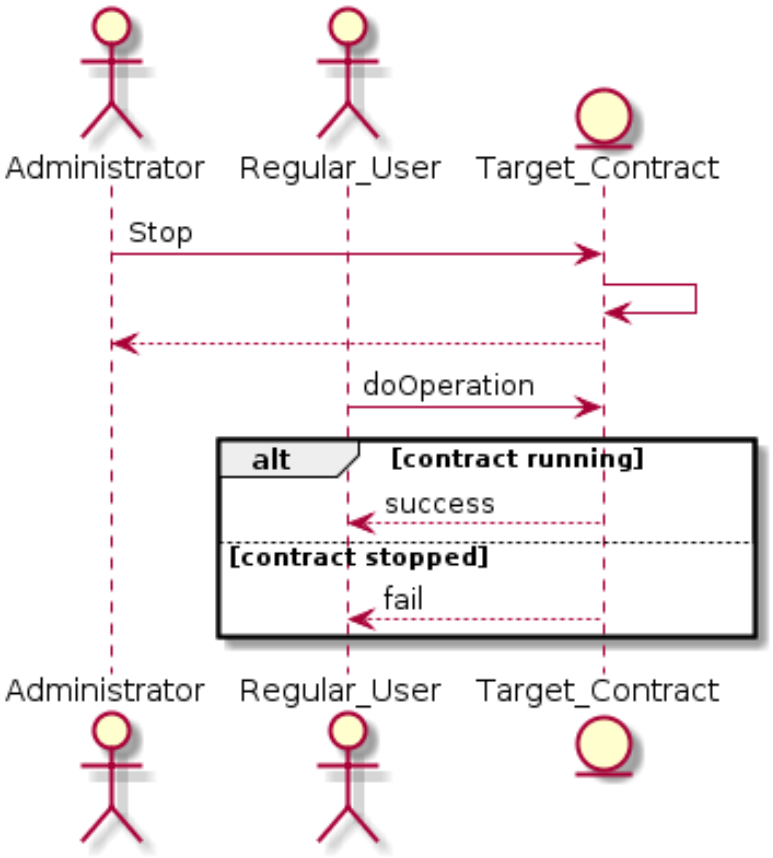

Figure 4.2: Sequence Diagram for Emergency Stop design pattern.

4.2.6 Consequences

While the emergency stop pattern offers an easy, reliable method of protecting against unforeseen faults in the contract code, it also adds a degree of unpredictability due to the fact that users need to trust the intents of the administrator - a malicious administrator could use this pattern to ransom the funds of the users. Given the *trustless* nature of the blockchain, requiring trust could be constructed as something to avoid.

The emergency stop pattern adds a negligible cost to the execution of the functions of a contract - the cost of checking the value of a state variable.

4.2.7 Sample Code

```
pragma solidity >=0.5.0 <0.6.0;

contract EmergencyStop {

    address public owner = msg.sender;
    bool isStopped = false;

    modifier notStopped {
        require(!isStopped, "Smart contract operations are stopped");
        _;
```

```
11      }
12
13      modifier onlyWhenStopped {
14          require(isStopped, "Smart contract is not stopped");
15          _;
16      }
17
18      modifier onlyOwner {
19          require(msg.sender == owner, "This operation can only be
    performed by the owner");
20          _;
21      }
22
23      function stopContract() public onlyOwner {
24          isStopped = true;
25          emit ContractIsStopped();
26      }
27
28      function resumeContract() public onlyOwner {
29          isStopped = false;
30          emit ContractIsResumed();
31      }
32
33      function performNormalOperation() public payable notStopped returns
    (bool) {
34          // perform operation logic
35          return true;
36      }
37
38      function emergencyOperation() public onlyWhenStopped {
39          // Emergency operation
40      }
41
42      event ContractIsStopped();
43      event ContractIsResumed();
44  }
```

4.2.8 Known Uses

The Emergency Stop pattern is implemented as part of the OpenZeppelin library [2], which offers several contracts that can be used to implement Ethereum standards and other general safe, audited snippets of code. The specific contract that implements this pattern is the Pausable contract and it is often used in deployed smart contracts.

One example is the OmiseGO contract [3], which aims to enable financial inclusion by providing bank-like services. This contract includes the Pausable implementation, and uses this pattern by having contracts extending Pausable.

[2] https://github.com/OpenZeppelin/openzeppelin-contracts
[3] https://omisego.co/

4.2.9 Related Patterns

The aim of the Emergency Stop is to provide a temporary safeguard against revealed faults. Since deployed contracts are immutable, in order to address these faults, it is often necessary to also implement some mechanism to allow upgradeability. A popular approach to achieve this is to employ the Contract Relay pattern, discussed in section 4.10.

4.3 Speed Bump

4.3.1 Intent

To delay the execution of important operations.

4.3.2 Motivation

Any operation performed in the blockchain is effectively irreversible and immutable. This is an important design decision for Ethereum, as it possibilities the existence of contracts between two entities that do not necessarily trust each other - without this guarantee, one of the parties could reverse any operation such as transfer of funds, voiding the guarantees of the contract.

While this characteristic is an important feature of Ethereum, it brings some potential problems. For instance, an attacker that manages to transfer funds to one of his accounts, leaves the victims without recourse, since its actions will be irreversible and the Ethereum framework does not incorporate tools such as blacklisting of accounts.

The Speed Bump pattern dictates that high value operations should only be effective after a certain "cool down" period, thus allowing reaction in case of an attack.

4.3.3 Applicability

This pattern can be used when:

- High value operations are allowed in a contract.
- High contract complexity may cause unexpected uses of certain operations.

4.3.4 Participants

There are two participating entities in this contract: the user and the target contract, which implements this pattern. The user initiates the interaction by requesting a certain operation to be performed. This order for the operation execution will then be stored. The user can then request the execution of the operation. If the pre-determined amount of time is elapsed, his requests will be fulfilled, otherwise they will fail.

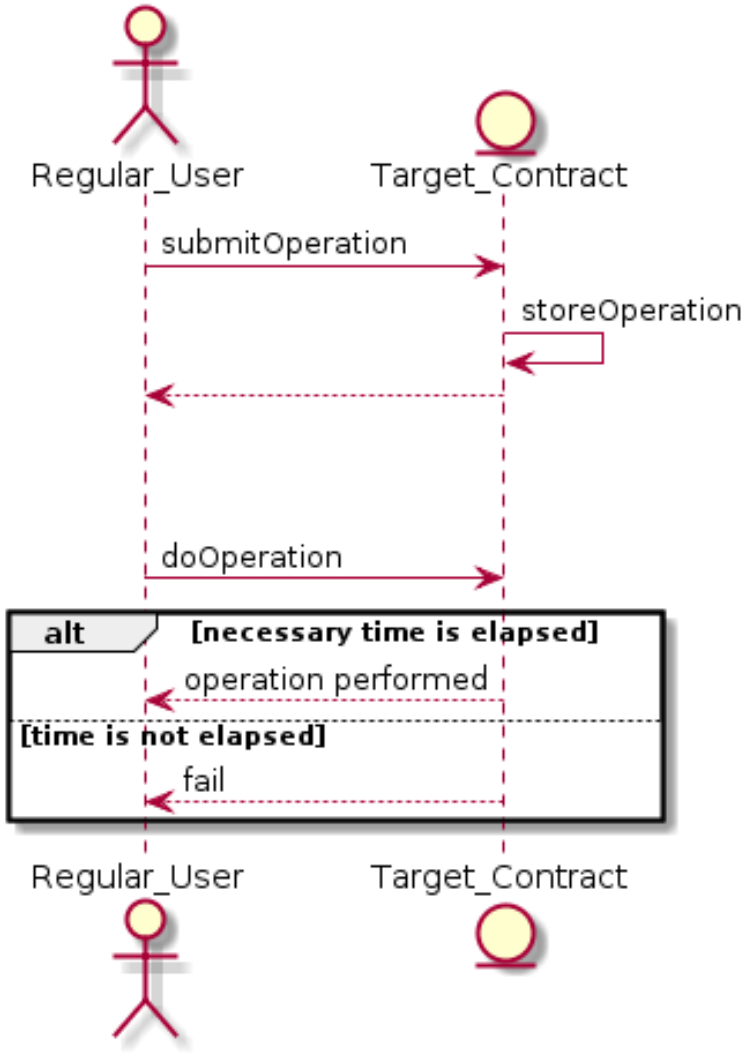

Figure 4.3: Sequence Diagram for Speed Bump design pattern.

4.3.5 Consequences

As a consequence of the employment of this pattern, the operations affected will be intentionally delayed. While the purpose of this delay may be justified by security requirements, it will also negatively affect user experience. In fact, the user will have to wait for an otherwise immediate operation. Also, for the operation to be performed, the user will need to initiate two transactions.

4.3.6 Sample Code

```
1 pragma solidity >=0.5.0 <0.6.0;
2
3 contract SpeedBump {
4
5     struct Request {
6         uint time;
```

```
7       }
8
9       mapping (address => Request) private requestedOperations;
10      uint constant withdrawalWaitPeriod = 7 days;
11
12      function requestOperation() public {
13          requestedOperations[msg.sender] = Request({
14              time: now
15          });
16          emit RequestedOperation(msg.sender);
17      }
18
19      function performOperation() public {
20          if(now > (requestedOperations[msg.sender].time +
        withdrawalWaitPeriod)) {
21              delete requestedOperations[msg.sender];
22              emit PerformedOperation(msg.sender);
23          }
24      }
25
26      event RequestedOperation(address addr);
27      event PerformedOperation(address addr);
28  }
```

4.3.7 Known Uses

The Speed Bump pattern was used in *The DAO* contract, well known for the attack it suffered (detailed in section 2.4). This contract required 27 days between a request for splitting the DAO and the effective performance of this operation. While this pattern was successfully implemented in *The DAO*, no provisions were made to allow to reverse the operation. As such, the attack was successful.

4.3.8 Related Patterns

In order to effectively halt attacks using this pattern, mechanisms should be in place to revert exploitative operations. One of the approaches to achieve this is the Emergency Stop pattern (analysed in section 4.2).

4.4 Rate Limit

4.4.1 Intent

Limit the amount of times an operation can be called in a specific time frame.

4.4.2 Motivation

A large number of requests in a smart contract may adversely affect its operational performance: a common case is when a large number of withdrawals are requested in a short time frame. The Rate Limit pattern applies, as the name suggests, a limitation to the number of invocations that a certain function receives in a specific time frame.

4.4.3 Applicability

This pattern can be used when:

- Operational requirements concern the rate of requests that certain (or all) functions receive.

4.4.4 Participants

There are two participants in this contract: the caller and the contract which exposes operations that are rate limited (target contract). The caller will initiate a transaction which will be executed by the smart contract. If a new transaction is required in an time period less than the stipulated by the rate limiting feature, the request will be denied.

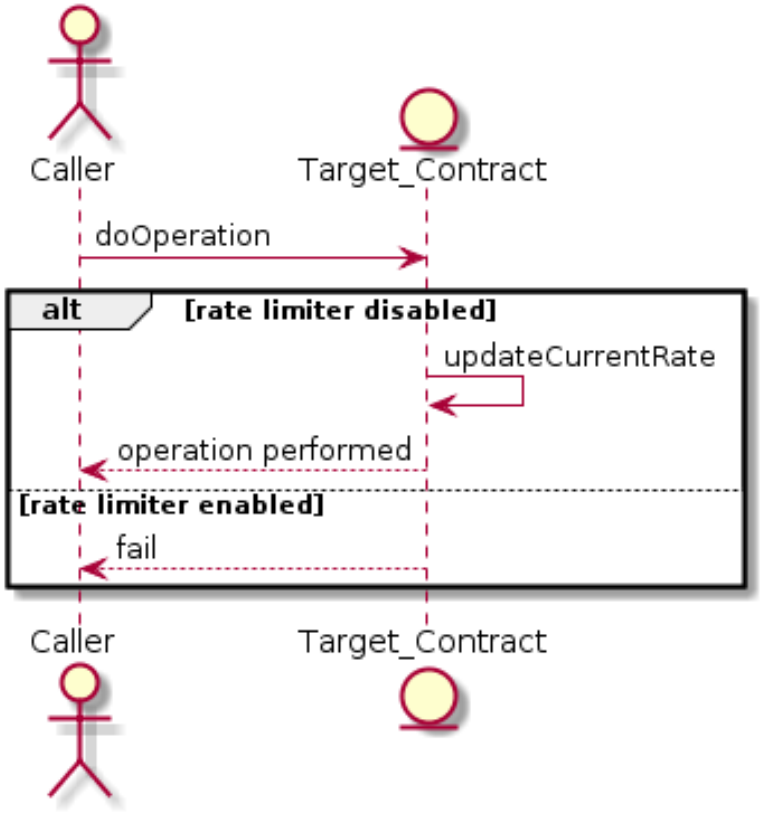

Figure 4.4: Sequence Diagram for Rate Limit design pattern.

4.4.5 Consequences

As a result of implementing the Rate Limit pattern, certain operations will be throttled, requiring a "cooling off" period. While this is useful for guaranteeing some operational requirements of the smart contract, it has negative effects in the user experience, since the user will need to wait for a certain period of time before executing the desired operation.

4.4.6 Sample Code

```
1  pragma solidity >=0.5.0 <0.6.0;
2
3  contract RateLimitByUser {
4
5    mapping (address => uint) enabledAt;
6
7    modifier allowOperationByUserEvery(uint t, address addr) {
8      if (now >= enabledAt[addr]) {
9        enabledAt[addr] = now + t;
10       _;
11     }
12   }
13
14   function rateLimitedOperation() public allowOperationByUserEvery(30
       minutes, msg.sender) {
15       // do something
16       emit PerformedOperation();
17   }
18
19   event PerformedOperation();
20 }
```

4.4.7 Known Uses

An example of the usage of the Rate Limit pattern can be verified in the Etherep project [4]. The Etherep offers a crowdsourced reputation system for the Ethereum platform. In line 70, the modifier delay() implements the rate limiting behaviour.

4.4.8 Related Patterns

Another pattern that can be used for dealing with serious operational abnormalities is the Emergency Stop, detailed in section 4.2.

[4] https://github.com/gointollc/etherep-contracts/blob/master/contracts/etherep.sol

4.5 Balance Limit

4.5.1 Intent

Manage smart contract risk by limiting the potentially exposed funds.

4.5.2 Motivation

When developing smart contracts, an universal concern is that large amounts of Ether in a contract can pose liability issues. This is compounded for cases where the developed code is not thoroughly tested.

The Balance Limit pattern can be used for these situations: this pattern proposes the implementation of mechanism to impose a maximum limit for the total balance that an account can hold. Before any deposit operation is performed, a check on the total balance is performed, rejecting new deposits if the current balance exceeds the threshold.

4.5.3 Applicability

The Balance Limit pattern can be used when:

- Developing alpha/beta versions of a contract.

4.5.4 Participants

There are two participant entities in this pattern: the user and the smart contract where the pattern is implemented. The user tries to deposit a certain amount of Ether in the smart contract. Internally, the contract checks if the limit would be surpassed if the deposit is accepted - in case it is, the deposit is denied.

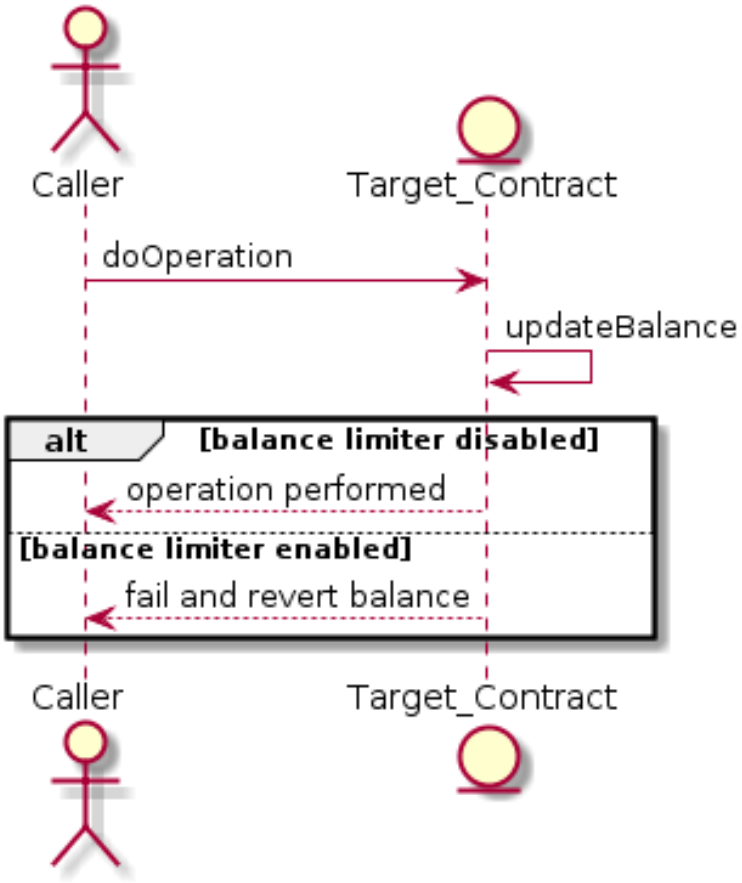

Figure 4.5: Sequence Diagram for Balance Limit design pattern.

4.5.5 Consequences

The Balance Limit pattern provides an effective and simple to implement mechanism to limit the liability in case a contract as critical flaws. However, it introduces a major constraint in the usability of these type of contracts, as they will have, by design, limited scalability.

4.5.6 Sample Code

```
pragma solidity >=0.5.0 <0.6.0;

contract BalanceLimit {
    uint256 public limit = 5000 wei;

    modifier limitedBalance() {
        require(address(this).balance + msg.value <= limit, "Contract
    balance would be exceeded after this operation");
        _;
    }

    function deposit() public payable limitedBalance {
        // do stuff
      }
}
```

4.6 Pull Payment

4.6.1 Intent

Decrease the risk associated with processing Ether transactions.

4.6.2 Also Known As

Pull over Push.

4.6.3 Motivation

Any Ether transaction from address to another requires a call to the receiving contract - this involves loosing the control over the flow of the execution, as discussed in 4.1.

Since the logic of the receiving contract is completely out of control for the caller, this poses some dangers: for instance, the receiving function may always throw an exception, thus causing the operation on the caller to always fail and any state changes to be reverted. While in most cases, intentionally causing an error when receiving Ether would negatively impact the receiver, this could also be used to freeze critical functionality in the contract.

Another potential issue is when a contract performs multiple transfers in the same transaction - any single failed transaction would cause all others to be reverted.

The pull payment patterns allows to address these issues, by isolating each external call from each other and also from the remaining state changing logic. Employing this pattern, successful execution of the contract logic can be effectively decoupled potentially failing calls to external entities.

4.6.4 Applicability

This pattern can be used for the following cases:

- avoiding the risk that ether transactions entails.
- safeguard critical flows that are prone to be abused by a malicious entity.

4.6.5 Participants

This pattern has three directly involved entities. One entity is the initial caller to the smart contract - the result of this call is that the contract (second entity) will update the currently owed balances so that a receiver will be able to trigger a payment to its address. This receiver will then initiate a second transaction, where it will request withdrawal of his funds.

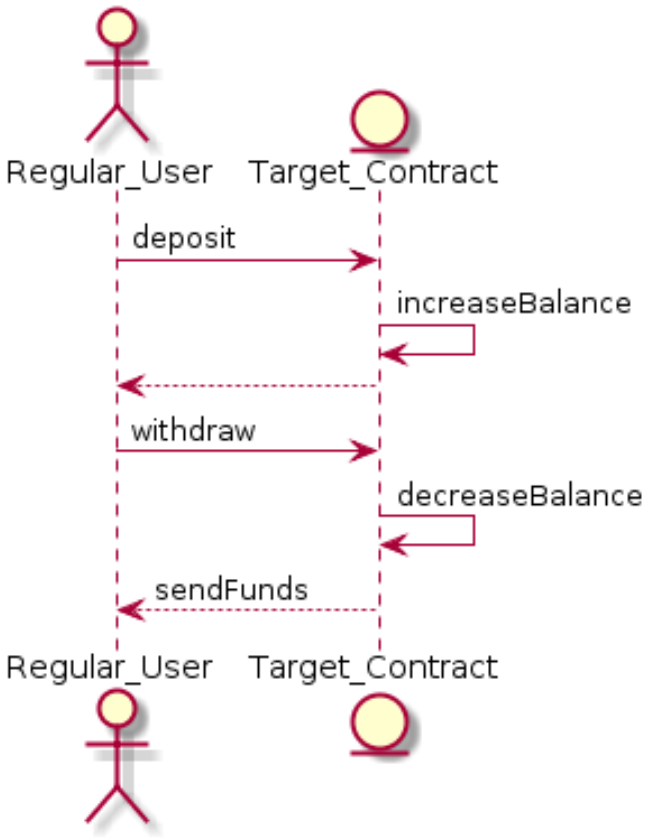

Figure 4.6: Sequence Diagram for Pull Payment design pattern.

4.6.6 Consequences

While the Pull Payment pattern offers significant safety advantages when transferring Ether in the context of a smart contract, it also brings some disadvantages that should be considered. The main disadvantage is related with the effect of this pattern on the user experience - the user needs to perform two transactions to withdraw their funds, which can be cumbersome, particularly for some type of contracts. This complexity is sometimes verified in the form of users not ever withdrawing their balances, reflecting the unintuitive nature of the user experience.

Another disadvantage is related with gas costs. Since the user needs to submit two transactions, gas costs are significantly higher than the alternative of only using a single transaction to perform the logic and the Ether transfer.

4.6.7 Sample Code

```
pragma solidity >=0.5.0 <0.6.0;

contract PullPayment {
    mapping(address => uint) refunds;

    function putPayment() public payable {
        refunds[msg.sender] += msg.value;
```

```
8           emit PutPayment(msg.sender);
9       }
10
11      function pullPayment() public {
12          uint refund = refunds[msg.sender];
13          refunds[msg.sender] = 0;
14          msg.sender.transfer(refund);
15          emit PaymentPulled(msg.sender);
16      }
17
18      event PutPayment(address target);
19      event PaymentPulled(address target);
20  }
```

4.6.8 Known Uses

An example of the usage of the Pull Payment pattern can be verified in the BlockPart contract [5]. BlockParty is a smart contract where users are expected to deposit a certain amount of Ether to reserve their place to attend a certain event. If the user ends up attending said event, its deposit is returned, otherwise, it is kept by the host.

In line 133 of the contract, the withdraw() function displays an example of this pattern. After performing some checks, the user deposit is sent back to him.

4.7 Oracle

4.7.1 Intent

Fetch information not otherwise accessible from the blockchain.

4.7.2 Motivation

The Ethereum Virtual Machine offers only a sandboxed environment in which computations are run. The implications of this is that several operations usually available to general computation are not possible in the Ethereum platform - namely those operations that require internet connectivity. There are several reasons for this design decision, one of which is the fact that data outside the blockchain does not offer the immutability guarantees necessary for the consensus mechanism to be employed.

Given these constraints, any data that needs to be made accessible to the blockchain should be provided directly, by means of a transaction, and stored in state variables. This data can then be accessed by other contracts or functionality on the platform.

External data is required for several relevant use cases, such as financial contracts, sports gambling, weather based applications, etc. An oracle acts as a bridge between the blockchain and the outside world, providing authenticated data for other contracts to rely on.

[5] https://github.com/makoto/blockparty/blob/master/contracts/Conference.sol

4.7.3 Applicability

The following cases are suitable for the application of the pattern:

- information outside the blockchain is required.
- the oracle can be considered as a trusted source.

4.7.4 Participants

There are three participants in this pattern. A contract that acts as a requester for information, the oracle and a data source entity that provides the data - the data source entity is external to the blockchain.

The requester starts by calling the oracle with a set of criteria for the information to be accessed. The oracle contract can work in various ways. In a simple implementation, it may already contain the information that the requester asked for and, in this case, it will relay this information to the requester - in this case, the data source entity should proactively insert the data in the contract. In another possible implementation, the oracle will actually not respond with the information immediately, instead creating a trigger for the data source to provide the information - this trigger will not be part of the blockchain operations. A callback will then be invoked on the requester, providing the required information.

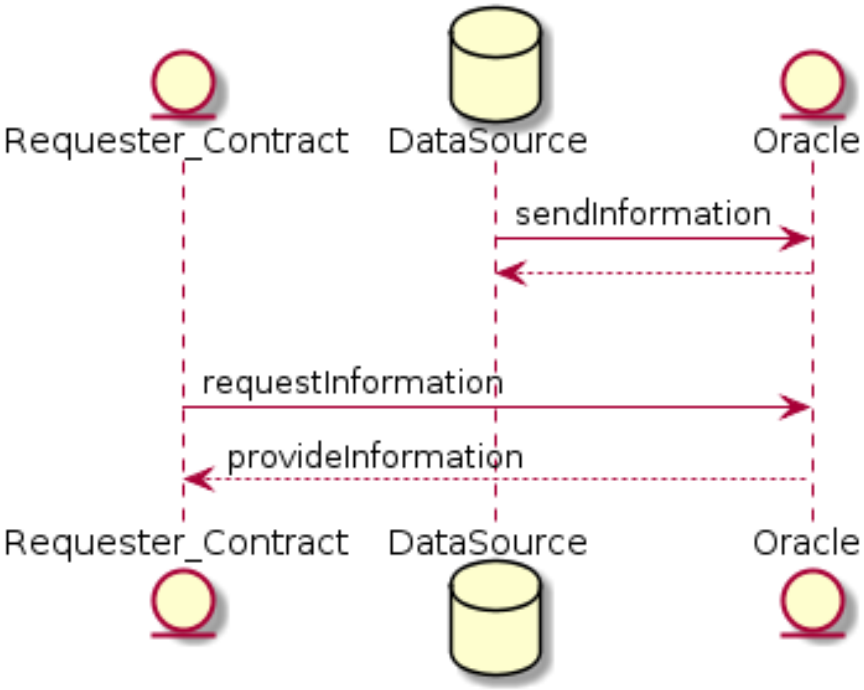

Figure 4.7: Sequence Diagram for Oracle design pattern.

4.7.5 Consequences

The use of the Oracle pattern offers an easy to use mechanism to provide outside data to smart contracts. However, relying on an Oracle smart contract introduces several risks. Any

user interacting with the requester contract would need to trust the accuracy of the oracle, thus introducing a "trust" requirement that is often frowned upon on the community.

An approach to deal with the requirement of trust is to use several independent oracles to provide the necessary data, employing an arbitration strategy to deal with incongruent data between them. A simple strategy could be to employ a voting scheme on the provided data, choosing the result with most votes. Alternatively, more complex approaches could include statistical analysis of the results. However, incrementing the amount of oracles used will also increase the gas costs associated with the contract operation.

4.7.6 Sample Code

```
pragma solidity >=0.5.0 <0.6.0;

contract WeatherOracle {
    address owner = msg.sender;
    int currentTemperature;
    bool isRaining;

    modifier onlyOwner {
        require(msg.sender == owner, "This operation can only be performed by the owner");
        _;
    }

    function setWeather(int temperature, bool raining) public onlyOwner {
        currentTemperature = temperature;
        isRaining = raining;
        emit WeatherEvent(temperature, raining);
    }

    function getWeather() public view returns (int, bool) {
        return (currentTemperature, isRaining);
    }

    event WeatherEvent(int temperature, bool raining);
}
```

4.7.7 Known Uses

A well known implementation of the Oracle pattern can be found in the FlightDelay contract [6]. This contract provides insurance for delayed flights, available to be subscribed by anyone. Since access to flight data is required for the operation of the contract, an oracle is used to fetch such data.

Whenever the user requests insurance for a flight, the request is relayed through the oracle, which will then provide risk parameters for the flight. Considering these parameters, the insurance could be accepted or denied. For the first case, a call to the requester contract is

[6] https://github.com/etherisc/flightDelay

scheduled for the date of arrival of the flight - this call contains information about the state of the flight and will trigger, if it is deemed that the flight is late, a transfer to the user.

4.8 Automatic Deprecation

4.8.1 Intent

Restrict the operational time frame of a contract.

4.8.2 Motivation

Certain contracts may expose operations which should not be available after a certain amount of time. For instance, token sales may provide special conditions for early buyers and limit sales after a predetermined time period. In other cases, test contracts may be deployed, which eventually should have their functionality turned off.

The Automatic Deprecation pattern can be used in this cases: this pattern considers that certain (eventually deprecated) operations should be guarded with a modifier, to ensure that they are not executed past a certain time.

4.8.3 Applicability

This pattern can be used when:

- test contracts are deployed.
- use cases with functionality that is not safe or should not be enabled past a certain time period.

4.8.4 Participants

There are two entities that participate in this pattern. The caller contract calls a function in the target contract, that contains this pattern. When the function is invoked, a check will be performed to ensure that the time threshold is not passed.

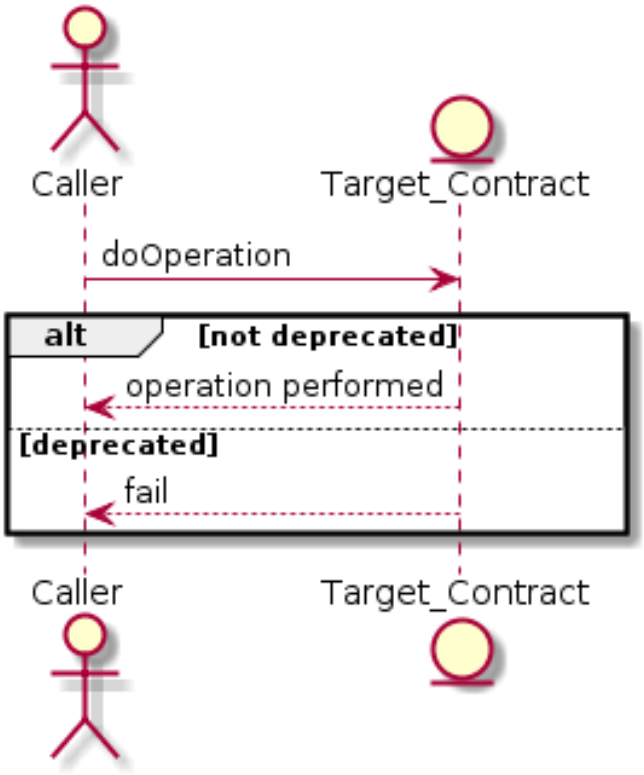

Figure 4.8: Sequence Diagram for Automatic Deprecation design pattern.

4.8.5 Consequences

The consequence of employing this contract is that some (or all) operations will not be available post some time point. While this is the intended behavior, this might limit the usefulness of a contract in certain situations. However, provided that the users are aware of this concern, its use is generally safe.

4.8.6 Sample Code

```
pragma solidity >=0.5.0 <0.6.0;

contract AutomaticDeprecation {
    address owner = msg.sender;
    uint activeUntil = now + 30 days;
    bool isRaining;

    modifier isActive {
        require(now <= activeUntil, "This contract is inactive");
        _;
    }

    function operation() public isActive {
        // Do something
        emit OperationExecuted();
    }

```

```
18      event OperationExecuted();
19  }
```

4.8.7 Known Uses

The Automatic Deprecation pattern is used in the Polkadot token sale smart contract [7]. The Polkadot project aims to provide a framework for interoperability between different blockchains, allowing the transfer of value and data. In order to ensure that state altering operations only occur in a specific time frame, the automatic deprecation pattern is applied.

A *when_active* modifier is used in most operations. This modifier ensures that the current time is not after a predetermined end date.

4.9 Data Segregation

4.9.1 Intent

Maintain contract storage separate from logic, thus allowing keeping it stable between contract upgrades.

4.9.2 Motivation

One of the challenges when performing upgrades of smart contracts is dealing with data storage migrations. Since upgrades (using the Contract Relay pattern, described in section 4.10) simply consist of deploying a newer version of a smart contract, without effective deprecation of the old version - which is, as with all Ethereum smart contracts, immutable-, all the contract state is kept in the old, legacy version.

Dealing with data migrations not only adds a significant development overhead in the deployment process, but can also be expensive, in terms of gas costs. In fact, storage operations are significantly costly in the EVM, which can be explained by the fact that all nodes of the network will have to store this data.

The Data Segregation pattern presents a solution to tackle these issues, avoiding complex data migration tasks during contract upgrades. This pattern establishes the need for a separate contract that holds all relevant contract state. This contract should be flexible, in order to accommodate unforeseen changes to the data structure.

4.9.3 Applicability

The Data Segregation pattern can be used when:

- the contract is upgradeable.

[7] `https://etherscan.io/address/0x54a2d42a40F51259DedD1978F6c118a0f0Eff078#code`

- data migration is deemed to be too complex or expensive.

4.9.4 Participants

There are three entities that participate in this pattern. The contract that actually implements this pattern stores all the data, as well as the address of the current active contract. Only requests that originate from that contract are resolved. An administrator is responsible for changing the active contract, which should be done at the same time of the change in the relay contract. Lastly, the contract that requires the storage is also a participant, querying the data storage contract whenever data is necessary for its logic.

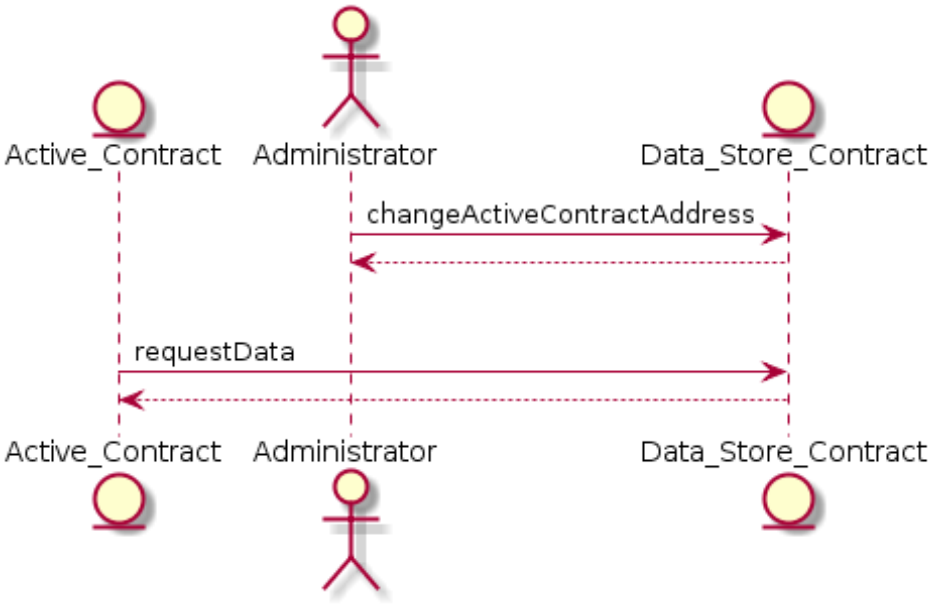

Figure 4.9: Sequence Diagram for Data Segregation design pattern.

4.9.5 Consequences

As a consequence of the employment of this pattern, data migrations following contract upgrades are avoided, which result in advantages in ether costs and deployment complexity. If the storage contract is built in that way, it can be flexible to support different data structures that were not initially conceived.

Some disadvantages should also be considered when deciding on the use of this pattern. Segregation storage and logic results in a more complex development, and often in more intricate code - primarily due to the need to perform external calls for each and every data request. As in the case of the Relay pattern implementation, trust is also required on the part of the users: a malicious administrator could manipulate the storage contract in ways that would cause loss of Ether for the users.

4.9.6 Sample Code

```
pragma solidity >=0.5.0 <0.6.0;

contract DataSegregationData {

    address dataSegregationLogicAddress;
    mapping(address=>uint) balances;
    mapping(address=>uint) accountNumbers;

    constructor(address _address) public {
        dataSegregationLogicAddress = _address;
    }

    modifier onlyOwner {
        require(msg.sender == dataSegregationLogicAddress, "This
    operation can only be performed by the data segregation logic
    contract");
        _;
    }

    function getBalance(address account) public view returns (uint) {
        return balances[account];
    }

    function getName(address account) public view returns (uint) {
        return accountNumbers[account];
    }

    function setBalance(address account, uint value) public onlyOwner {
        balances[account] = value;
    }

    function setName(address account, uint accountNumber) public
    onlyOwner {
        accountNumbers[account] = accountNumber;
    }
}
```

```
pragma solidity >=0.5.0 <0.6.0;

import "./DataSegregationData.sol";

contract DataSegregationLogic {

    DataSegregationData dataStorage = new DataSegregationData(address(
    this));

    function addBalance() public {
        dataStorage.setBalance(msg.sender, dataStorage.getBalance(msg.
    sender) + 1);
    }
}
```

4.9.7 Known Uses

The Data Segregation pattern is used, for instance, in the SAN Token project [8]. This project concerns the sale of tokens for a platform supplying financial data for cryptocurrency related assets.

For the sale of tokens, the main contract relies on several helper contracts, each holding specific pieces of data [9]. These contracts store the balances, the addresses of the buyers and min and max amount of tokens that can be bought.

4.9.8 Related Patterns

The Data Segregation pattern is deeply related with the Contract Relay pattern discussed in section 4.10. Both are widely used for allowing smart contract upgradeability, thus enabling correction of faults and improvement of functional behavior in smart contracts.

4.10 Contract Relay

4.10.1 Intent

Allow smart contract upgrades without breaking the functionality of dependents.

4.10.2 Also Known As

Proxy.

4.10.3 Motivation

One of the main constraints of Ethereum contract development is the inherent immutability of deployment - once any contract is deployed, it is kept without changes in the blockchain. However, this constraint causes significant hurdles for the developers, since any faults that are found will be, by design, unfixable. This can lead, in the worst cases, to loss of funds.

The impossibility to upgrade contracts also inhibits the reaction to different usage patterns and stales improvement of the whole smart contract ecosystem. In order to address these limitations, the Contract Relay pattern can be used. This design pattern uses the modularization of a smart contract into different entities to achieve effective mutability, even though all deployed contracts are kept in the blockchain.

4.10.4 Applicability

The following cases are suitable for the application of this pattern:

[8] https://docs.google.com/document/d/1yutgXroxwDC5wHGzSZsPZDTdDDqXVJXqtOA5lraTa1U/edit
[9] https://etherscan.io/address/0xDa2Cf810c5718135247628689D84F94c61B41d6A#code

- upgrading smart contracts is a requirement.
- there is a need to proxy calls to other contracts.

4.10.5 Participants

There are three participants in this pattern. The caller, the relay contract and the target contract. The caller only interacts with the relay directly, never with the target contract. Whenever a relay receives a call it will proxy that call to a configurable - by an administrator - address, which corresponds to the current target contract.

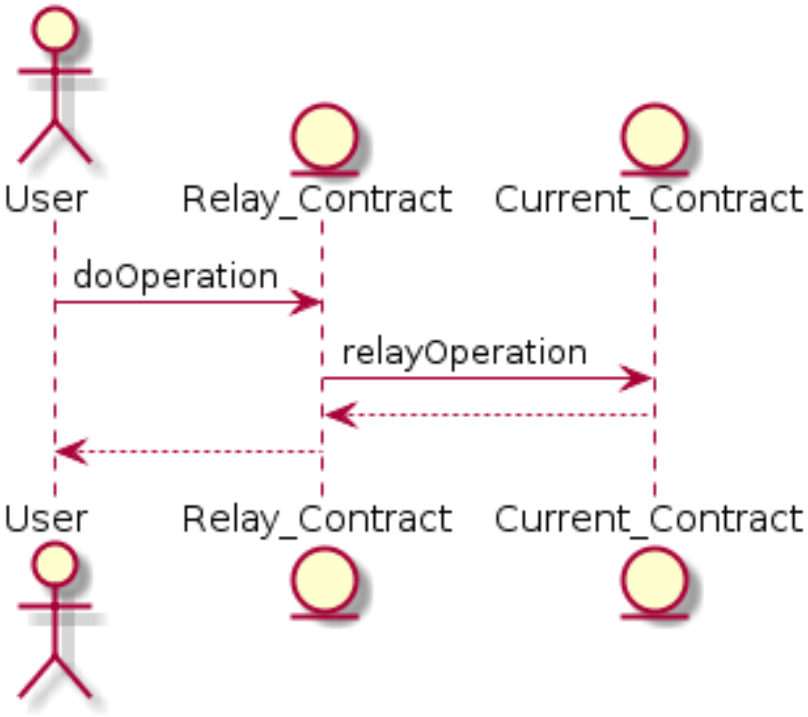

Figure 4.10: Sequence Diagram for Contract Relay design pattern.

4.10.6 Consequences

The Contract Relay pattern offers a straightforward mechanism to address one of the limitations concerning the development of complex use cases in the Ethereum ecosystem - the immutability of smart contract deployments.

As with other patterns, one of the disadvantages of this approach is that trust is required on the administrator, as the contract is effectively not immutable anymore and its functionality can be substantially altered by this administrator. In fact, it would be trivial for an administrator to alter a contract with user funds and change one of the operations to transfer ether to an address of his choice. To avoid full trust, a compromise could be to deploy immutable smart contracts that contain sensitive operations and high access level, and only consider for mutability other, less sensitive components.

4.10.7 Sample Code

```
pragma solidity >=0.5.0 <0.6.0;

contract Relay {
    address owner = msg.sender;
    address public currentVersion = msg.sender;

    modifier onlyOwner {
        require(msg.sender == owner, "This operation can only be performed by the owner");
        _;
    }

    function changeContract(address newVersion) public onlyOwner {
        currentVersion = newVersion;
    }

    function() external {
        (bool success,) = currentVersion.delegatecall(msg.data);
        require(success, "An error occurred while delegating the call");
    }
}
```

4.10.8 Related Patterns

As this pattern mostly addresses the immutability concerns of smart contracts, there are other patterns related to it. One of those is the Emergency Stop pattern, discussed in section 4.2. Emergency stop could be used as a temporary measure to avoid exploitation of a serious fault, while the Contract Relay pattern could be used to fix the root cause of the issue, thus restoring normal functionality.

The Data Segregation pattern, discussed in section 4.9 is also deeply associated with the Contract Relay pattern. It can be used to provide a clear, persistent, storage mechanism for different versions of a target contract, under a proxy.

4.11 Mutex

4.11.1 Intent

Eliminate the possibility of reentrancy attacks by disabling reentrancy.

4.11.2 Motivation

The possibility of reentrancy attacks, when an called contract recalls the original function is a serious issue that has already resulted in successful attacks. Some patterns exist whose major purpose is providing a defense against this type of attack vector: for instance, the Checks Effects Interaction and the Pull Payment pattern.

The Mutex pattern offers another strategy to avoid these type of attacks. In computer science, mutex is a technique to avoid simultaneous access to a shared resource, by using a flag to indicate if the resource is already available or not. The mutex pattern borrows from this concept, by proposing the definition of a state variable which is set when a flow that should not be reentrable is started - having this flag unset is a requirement to start this flow, so any attempt on reentrancy will not be allowed.

4.11.3 Applicability

The Mutex can be used when:

- the control flow should not simultaneously enter the same flow.

4.11.4 Participants

The mutex pattern has three participant entities: the target contract, that contains the mutex flag, the caller contract and an interacting contract. The caller contract will start the transaction by calling the target contract. Once this contract is called, its logic will be run, setting the mutex flag. An eventual interaction can occur, and the interacting contract will have control over the flow of the transaction - however, since the mutex flag is active, he will not be able to recall the original flow of the target contract, thus avoiding the reentrancy attack vector.

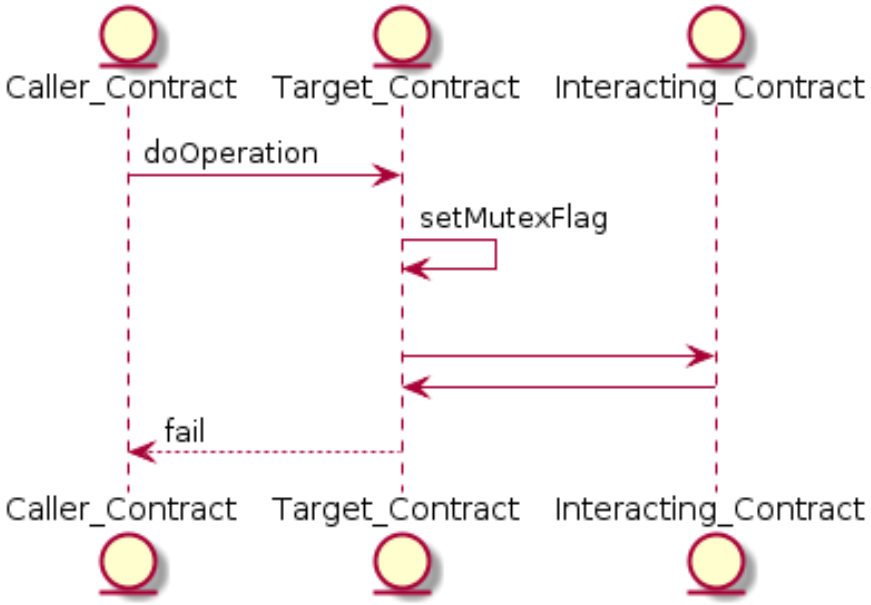

Figure 4.11: Sequence Diagram for Mutex design pattern.

4.11.5 Consequences

The mutex pattern guards critical flows against reentrancy. This can be extremely useful to safeguard contracts against a very common type of attack, whose safety analysis tends

to be non trivial. However, by limiting reentrancy, legitimate development options can be affected: for instance, no recursive calls would be possible.

The affects of the mutex pattern in gas costs are mostly negligible - the overhead is limited to storing an extra boolean field and writing to the field twice, per operation (setting and unsetting the flag).

4.11.6 Sample Code

```
pragma solidity >=0.5.0 <0.6.0;

contract Mutex {
    bool locked = false;

    modifier avoidRecursion() {
        require(locked == false, "Reentrancy detected");
        locked = true;
        _;
        locked = false;
    }

    function doSomething() public avoidRecursion {
        // Do something here
      }
}
```

4.11.7 Known Uses

The Mutex pattern is used, for instance, in the Veredictum project [10], specifically on the smart contract for the Ventana token. The goal of Veredictum is to provide a platform to protect copyrights of film and video producers, as well as to offer new distribution channels.

In the Ventana token sale smart contract, all public mutating functions are executed under a mutex. In the contract ReentryProtected, two modifiers (preventReentry and noReentry) and a boolean flag are defined, to implement the desired behaviour.

4.11.8 Related Patterns

Patterns that are related to the Mutex include the Checks Effects Interaction, discussed in section 4.1, and the Pull Payment, discussed in section 4.6.

[10] https://www.veredictum.io/faq

Chapter 5

Betting On The Block dApp

As a platform to showcase the appropriate use of several design patterns in a real context, the development of a decentralized app was devised - the *Betting On The Block* dApp. This application, which is described and analysed in this chapter, has the goal of providing a platform for betting, along with all the operational concerns that are required for this main use case. Since the Ethereum platform provides support for currency transactions, along with features that allow easy access control, this goal can be relatively simple to achieve.

The use cases of the *Betting On The Block* dApp are illustrated in the use case diagram of figure 5.1.

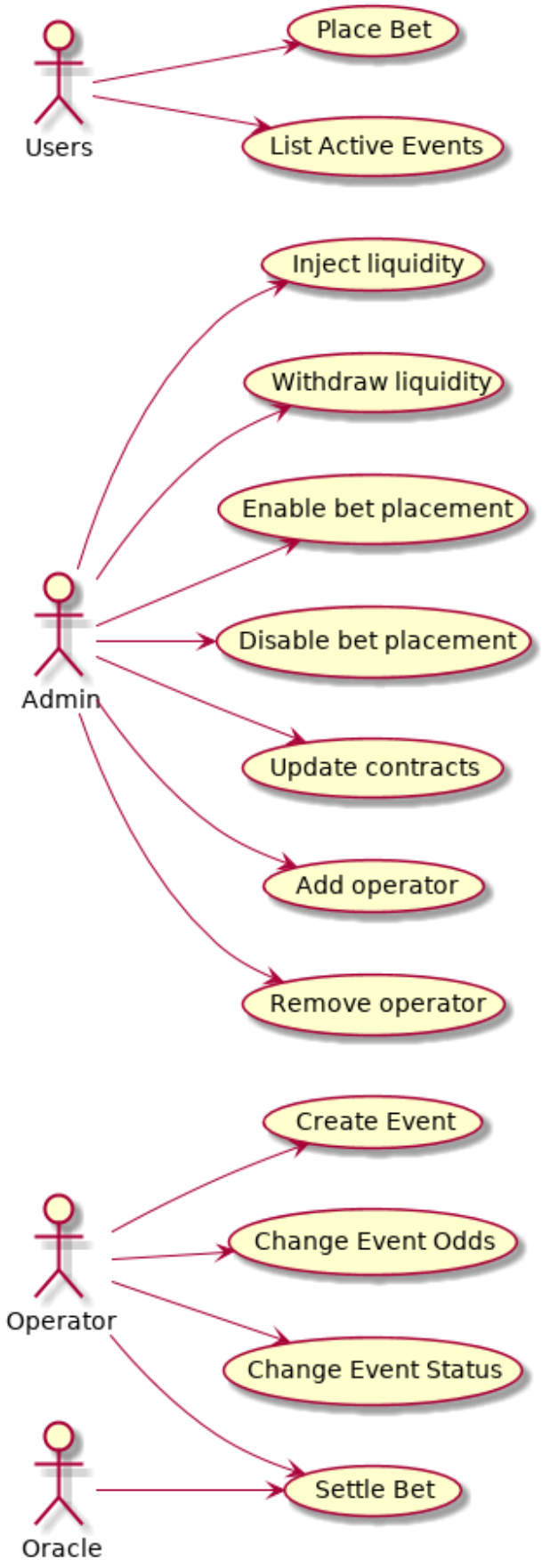

Figure 5.1: Use case diagram of *Betting On The Block* dApp.

An user is able to view multiple events and bet in their outcome. An event can be anything, but it generally would correspond to some type of sport event.

For the user to be able to bet in the outcome of an event, he should have the expectation to win some amount of Ether if his selection is correct, and loose the amount of the bet if his selection is not correct. The multiplier that determines the amount the user can win is called the **odd**. For instance, if the user bets 5 wei in an outcome with odd 3.0, he would receive back 15 wei (including his stake) if he won. Otherwise, he would loose his 5 wei stake.

The proposed smart contract should mediate the placement and settlement of bets. In order for the user to receive Ether in case his selection is correct, someone should inject liquidity. This is the responsibility of the contract administrator. Once a user places a bet, his stake is withheld until the event is settled; the same happens to the liquidity required to pay off his potential earnings. For a bet of 15 wei at 3.0 odds, a total liquidity of 10 wei is required (the stake itself is only returned, not part of the earnings).

The main goal of the contract, as a mediator, is to ensure that the user should not trust the administrator of the contract: by design, the user balance and bets should be safe due to the guarantees provided (again, by design) by the contract. This concern is a primary driver for decisions taken in the implementation.

5.1 Architecture

In order for the *Betting On The Block* dApp to comply with the proposed requirements, the architecture represented by the class diagram in figure 5.2 was used.

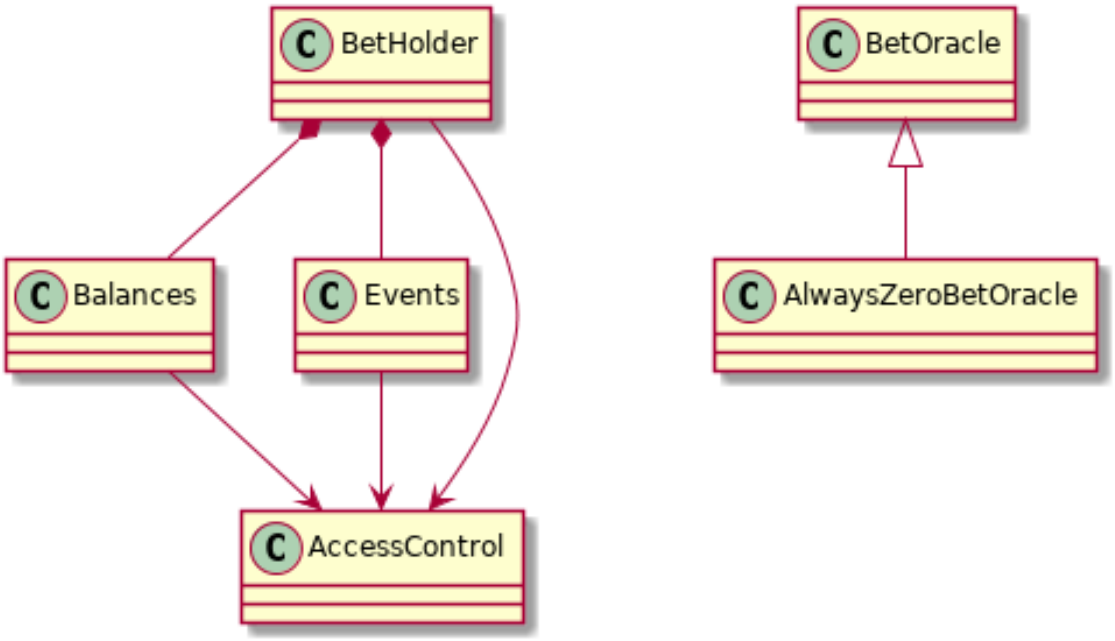

Figure 5.2: Class diagram of *Betting On The Block* dApp.

A total of six contracts are considered:

- AccessControl
- Balances
- Events
- BetHolder
- BetOracle
- AlwaysZeroBetOracle

The AccessControl contract holds the administrator and the operators addresses. It also provides several methods to allow verification if a certain address is an operator or an administrator. Also, it allows the administrator to insert a new address of an operator and to remove an address from the operators list.

The Balances contract holds the current balance of each user of the *Betting On The Block* dApp, as well as the total liquidity available for betting. It provides several methods, to allow a user to deposit and withdraw from their balance, to allow adding liquidity as well as withdrawing this liquidity - this last operation is only available to the administrator. Other operations for incrementing and decrementing account and liquidity balances are available, but can only be called by the BetHolder contract.

The Events contract holds all the information about the events that an user can bet on, including details from these events and the odds of each selection. It provides operations for adding new events, changing selection odds, closing events - all these can only be used by operators - and fetching several event related informations.

The BetHolder contract holds all the bets placed in the dApp. It allows users to make bets and to settle event results. Also, it provides several methods for retrieving user bets and the bets places in a specific event.

The BetOracle contract is responsible for providing information about the winner of an event. The address of the BetOracle should be provided at the time of event creation, by the operators, and cannot be changed once an event is created. This contract provides methods for verifying if a winner is already set and for setting a winner - only available to the contract owner - which preferably should be a third-party, not an administrator or operator of the *Betting On The Block* dApp. Once set, the winner cannot be changed.

The AlwaysZeroBetOracle contract inherits from the BetOracle contract, but considers the result to be already set upon creation, and the winner will always be the selection with index 0. This contract is used for testing purposes.

The full code for these contracts is available in annex A.

5.2 Flows

5.2.1 End-to-end

A full end-to-end flow of this contract is represented in figure 5.3.

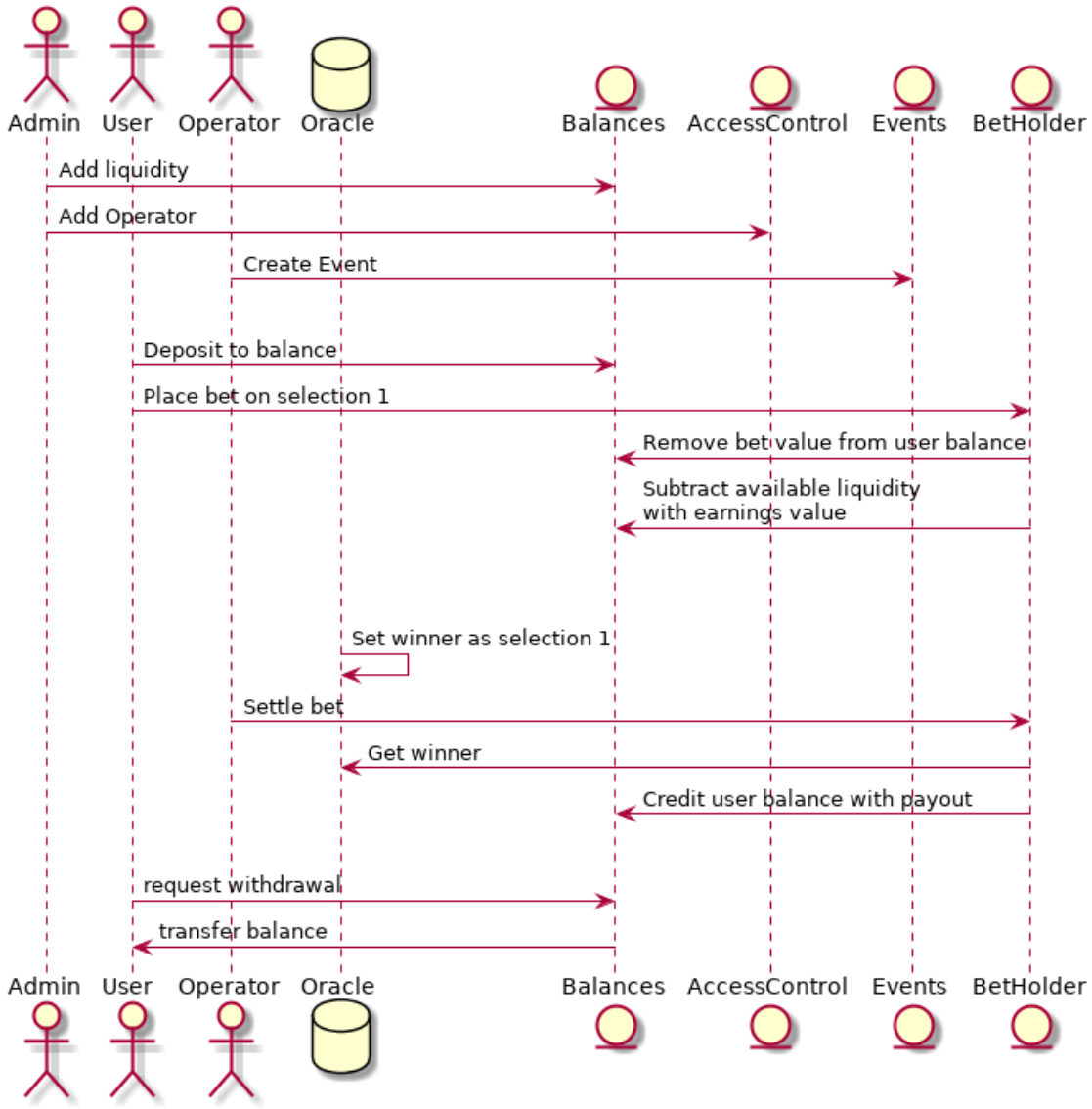

Figure 5.3: Sequence diagram of end-to-end flow of *Betting On The Block* dApp.

The administrator starts by adding liquidity to the *Betting On The Block* dApp. Without liquidity, the user is not able to place a bet, as there is no Ether to use in order to pay his potential earnings. All earnings from a user are limited by the liquidity available in the Balances contract.

The administrator will then set an address as an operator, thus enabling this address to perform several authenticated functions on the dApp. The operator will create an event, specifying the selections available for this event, and their respective odds, as well as the oracle that will supply the data of the selection winner.

An user can now place a bet, but first a deposit should be performed, otherwise any bet placement will fail due to lack of balance. After the outcome of the event is known, the oracle will update the winner selection. The bet can now be settled by anyone, but this task should fall upon the operator, preferably. The rationale behind allowing anyone to settle

an event is to protect the user against potential abuses by the dApp owners, where these owners would block some bet settlements that would result in large losses.

The settlement of the bet will trigger the update of user balances for the users that place a bet on winner selection, and the increment of the liquidity for the remaining bets. The user will then be able to withdraw all the earnings reflected in his account balance.

5.2.2 Odd changed after bet placed

The sequence diagram representing the result of a bet settlement with a user placing a bet before the odd is changed is represented in figure 5.4.

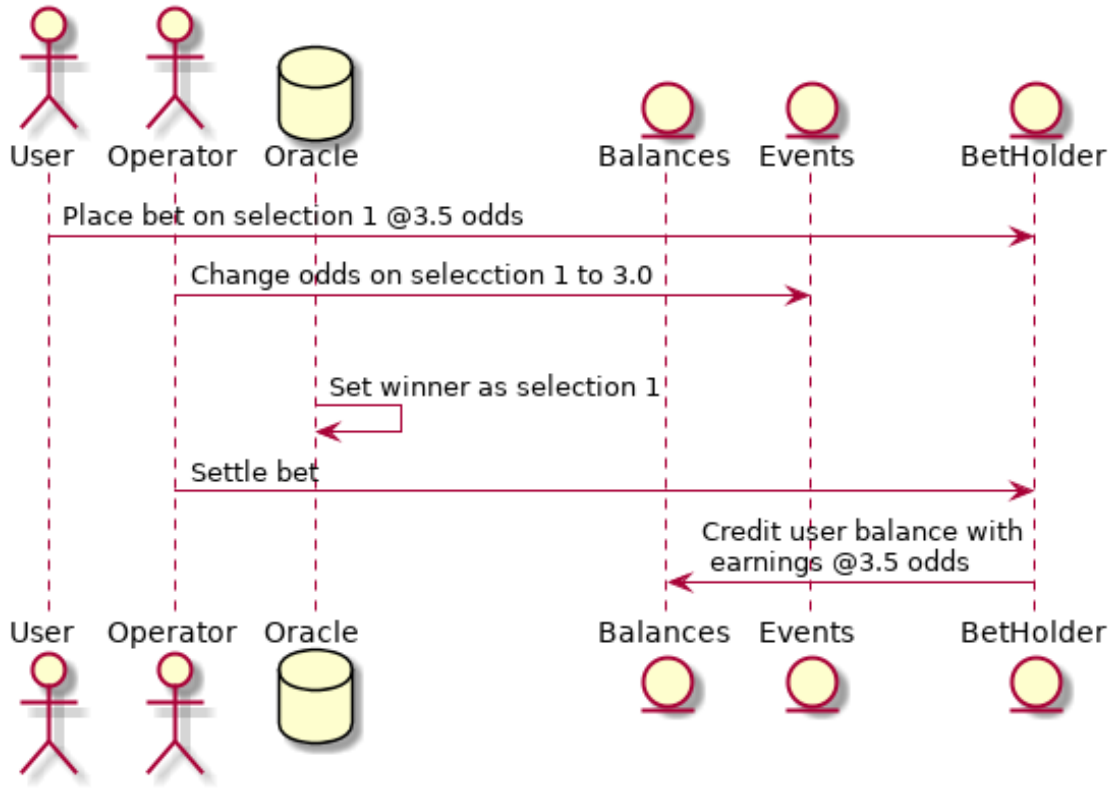

Figure 5.4: Sequence diagram of a flow where a bet is made prior to a change of odds.

As can be observed, the user will receive the payout with the original odd he bet on (3.5). In fact, the new odd set by the operator will only be used for new bets.

5.2.3 Stop BetHolder contract

The sequence diagram for the flow where an administrator stops the BetHolder contract is shown in figure 5.5.

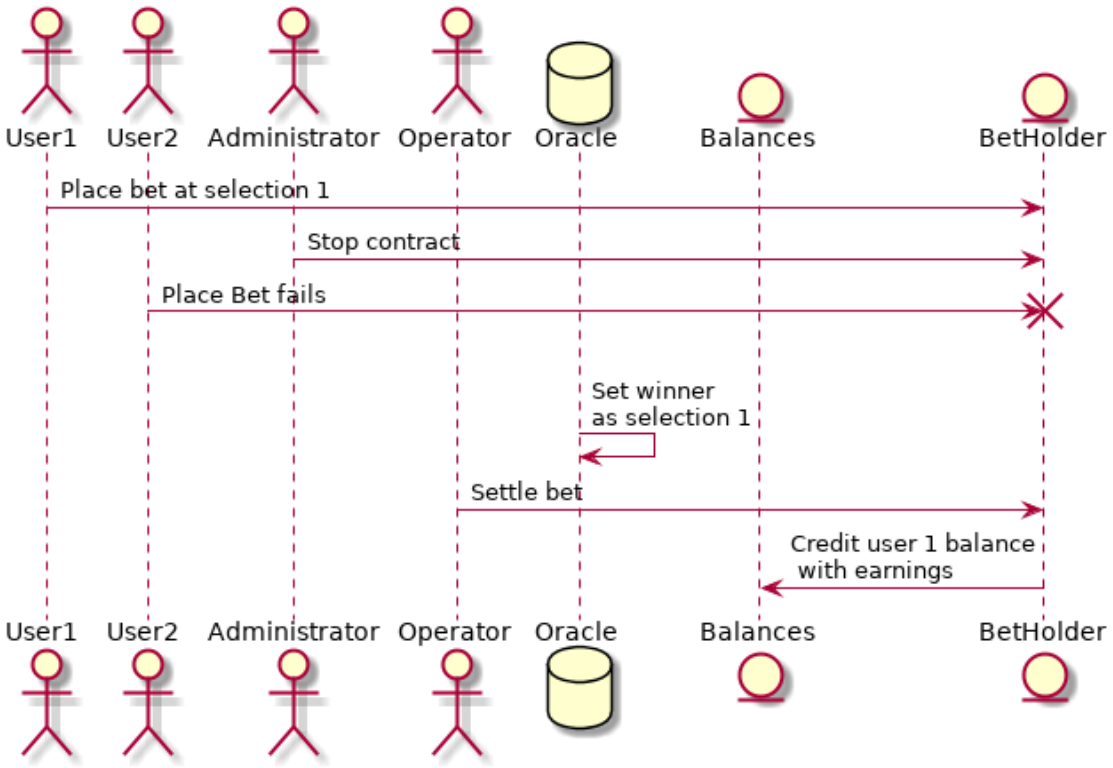

Figure 5.5: Sequence diagram of a flow where the BetHolder contract is stopped.

As can be observed, User 1 is able to place a bet normally. However, after the administrator stops the contract, User 2 is not able to place a bet anymore. It should be noted however that settling an event is still possible, even in the stopped contract - in this way, the user is protected against abuse by the owners of the dApp.

5.2.4 Close Event

The sequence diagram for the flow where an operator closes an event is shown in figure 5.6.

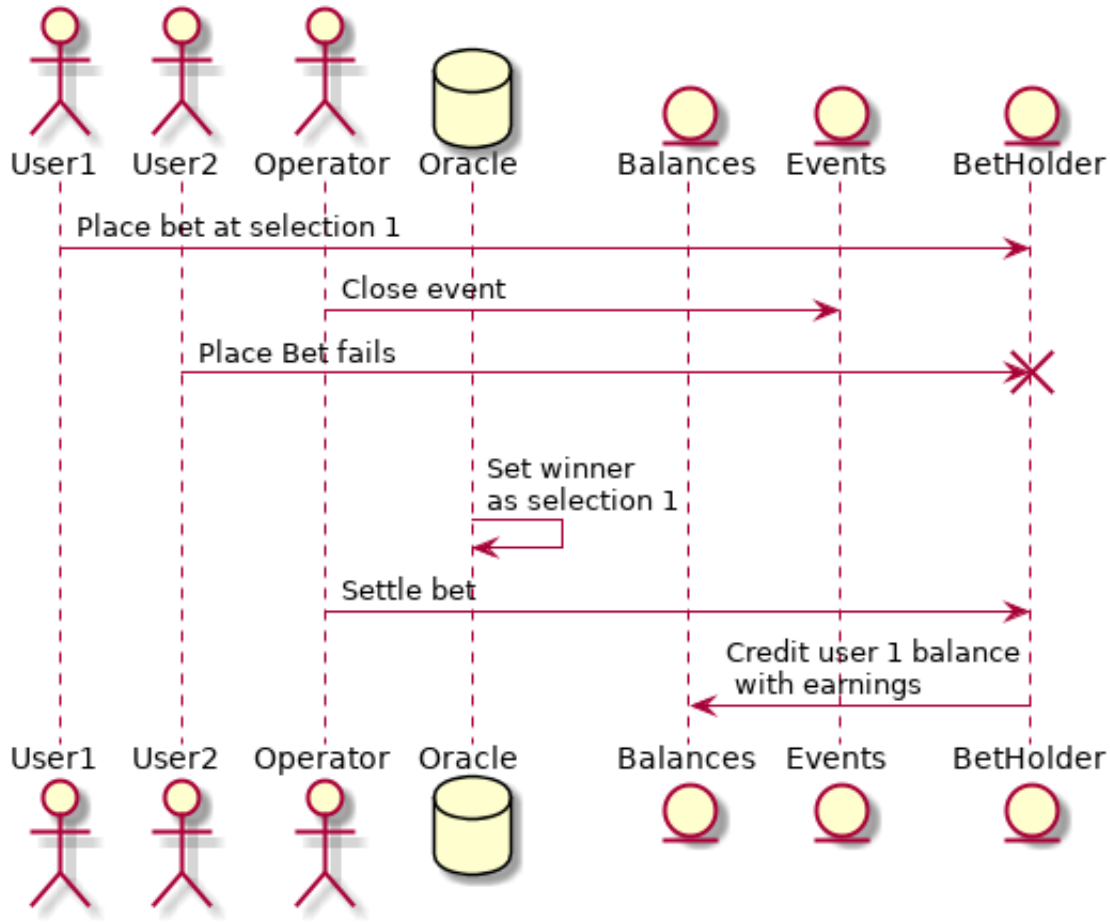

Figure 5.6: Sequence diagram of a flow where an Event is closed.

Closing an event could be used for cases where betting on the event would not be appropriate, but the result is not yet known. For instance, when a football game has already started, further betting should not be allowed, as this could allow for easy to abuse betting opportunities.

5.3 Applied patterns

In then context of the development of *Betting On The Block*, several design patterns were used. This section provides an overview of the motivation for their employment as well as the advantages and caveats of the implementation.

5.3.1 Checks Effects Interaction

There is only three single occurrences where the *Betting On The Block* dApp delegates the control flow to an external account/contract - when it sends balance to an user, when it transfers liquidity to an administrator and when it retrieves the event winner information from the Oracle. Other calls are all made to other contracts in the dApp.

In order to deal with first two instances of delegation of control, the Check Effects Interaction pattern is applied, for the Balances contract. As such, the transfer of control is performed as the last operation in each method call:

```
function withdrawLiquidity(uint valueToWithdraw) public onlyAdmin {
  require(valueToWithdraw <= totalLiquidity, "Withdraw value is not
    allowed for the total liquidity");
  totalLiquidity -= valueToWithdraw;
  emit UpdateLiquidity(totalLiquidity);
  msg.sender.transfer(valueToWithdraw);
}
```

```
function withdraw(uint amount) public {
  require(amount <= accountBalances[msg.sender], "Withdraw value is not
    allowed");
  accountBalances[msg.sender] -= amount;
  emit Withdraw(msg.sender, amount);
  msg.sender.transfer(amount);
}
```

5.3.2 Mutex

As described in section 5.3.1, there are three instance where delegation of control occurs to external entities, one of which is when a call to fetch the Oracle results is made. Since fetching these results could not be done as the last operation in the contract, a possibility to avoid attacks is to use the mutex pattern, this way, it is not possible for the Oracle contract to recall the original method to try to obtain an advantage.

```
contract BetHolder {

    bool locked = false;

    modifier avoidRecursion() {
        require(locked == false, "Reentrancy detected");
        locked = true;
        _;
        locked = false;
    }

  function setWinnerForEvent(uint32 eventId) public avoidRecursion {
        require(!isWinnerKnownForEventId[eventId], "Winner is already
    set");

        uint winnerSelection = eventsContract.getEventOracle(eventId).
    getWinnerSelection();
        isWinnerKnownForEventId[eventId] = true;
        winnerForEventId[eventId] = winnerSelection;
    }
```

Another possible guard against this attack vector is to make the getEventOracle call of the Oracle contract to be annotated with the *view* modifier, thus voiding the possibility of this method to perform any state change to the blockchain.

5.3.3 Pull Payment

The pull payment pattern is applied in the Balances contract. Instead of immediately transferring the earnings of a user to him when settling an event result, the user balance is just updated, and the user is then able to withdraw his (updated) balance. Since settling a bet involves updating several user balances, to perform Ether transfers in this method would be a serious security vulnerability. The implementation of the pull payment pattern (along with the deposit action) is shown bellow:

```
1
2  function deposit() public payable {
3      accountBalances[msg.sender] += msg.value;
4      emit Deposit(msg.sender, msg.value);
5  }
6
7  function withdraw(uint amount) public {
8      require(amount <= accountBalances[msg.sender], "Withdraw value is
       not allowed");
9      accountBalances[msg.sender] -= amount;
10     emit Withdraw(msg.sender, amount);
11     msg.sender.transfer(amount);
12 }
```

5.3.4 Emergency Stop

In order to react against unforeseen vulnerabilities, the emergency stop pattern is applied to the BetHolder contract. In this way, the administrator is able to stop all bet placements for an undefined amount of time, thus avoiding exploiting issues in the dApp. This functionality (bet placement) is the only one that is stopped: event settlement is still enabled and cannot be stopped - in this way, an user is protected since he can be sure that an administrator will not be able to stop the contract to avoid payment of a bet that causes the owners losses. Also, user balance withdrawal is not covered by the emergency stop for the same reasons. The implementation of this pattern is shown bellow:

```
1  contract BetHolder {
2      bool public isStopped = false;
3
4      modifier nonStopped() {
5          require(!isStopped, "Contract is stopped, operation is not
       available");
6          _;
7      }
8
9      function makeBet(uint32 eventId, uint32 selectionId, uint value)
       public nonStopped {
10         require(!eventsContract.isEventClosed(eventId), "Event is closed
       ");
```

```
11          require(!eventsContract.isEventSettled(eventId), "Event is
    already settled");
12          require(balancesContract.getBalance(msg.sender) >= value, "There
     is not enough balance");
13          require(value > 0, "Value should be greater than 0");
14
15          uint selectionOdd = eventsContract.getSelectionOdd(eventId,
    selectionId);
16          uint potentialEarnings = selectionOdd * value / 100 - value;
17          balancesContract.updateBalance(msg.sender, value, false);
18          balancesContract.updateLiquidity(potentialEarnings, false);
19
20          Bet memory newBet = Bet(msg.sender, eventId, selectionId,
    selectionOdd, value, false);
21          betsByEvent[eventId].push(newBet);
22          betsByUser[msg.sender].push(newBet);
23      }
24
25  }
```

5.3.5 Speed Bump

The Balances contract holds all the Ether of the *Betting On The Block* dApp. All transactions within this contract, excluding withdrawals and deposits, which are able to change user balances and available liquidity are performed through the BetHolder contract. As such, changing the address of this contract could be an attack vector to arbitrarily transfer user balances to liquidity and thus enable the administrator to fraudulently withdraw Ether from the users.

To avoid this issue, any change to the BetHolder address, after the initial set up, should await a trial period of seven days, during which any user could audit the new contract to ensure that the new contract does not introduce vulnerabilities. The administrator should initially request an update to the BetHolder contract address, specifying the new address. After this request, the administrator should submit a transaction to set this new bet holder address, which will only successful if the seven days have elapsed.

The implementation is shown bellow:

```
1   contract Balances {
2
3       struct UpdateBetHolderRequest {
4           uint time;
5           address newBetHolderAddress;
6       }
7
8       address betHolderAddress;
9       mapping (address => UpdateBetHolderRequest)
        requestsForUpdatingBetHolder;
10      uint constant changeBetHolderAddressWaitingPeriod = 7 days;
11
12      function requestUpdateBetHolderAddress(address newAddress) public
        onlyAdmin {
```

```
13          requestsForUpdatingBetHolder[msg.sender] =
    UpdateBetHolderRequest(now + changeBetHolderAddressWaitingPeriod,
    newAddress);
14          emit RequestToChangeBetHolderAddress(msg.sender, now +
    changeBetHolderAddressWaitingPeriod, newAddress);
15      }
16
17      modifier validateUpdateBetHolderAddress(address newAddress) {
18          if (betHolderAddress != address(0)) { // Allow setup of the
    BetHolder address immediately if it was never set
19              UpdateBetHolderRequest memory request =
    requestsForUpdatingBetHolder[msg.sender];
20              require(request.time <= now, "Operation can not be executed
    right now");
21              require(request.newBetHolderAddress == newAddress, "New
    address does not match previous request");
22              _;
23              delete requestsForUpdatingBetHolder[msg.sender];
24          } else {
25              _;
26          }
27      }
28
29      function setBetHolderAddress(address newAddress) public onlyAdmin
    validateUpdateBetHolderAddress(newAddress) {
30          betHolderAddress = newAddress;
31      }
32
33      event RequestToChangeBetHolderAddress(address, uint, address); //
    requester, time where change occur, new bet holder address
```

5.3.6 Oracle

In order to settle events, the outcome must be known. As such, since the result of the event can not be obtained easily from inside the blockchain, an oracle is required to provide this information. The oracle is a contract that should be set up upon the event creation and, in this case, should be administered by a third party, to avoid setting results that would benefit the dApp owners.

The implementation of the BetOracle is shown bellow:

```
1  contract BetOracle {
2
3      bool public isWinnerAvailable = false;
4      uint winner;
5      address public admin = msg.sender;
6
7      modifier onlyAdmin() {
8          require(msg.sender == admin, "This operation is only authorized
    for an admin");
9          _;
10     }
11
12     // Returns the winner index selection if it is available, otherwise
    throws.
```

```
13      function getWinnerSelection() public view returns (uint) {
14          require(isWinnerAvailable, "Winner is not available yet");
15          return winner;
16      }
17
18      function setWinner(uint winnerToSet) public onlyAdmin {
19          require(!isWinnerAvailable, "Winner is already set");
20          isWinnerAvailable = true;
21          winner = winnerToSet;
22          emit WinnerSet(winner);
23      }
24
25      event WinnerSet(uint);
26  }
```

5.4 Tests

In order to guarantee that the system behaves as intended, a comprehensive suite of 49 tests was developed. These tests not only verify the correct functionality of each individual smart contract (unit tests) but also test the end-to-end flows, both common and uncommon. Bellow, a summary of each implemented test is provided:

1. AccessControl Contract
   (a) is admin
   (b) add operator
   (c) remove operator
   (d) fail add operator
2. AlwaysZeroBetOracle Contract
   (a) is winner 0
   (b) winner should not be changeable
3. Balances Contract
   (a) deposit
   (b) withdraw balance with available funds
   (c) fail to withdraw balance without available funds
   (d) add liquidity
   (e) withdraw liquidity with available funds
   (f) fail to withdraw liquidity without available funds
   (g) fail to withdraw liquidity for non admin user
   (h) fail to set bet holder without previous request
   (i) fail to set bet holder with not enough time passed

4. BetOracle Contract
    (a) winner should not be set
    (b) fail to set winner with non admin account
    (c) set winner
    (d) winner should not be changeable
5. BetHolder Contract
    (a) make simple bet
    (b) fail to make bet in non existing selection
    (c) fail to make bet in non existing event
    (d) fail to make bet in closed event
    (e) fail to make bet without enough user balance
    (f) fail to make bet without liquidity
    (g) fail to make bet with value 0
    (h) fail to make bet in stopped contract
    (i) settle simple bet (user won)
    (j) settle simple bet (user lost)
    (k) fail to settle bet without result available
    (l) fail to settle bet without setting result
    (m) stop contract and resume it
    (n) fail to stop contract for non admin account
    (o) get user bets
6. Events Contract
    (a) create event
    (b) create selections with event
    (c) change odds
    (d) settle event
    (e) close event

The code for each test is available in annex B.

5.5 Improvements

Several improvements were identified in the development of this dApp, but were not considered, either for being out of scope of this work or due to the development effort required for proper implementation:

1. Implement multiple oracles to be able to avoid a single oracle selecting the wrong result and collect undue earnings.
2. Set the closing time of an event, in order to not allow bets after the closing time is passed.
3. Allow setting the maximum liquidity on a per event basis, instead of globally. This allows greater granularity and significant lowering of total business risks.
4. Allow voiding of events in oracle: some events may never have a valid result - for instance, a cancelled football match. In order to handle this situation, the Oracle contract should be modified to be able to provide a "void" result.

Chapter 6

Conclusion

This chapter provides an overview of the objectives achieved in this work, along with future work to be done and a personal overview.

6.1 Work Summary

This dissertation aims to document and analyse the most relevant design patterns employed in Ethereum Smart Contracts. A thorough review of the literature, explained in section 1.4.1 was carried out, along with an analysis of smart contracts deployed in the Ethereum main network.

A total of eleven smart contracts were identified and systematically analysed in chapter 4:

- Checks Effects Interaction (in section 4.1)
- Emergency Stop (in section 4.2)
- Speed Bump (in section 4.3)
- Rate Limit (in section 4.4)
- Balance Limit (in section 4.5)
- Pull Payment (in section 4.6)
- Oracle (in section 4.7)
- Automatic Deprecation (in section 4.8)
- Data Segregation (in section 4.9)
- Contract Relay (in section 4.10)
- Mutex (in section 4.11)

The analysis of each design pattern followed a consistent schema, comprising the *Intent*, the *Motivation*, the *Applicability*, the *Participants*, *Known Uses*, *Related Patterns* and a sample implementation in Solidity.

The design patterns identified as part of this work were effectively used in a complex use case, demonstrated in chapter 5. Considering the specific concerns of blockchain software engineering, extensive automated testing was prioritized and the functional requirements of the dApp were implemented successfully.

6.2 Gas Costs with Proof Of Stake Consensus

Currently, gas costs are an important concern for the application of design patterns - in fact, gas costs tend to be the largest operational expense to run a smart contract. As such, in chapter 4, the impact of gas costs of each pattern was discussed.

However, recent developments, during the development of this work, point to the fact that gas costs will tend to be less relevant. In fact, the switch of the consensus mechanism to Proof Of Stake (see section 2.2.1), from Proof Of Work, is largely anticipated to slash gas prices. The price decrease is expected due to the larger scalability provided by the new consensus mechanism, as well as the lower costs required to maintain the network, compared to the current Proof Of Work approach. The change to Proof Of Stake is set to occur in the deployment of Ethereum 2.0.

6.3 Limitations and Future Work

While the objectives set in chapter 1 were achieved, several limitations were identified.

An important improvement upon this work would be to perform a taxonomic analysis of the identified design patterns. The main reason for this to not have been done is the relative low number of identified patterns.

In relation to the Betting On The Block dApp, several limitations were identified and analysed in section 5.5.

As part of the future work, the development of a Front End for the Betting On The Block dApp is also considered. This was not considered a priority since it is completely aligned with the scope of this dissertation, but it would be valuable to showcase the dApp design decisions and use cases.

6.4 Personal overview

A noteworthy observation in relation to this work is that a significant amount of knowledge and innovation in this field is not part of the white literature but rather of the grey literature, often spearheaded by enthusiasts, entrepreneurs and the cryptocurrency industry in general. Innovative approaches to design patterns are often developed by the industry, instead of being done in an academic context.

Another relevant observation is the relative lack of features of the Solidity language. For instance, operations over mappings are very limited - even simple operations such as fetching the number of elements in a mapping are not present.

The development tools for smart contracts development, namely the Truffle suite, provide a productive environment for achieving the goals set in chapter 5.

As a final observation, the simplicity of the Solidity language and the intrinsic features of the blockchain, making access control and currency transfers easy to implement, allowed for the relatively ambitious functional requirements of the Betting On The Block dApp to be met

with few lines of code - under 600 lines. While a reason for this is due to the chosen design, a similar implementation using a centralized approach would probably be more complex.

Bibliography

[1] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: (2008).

[2] Vitalik Buterin et al. "A next-generation smart contract and decentralized application platform". In: *white paper* (2014).

[3] Maximilian Wohrer and Uwe Zdun. "Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity". In: *Blockchain Oriented Software Engineering (IWBOSE), 2018 International Workshop on*. IEEE. 2018, pp. 2–8.

[4] Loi Luu et al. "Making smart contracts smarter". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 254–269.

[5] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A survey of attacks on ethereum smart contracts (sok)". In: *Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[6] Partha Chakraborty et al. "Understanding the software development practices of blockchain projects: a survey". In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM. 2018, p. 28.

[7] Simone Porru et al. "Blockchain-oriented software engineering: challenges and new directions". In: *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press. 2017, pp. 169–171.

[8] Xiwei Xu et al. "A Pattern Collection for Blockchain-based Applications". In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs - EuroPLoP '18*. ACM Press, 2018. doi: 10.1145/3282308.3282312. url: https://doi.org/10.1145%2F3282308.3282312.

[9] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. "A Semantic Framework for the Security Analysis of Ethereum Smart Contracts". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 243–269. doi: 10.1007/978-3-319-89722-6_10. url: https://doi.org/10.1007%2F978-3-319-89722-6_10.

[10] Xiwei Xu et al. "A Taxonomy of Blockchain-Based Systems for Architecture Design". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Apr. 2017. doi: 10.1109/icsa.2017.33. url: https://doi.org/10.1109%2Ficsa.2017.33.

[11] Massimo Bartoletti and Livio Pompianu. "An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns". In: *Financial Cryptography and Data Security*. Springer International Publishing, 2017, pp. 494–509. doi: 10.1007/978-3-319-70278-0_31. url: https://doi.org/10.1007%2F978-3-319-70278-0_31.

[12] Peng Zhang et al. "Applying software patterns to address interoperability in blockchain-based healthcare apps". In: *arXiv preprint arXiv:1706.03700* (2017).

[13] Maximilian Wöhrer and Uwe Zdun. "Design patterns for smart contracts in the ethereum ecosystem". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE. 2018, pp. 1513–1520.

[14] Peng Zhang et al. "Design of blockchain-based apps using familiar software patterns to address interoperability challenges in healthcare". In: *PLoP-24th Conference On Pattern Languages Of Programs*. 2017.

[15] Anastasia Mavridou and Aron Laszka. "Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach". In: *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2018, pp. 523–540. doi: 10.1007/978-3-662-58387-6_28. url: https://doi.org/10.1007%2F978-3-662-58387-6_28.

[16] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. "Detecting Token Systems on Ethereum". In: *Financial Cryptography and Data Security*. Springer International Publishing, 2019, pp. 93–112. doi: 10.1007/978-3-030-32101-7_7. url: https://doi.org/10.1007%2F978-3-030-32101-7_7.

[17] Mayukh Mukhopadhyay. *Ethereum Smart Contract Development: Build blockchain-based decentralized applications using solidity*. Packt Publishing Ltd, 2018.

[18] Ardit Dika. "Ethereum smart contracts: Security vulnerabilities and security tools". MA thesis. NTNU, 2017.

[19] Wren Chan and Aspen Olmsted. "Ethereum transaction graph analysis". In: *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, Dec. 2017. doi: 10.23919/icitst.2017.8356459. url: https://doi.org/10.23919%2Ficitst.2017.8356459.

[20] Sergei Tikhomirov. "Ethereum: State of Knowledge and Research Perspectives". In: *Foundations and Practice of Security*. Springer International Publishing, 2018, pp. 206–221. doi: 10.1007/978-3-319-75650-9_14. url: https://doi.org/10.1007%2F978-3-319-75650-9_14.

[21] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. "Foundations and Tools for the Static Analysis of Ethereum Smart Contracts". In: *Computer Aided Verification*. Springer International Publishing, 2018, pp. 51–78. doi: 10.1007/978-3-319-96145-3_4. url: https://doi.org/10.1007%2F978-3-319-96145-3_4.

[22] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'Reilly Media, 2018.

[23] Alexander Mense and Markus Flatscher. "Security Vulnerabilities in Ethereum Smart Contracts". In: *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services - iiWAS2018*. ACM Press, 2018. doi: 10.1145/3282373.3282419. url: https://doi.org/10.1145%2F3282373.3282419.

[24] Maximilian Wohrer and Uwe Zdun. "Smart contracts: security patterns in the ethereum ecosystem and solidity". In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, Mar. 2018. doi: 10.1109/iwbose.2018.8327565. url: https://doi.org/10.1109%2Fiwbose.2018.8327565.

[25] Anastasia Mavridou and Aron Laszka. "Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 270–277. doi: 10.1007/978-3-319-89722-6_11. url: https://doi.org/10.1007%2F978-3-319-89722-6_11.

[26] Anastasia Mavridou et al. "VeriSolid: Correct-by-Design Smart Contracts for Ethereum". In: *Financial Cryptography and Data Security*. Springer International Publishing, 2019, pp. 446–465. doi: 10.1007/978-3-030-32101-7_27. url: https://doi.org/10.1007%2F978-3-030-32101-7_27.

[27] Victoria L. Lemieux. "A typology of blockchain recordkeeping solutions and some reflections on their implications for the future of archival preservation". In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, Dec. 2017. doi: 10.1109/

bigdata.2017.8258180. url: https://doi.org/10.1109%2Fbigdata.2017.8258180.

[28] Yue Liu et al. "Applying Design Patterns in Smart Contracts". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 92–106. doi: 10.1007/978-3-319-94478-4_7. url: https://doi.org/10.1007%2F978-3-319-94478-4_7.

[29] Harry Kalodner et al. "BlockSci: Design and applications of a blockchain analysis platform". In: *arXiv preprint arXiv:1709.02489* (2017).

[30] Shuai Wang et al. "Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* (2019), pp. 1–12. doi: 10.1109/tsmc.2019.2895123. url: https://doi.org/10.1109%2Ftsmc.2019.2895123.

[31] Peng Zhang et al. "Blockchain technology use cases in healthcare". In: *Advances in Computers*. Vol. 111. Elsevier, 2018, pp. 1–41.

[32] Irish Singh and Seok-Won Lee. "Comparative Requirements Analysis for the Feasibility of Blockchain for Secure Cloud". In: *Communications in Computer and Information Science*. Springer Singapore, 2018, pp. 57–72. doi: 10.1007/978-981-10-7796-8_5. url: https://doi.org/10.1007%2F978-981-10-7796-8_5.

[33] Peng Zhang et al. "FHIRChain: Applying Blockchain to Securely and Scalably Share Clinical Data". In: *Computational and Structural Biotechnology Journal* 16 (2018), pp. 267–278. doi: 10.1016/j.csbj.2018.07.004. url: https://doi.org/10.1016%2Fj.csbj.2018.07.004.

[34] Florian Wessling et al. "How much blockchain do you need?: Towards a concept for building hybrid DApp architectures". In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain - WETSEB '18*. ACM Press, 2018. doi: 10.1145/3194113.3194121. url: https://doi.org/10.1145%2F3194113.3194121.

[35] Jonatan Bergquist et al. "On the design of communication and transaction anonymity in blockchain-based transactive microgrids". In: *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers - SERIAL '17*. ACM Press, 2017. doi: 10.1145/3152824.3152827. url: https://doi.org/10.1145%2F3152824.3152827.

[36] Henry M. Kim and Marek Laskowski. "Toward an ontology-driven blockchain design for supply-chain provenance". In: *Intelligent Systems in Accounting, Finance and Management* 25.1 (Jan. 2018), pp. 18–27. doi: 10.1002/isaf.1424. url: https://doi.org/10.1002%2Fisaf.1424.

[37] Mayra Samaniego and Ralph Deters. "Virtual resources & blockchain for configuration management in IoT". In: *Journal of Ubiquitous Systems & Pervasive Networks* 9.2 (2017), pp. 01–13.

[38] Jeff Herbert and Alan Litchfield. "A novel method for decentralised peer-to-peer software license validation using cryptocurrency blockchain technology". In: *Proceedings of the 38th Australasian Computer Science Conference (ACSC 2015)*. Vol. 27. 2015, p. 30.

[39] Qinghua Lu and Xiwei Xu. "Adaptable Blockchain-Based Systems: A Case Study for Product Traceability". In: *IEEE Software* 34.6 (Nov. 2017), pp. 21–27. doi: 10.1109/ms.2017.4121227. url: https://doi.org/10.1109%2Fms.2017.4121227.

[40] Michele Marchesi, Lodovica Marchesi, and Roberto Tonelli. "An Agile Software Engineering Method to Design Blockchain Applications". In: *Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia on ZZZ -*

CEE-SECR '18. ACM Press, 2018. doi: 10.1145/3290621.3290627. url: https://doi.org/10.1145%2F3290621.3290627.

[41] Pradip Kumar Sharma, Seo Yeon Moon, and Jong Hyuk Park. "Block-VN: A distributed blockchain based vehicular network architecture in smart City." In: *JIPS* 13.1 (2017), pp. 184–195.

[42] Arshdeep Bahga and Vijay K Madisetti. "Blockchain platform for industrial internet of things". In: *Journal of Software Engineering and Applications* 9.10 (2016), p. 533.

[43] Tshilidzi Marwala and Bo Xing. "Blockchain and artificial intelligence". In: *arXiv preprint arXiv:1802.04451* (2018).

[44] Fredrik Milani, Luciano Garcıa-Bañuelos, and Marlon Dumas. "Blockchain and business process improvement". In: *BPTrends newsletter (October 2016)* (2016).

[45] Zibin Zheng et al. "Blockchain challenges and opportunities: A survey". In: *International Journal of Web and Grid Services* 14.4 (2018), pp. 352–375.

[46] Tareq Ahram et al. "Blockchain technology innovations". In: *2017 IEEE Technology & Engineering Management Conference (TEMSCON)*. IEEE. 2017, pp. 137–141.

[47] Simone Porru et al. "Blockchain-Oriented Software Engineering: Challenges and New Directions". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, May 2017. doi: 10.1109/icse-c.2017.142. url: https://doi.org/10.1109%2Ficse-c.2017.142.

[48] Simona Ibba et al. "CitySense: blockchain-oriented smart cities". In: *Proceedings of the XP2017 Scientific Workshops on - XP '17*. ACM Press, 2017. doi: 10.1145/3120459.3120472. url: https://doi.org/10.1145%2F3120459.3120472.

[49] Paul Rimba et al. "Comparing Blockchain and Cloud Services for Business Process Execution". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Apr. 2017. doi: 10.1109/icsa.2017.44. url: https://doi.org/10.1109%2Ficsa.2017.44.

[50] Rogelio Rivera et al. "How digital identity on blockchain can contribute in a smart city environment". In: *2017 International Smart Cities Conference (ISC2)*. IEEE, Sept. 2017. doi: 10.1109/isc2.2017.8090839. url: https://doi.org/10.1109%2Fisc2.2017.8090839.

[51] Michael Coblenz. "Obsidian: A Safer Blockchain Programming Language". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, May 2017. doi: 10.1109/icse-c.2017.150. url: https://doi.org/10.1109%2Ficse-c.2017.150.

[52] Giuseppe Destefanis et al. "Smart contracts vulnerabilities: A call for blockchain software engineering?" In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, Mar. 2018. doi: 10.1109/iwbose.2018.8327567. url: https://doi.org/10.1109%2Fiwbose.2018.8327567.

[53] Ilya Sergey and Aquinas Hobor. "A Concurrent Perspective on Smart Contracts". In: *Financial Cryptography and Data Security*. Springer International Publishing, 2017, pp. 478–493. doi: 10.1007/978-3-319-70278-0_30. url: https://doi.org/10.1007%2F978-3-319-70278-0_30.

[54] Claus Pahl et al. "An architecture pattern for trusted orchestration in IoT edge clouds". In: *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, Apr. 2018. doi: 10.1109/fmec.2018.8364046. url: https://doi.org/10.1109%2Ffmec.2018.8364046.

[55] Andrea M Rozario and Miklos A Vasarhelyi. "Auditing with Smart Contracts." In: *International Journal of Digital Accounting Research* 18 (2018).

[56] Massimo Bartoletti and Roberto Zunino. "BitML: A Calculus for Bitcoin Smart Contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18*. ACM Press, 2018. doi: 10.1145/3243734.3243795. url: https://doi.org/10.1145%2F3243734.3243795.

[57] Lin William Cong and Zhiguo He. "Blockchain disruption and smart contracts". In: *The Review of Financial Studies* 32.5 (2019), pp. 1754–1797.

[58] Mustafa Al-Bassam et al. "Chainspace: A Sharded Smart Contracts Platform". In: *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, 2018. doi: 10.14722/ndss.2018.23241. url: https://doi.org/10.14722%2Fndss.2018.23241.

[59] Bo Jiang, Ye Liu, and W. K. Chan. "ContractFuzzer: fuzzing smart contracts for vulnerability detection". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*. ACM Press, 2018. doi: 10.1145/3238147.3238177. url: https://doi.org/10.1145%2F3238147.3238177.

[60] Lorenz Breindenbach et al. "Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 1335–1352.

[61] Yi Zhou et al. "Erays: reverse engineering ethereum's opaque smart contracts". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 1371–1385.

[62] Karthikeyan Bhargavan et al. "Formal verification of smart contracts: Short paper". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM. 2016, pp. 91–96.

[63] Neville Grech et al. "MadMax: surviving out-of-gas conditions in Ethereum smart contracts". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (Oct. 2018), pp. 1–27. doi: 10.1145/3276486. url: https://doi.org/10.1145%2F3276486.

[64] Chao Liu et al. "ReGuard: finding reentrancy bugs in smart contracts". In: *Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE '18*. ACM Press, 2018. doi: 10.1145/3183440.3183495. url: https://doi.org/10.1145%2F3183440.3183495.

[65] Jack Pettersson and Robert Edström. "Safer smart contracts through type-driven development". In: *Master's thesis. Chalmers University of Technology & University of Gothenburg, Gothenburg, Sweden* (2016).

[66] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. "Scilla: a smart contract intermediate-level language". In: *arXiv preprint arXiv:1801.00687* (2018).

[67] Petar Tsankov et al. "Securify: Practical Security Analysis of Smart Contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18*. ACM Press, 2018. doi: 10.1145/3243734.3243780. url: https://doi.org/10.1145%2F3243734.3243780.

[68] Reza M. Parizi, Amritraj, and Ali Dehghantanha. "Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2018, pp. 75–91. doi: 10.1007/978-3-319-94478-4_6. url: https://doi.org/10.1007%2F978-3-319-94478-4_6.

[69] Roberto Tonelli et al. "Smart contracts software metrics: a first study". In: *arXiv preprint arXiv:1802.01517* (2018).

[70] Daniel Macrinici, Cristian Cartofeanu, and Shang Gao. "Smart contract applications within blockchain technology: A systematic mapping study". In: *Telematics and Informatics* 35.8 (Dec. 2018), pp. 2337–2354. doi: 10.1016/j.tele.2018.10.004. url: https://doi.org/10.1016%2Fj.tele.2018.10.004.

[71] Sergei Tikhomirov et al. "SmartCheck: static analysis of ethereum smart contracts". In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain - WETSEB '18*. ACM Press, 2018. doi: 10.1145/3194113.3194115. url: https://doi.org/10.1145%2F3194113.3194115.

[72] Sukrit Kalra et al. "ZEUS: Analyzing Safety of Smart Contracts". In: *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, 2018. doi: 10.14722/ndss.2018.23082. url: https://doi.org/10.14722%2Fndss.2018.23082.

[73] World Bank. *Enforcing Contracts*. 2018. url: http://www.doingbusiness.org/en/data/exploretopics/enforcing-contracts (visited on 12/12/2018).

[74] Nick Szabo. "Smart contracts: building blocks for digital markets". In: *EXTROPY: The Journal of Transhumanist Thought,(16)* (1996).

[75] Adam Back et al. *Hashcash-a denial of service counter-measure*. 2002.

[76] Apache HTTP Server. *SSL/TLS Strong Encryption: An Introduction*. 2018. url: http://httpd.apache.org/docs/2.2/ssl/ssl_intro.html#cryptographictech (visited on 12/10/2018).

[77] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151 (2014), pp. 1–32.

[78] *One of Ethereum's Earliest Smart Contract Languages Is Headed for Retirement - CoinDesk*. https://www.coindesk.com/one-of-ethereums-earliest-smart-contract-languages-is-headed-for-retirement. (Accessed on 01/27/2019).

[79] *Zeppelin Solutions Serpent Compiler Audit v1.0.0*. https://docs.google.com/document/d/1_PqXuAkvgUAOG3jbBvaUvqN6W90eJ3N4IdTLNMRAijo. (Accessed on 01/27/2019).

[80] *Solidity by Example — Solidity 0.5.1 documentation*. https://solidity.readthedocs.io/en/v0.5.1/solidity-by-example.html. (Accessed on 01/27/2019).

[81] Susana Nicola, Eduarda Pinto Ferreira, and João José Pinto Ferreira. "Conceptual model for decomposing the value for the customer". In: *IEMS'12—3rd Industrial Engineering and Management Symposium*. Universidade do Porto. 2012, pp. 41–43.

[82] Valarie A Zeithaml. "Consumer perceptions of price, quality, and value: a means-end model and synthesis of evidence". In: *Journal of marketing* 52.3 (1988), pp. 2–22.

[83] Peter A Koen. "The fuzzy front end for incremental, platform and breakthrough products and services". In: *PDMA Handbook* (2004), pp. 81–91.

[84] Erich Gamma et al. "Elements of Reusable Object-Oriented Software". In: *Design Patterns. massachusetts: Addison-Wesley Publishing Company* (1995).

Appendix A

Betting On The Block dApp Smart Contracts

A.1 AccessControl Contract

```
pragma solidity ^0.5.0;

contract AccessControl {

    mapping (address=>bool) internal operators;
    address internal admin = msg.sender;

    modifier onlyAdmin() {
        require(msg.sender == admin, "This is only available to admins")
    ;
        _;
    }

    function isAdmin(address addr) public view returns (bool) {
        return admin == addr;
    }

    function isOperator(address addr) public view returns (bool) {
        return operators[addr];
    }

    function addOperator(address toAdd) public onlyAdmin {
        operators[toAdd] = true;
        emit OperatorAdded(toAdd);
    }

    function removeOperator(address toRemove) public onlyAdmin {
        operators[toRemove] = false;
        emit OperatorRemoved(toRemove);
    }

    event OperatorAdded(address);
    event OperatorRemoved(address);
}
```

A.2 BetOracle Contract

```
pragma solidity ^0.5.0;

contract BetOracle {

    bool public isWinnerAvailable = false;
    uint winner;
    address public admin = msg.sender;

    modifier onlyAdmin() {
        require(msg.sender == admin, "This operation is only authorized for an admin");
        _;
    }

    // Returns the winner index selection if it is available, otherwise throws.
    function getWinnerSelection() public view returns (uint) {
        require(isWinnerAvailable, "Winner is not available yet");
        return winner;
    }

    function setWinner(uint winnerToSet) public onlyAdmin {
        require(!isWinnerAvailable, "Winner is already set");
        isWinnerAvailable = true;
        winner = winnerToSet;
        emit WinnerSet(winner);
    }

    event WinnerSet(uint);
}
```

A.3 AlwaysZeroBetOracle Contract

```
pragma solidity ^0.5.0;

import "./BetOracle.sol";

// Returns the winner as the selection with index 0.
contract AlwaysZeroBetOracle is BetOracle {

    constructor() public {
        isWinnerAvailable = true;
        winner = 0;
    }
}
```

A.4 Balances

```
pragma solidity ^0.5.0;

import "./AccessControl.sol";

contract Balances {

    mapping(address => uint) internal accountBalances;
    uint public totalLiquidity = 0;
    address betHolderAddress;
    AccessControl internal accessControl;

    constructor(AccessControl accessCtrl) public {
        accessControl = accessCtrl;
    }

    modifier onlyBetHolder() {
        require (msg.sender == betHolderAddress, "Operation only
    authorized for the Bet Holder contract");
        _;
    }

    modifier onlyAdmin() {
        require(accessControl.isAdmin(msg.sender), "This operation is
    only available to admins");
        _;
    }

    function deposit() public payable {
        accountBalances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    function withdraw(uint amount) public {
        require(amount <= accountBalances[msg.sender], "Withdraw value
    is not allowed");
        accountBalances[msg.sender] -= amount;
        emit Withdraw(msg.sender, amount);
        msg.sender.transfer(amount);
    }

    function getBalance(address addr) public view returns (uint) {
        return accountBalances[addr];
    }

    function addLiquidity() public payable {
        totalLiquidity += msg.value;
        emit LiquidityDeposit(msg.value);
    }

    function withdrawLiquidity(uint valueToWithdraw) public onlyAdmin {
        require(valueToWithdraw <= totalLiquidity, "Withdraw value is
    not allowed for the total liquidity");
        totalLiquidity -= valueToWithdraw;
        emit UpdateLiquidity(totalLiquidity);
        msg.sender.transfer(valueToWithdraw);
    }

    function updateBalance(address toUpdate, uint value, bool
    shouldIncrease) public onlyBetHolder {
```

```
55          if (shouldIncrease) {
56              accountBalances[toUpdate] += value;
57          } else {
58              require(accountBalances[toUpdate] >= value, "Balance is
    insufficient");
59              accountBalances[toUpdate] -= value;
60          }
61          emit UpdateBalance(toUpdate, accountBalances[toUpdate]);
62      }
63
64      function updateLiquidity(uint value, bool shouldIncrease) public
    onlyBetHolder {
65          if (shouldIncrease) {
66              totalLiquidity += value;
67          } else {
68              require(totalLiquidity >= value, "Total liquidity is
    insufficient");
69              totalLiquidity -= value;
70          }
71          emit UpdateLiquidity(totalLiquidity);
72      }
73
74      function requestUpdateBetHolderAddress(address newAddress) public
    onlyAdmin {
75          requestsForUpdatingBetHolder[msg.sender] =
    UpdateBetHolderRequest(now + changeBetHolderAddressWaitingPeriod,
    newAddress);
76          emit RequestToChangeBetHolderAddress(msg.sender, now +
    changeBetHolderAddressWaitingPeriod, newAddress);
77      }
78
79      function setBetHolderAddress(address newAddress) public onlyAdmin
    validateUpdateBetHolderAddress(newAddress) {
80          betHolderAddress = newAddress;
81      }
82
83      modifier validateUpdateBetHolderAddress(address newAddress) {
84          if (betHolderAddress != address(0)) {
85              UpdateBetHolderRequest memory request =
    requestsForUpdatingBetHolder[msg.sender];
86              require(request.time <= now, "Operation can not be executed
    right now");
87              require(request.newBetHolderAddress == newAddress, "New
    address does not match previous request");
88              _;
89              delete requestsForUpdatingBetHolder[msg.sender];
90          } else {
91              _;
92          }
93      }
94
95      struct UpdateBetHolderRequest {
96          uint time;
97          address newBetHolderAddress;
98      }
99      uint constant changeBetHolderAddressWaitingPeriod = 7 days;
100     mapping (address => UpdateBetHolderRequest)
    requestsForUpdatingBetHolder;
101
```

```
102     event Deposit(address, uint);
103     event Withdraw(address, uint);
104     event UpdateBalance(address, uint);
105     event UpdateLiquidity(uint);
106     event LiquidityDeposit(uint);
107     event RequestToChangeBetHolderAddress(address, uint, address); //
        requester, time where change occur, new bet holder address
108 }
```

A.5 BetHolder Contract

```
1  pragma solidity ^0.5.0;
2  pragma experimental ABIEncoderV2;
3
4  import "./Events.sol";
5  import "./Balances.sol";
6  import "./SharedStructs.sol";
7  import "./AccessControl.sol";
8
9  contract BetHolder {
10
11     struct Bet {
12         address from;
13         uint32 eventId;
14         uint32 selectionId;
15         uint odd;
16         uint value;
17         bool isSettled;
18     }
19
20     mapping (uint32 => Bet[]) betsByEvent;
21     mapping (address => Bet[]) betsByUser;
22     mapping (uint32 => bool) settledEvents;
23     mapping (uint32 => bool) isWinnerKnownForEventId;
24     mapping (uint32 => uint) winnerForEventId;
25     Events eventsContract;
26     Balances balancesContract;
27     AccessControl accessControlContract;
28     bool public isStopped = false;
29     bool locked = false;
30
31     modifier avoidRecursion() {
32         require(locked == false, "Reentrancy detected");
33         locked = true;
34         _;
35         locked = false;
36     }
37
38     constructor(Events events, Balances balances, AccessControl
       accessControl) public {
39         eventsContract = events;
40         balancesContract = balances;
41         accessControlContract = accessControl;
42     }
43
```

```
44      modifier onlyAdmin() {
45          require(accessControlContract.isAdmin(msg.sender), "This
    operation is only available to admins");
46          _;
47      }
48
49      modifier nonStopped() {
50          require(!isStopped, "Contract is stopped, operation is not
    available");
51          _;
52      }
53
54      function makeBet(uint32 eventId, uint32 selectionId, uint value)
    public nonStopped {
55          require(!eventsContract.isEventClosed(eventId), "Event is closed
    ");
56          require(!eventsContract.isEventSettled(eventId), "Event is
    already settled");
57          require(balancesContract.getBalance(msg.sender) >= value, "There
     is not enough balance");
58          require(value > 0, "Value should be greater than 0");
59
60          uint selectionOdd = eventsContract.getSelectionOdd(eventId,
    selectionId);
61          uint potentialEarnings = selectionOdd * value / 100 - value;
62          balancesContract.updateBalance(msg.sender, value, false);
63          balancesContract.updateLiquidity(potentialEarnings, false);
64
65          Bet memory newBet = Bet(msg.sender, eventId, selectionId,
    selectionOdd, value, false);
66          betsByEvent[eventId].push(newBet);
67          betsByUser[msg.sender].push(newBet);
68      }
69
70      function setWinnerForEvent(uint32 eventId) public avoidRecursion {
71          require(!isWinnerKnownForEventId[eventId], "Winner is already
    set");
72
73          uint winnerSelection = eventsContract.getEventOracle(eventId).
    getWinnerSelection();
74          isWinnerKnownForEventId[eventId] = true;
75          winnerForEventId[eventId] = winnerSelection;
76      }
77
78      function settleEvent(uint32 eventId) public {
79          require(!settledEvents[eventId], "Event is already settled");
80          require(isWinnerKnownForEventId[eventId], "Winner is not yet set
    ");
81
82          uint winnerSelection = winnerForEventId[eventId];
83
84          Bet[] memory bets = betsByEvent[eventId];
85          for (uint i = 0; i < bets.length; i++) {
86              if (bets[i].selectionId == winnerSelection) {
87                  balancesContract.updateBalance(bets[i].from, bets[i].odd
     * bets[i].value / 100, true);
88              } else {
89                  balancesContract.updateLiquidity(bets[i].odd * bets[i].
    value / 100, true);
```

```
            }
        }
        settledEvents[eventId] = true;
        eventsContract.settleEvent(eventId);
    }

    function getBetsByEventId(uint32 eventId) public view returns (Bet[]
    memory) {
        bool isEventSettled = settledEvents[eventId];
        Bet[] memory bets = betsByEvent[eventId];
        for (uint i = 0; i < bets.length; i++) {
            bets[i].isSettled = isEventSettled;
        }
        return bets;
    }

    function getBetsByUser(address user) public view returns (Bet[]
    memory) {
        Bet[] memory bets = betsByUser[user];
        for (uint i = 0; i < bets.length; i++) {
            bets[i].isSettled = settledEvents[bets[i].eventId];
        }
        return bets;
    }

    function setStopped(bool stopContract) public onlyAdmin {
        isStopped = stopContract;
        emit StartStopBetHolder(stopContract);
    }

    event StartStopBetHolder(bool);
}
```

A.6 Events Contract

```
pragma solidity ^0.5.0;
pragma experimental ABIEncoderV2;

import "./AccessControl.sol";
import "./SharedStructs.sol";

contract Events {

    mapping(uint32 => uint256) internal eventIdToIndex;
    SharedStructs.Event[] public events;
    mapping(uint32 => SharedStructs.Selection[]) internal
    selectionsByEventId;
    AccessControl internal accessControl;
    address betHolderAddress;

    constructor(AccessControl accessCtrl) public {
        events.push(SharedStructs.Event(0, "void", new BetOracle(), true
    , true));
        accessControl = accessCtrl;
    }
```

```
19
20      modifier onlyBetHolder() {
21          require (msg.sender == betHolderAddress, "Operation only
        authorized for the Bet Holder contract");
22          _;
23      }
24
25      modifier onlyOperators() {
26          require(accessControl.isOperator(msg.sender), "This operation is
         only available to operators");
27          _;
28      }
29
30      modifier onlyAdmin() {
31          require(accessControl.isAdmin(msg.sender), "This operation is
        only available to admins");
32          _;
33      }
34
35      function createActiveEvent(SharedStructs.Event memory evnt,
        SharedStructs.Selection[] memory selections) public onlyOperators {
36          uint256 length = events.push(evnt);
37          eventIdToIndex[evnt.id] = length-1;
38          for (uint i = 0; i<selections.length; i++) {
39              selectionsByEventId[evnt.id].push(selections[i]);
40          }
41      }
42
43      function changeOdds(uint32 eventId, uint32 runnerId, uint newOdds)
        public onlyOperators {
44          require(eventIdToIndex[eventId] != 0, "Event ID does not exist")
        ;
45          selectionsByEventId[eventId][runnerId].odd = newOdds;
46      }
47
48      function closeEvent(uint32 eventId) public onlyOperators() {
49         require(eventIdToIndex[eventId] != 0, "Event ID does not exist");
50          events[eventIdToIndex[eventId]].isClosed = true;
51          emit EventClosed(eventId);
52      }
53
54      function getEvent(uint32 id) public view returns (SharedStructs.
        Event memory) {
55          require(eventIdToIndex[id] != 0, "Event ID does not exist");
56          return events[eventIdToIndex[id]];
57      }
58
59      function isEventSettled(uint32 id) public view returns (bool) {
60          require(eventIdToIndex[id] != 0, "Event ID does not exist");
61          return events[eventIdToIndex[id]].isSettled;
62      }
63
64      function isEventClosed(uint32 id) public view returns (bool) {
65          require(eventIdToIndex[id] != 0, "Event ID does not exist");
66          return events[eventIdToIndex[id]].isClosed;
67      }
68
69      function getEventOracle(uint32 id) public view returns (BetOracle) {
70          require(eventIdToIndex[id] != 0, "Event ID does not exist");
```

```
71          return events[eventIdToIndex[id]].betOracle;
72      }
73
74      function getSelections(uint32 eventId) public view returns (
        SharedStructs.Selection[] memory) {
75          require(eventIdToIndex[eventId] != 0, "Event ID does not exist")
        ;
76          return selectionsByEventId[eventId];
77      }
78
79      function getSelectionOdd(uint32 eventId, uint selectionIdx) public
        view returns (uint) {
80          require(eventIdToIndex[eventId] != 0, "Event ID does not exist")
        ;
81          return selectionsByEventId[eventId][selectionIdx].odd;
82      }
83
84      function setBetHolderAddress(address newAddress) public onlyAdmin {
85          betHolderAddress = newAddress;
86      }
87
88      function settleEvent(uint32 eventId) public onlyBetHolder {
89          require(eventIdToIndex[eventId] != 0, "Event ID does not exist")
        ;
90          events[eventIdToIndex[eventId]].isSettled = true;
91          events[eventIdToIndex[eventId]].isClosed = true;
92          emit EventSettled(eventId);
93      }
94
95      event EventClosed(uint32);
96      event EventSettled(uint32);
97  }
```

A.7 SharedStructs

```
1   pragma solidity ^0.5.0;
2   pragma experimental ABIEncoderV2;
3
4   import "./BetOracle.sol";
5
6   library SharedStructs {
7       struct Selection {
8           string name;
9           uint odd;
10      }
11
12      struct Event {
13          uint32 id;
14          string name;
15          BetOracle betOracle;
16          bool isSettled;
17          bool isClosed;
18      }
19  }
```

Appendix B

Betting On The Block dApp Tests

B.1 AccessControl Tests

```
const AccessControl = artifacts.require("AccessControl");
const assert = require('assert');

contract('Access Control', async accounts => {
    let accessControl;

    let userAccount;
    let adminAccount;
    let operatorAccount;

    beforeEach(async function() {
        accessControl = await AccessControl.new();

        adminAccount = accounts[0];
        userAccount = accounts[1];
        operatorAccount = accounts[5];
    });

    it('is admin', async () => {
        assert.ok(await accessControl.isAdmin(adminAccount), "Should be admin");
        assert.ok(false === await accessControl.isAdmin(userAccount), "Should not be admin");
        assert.ok(false === await accessControl.isAdmin(operatorAccount), "Should not be admin");
    });

    it('add operator', async () => {
        assert.ok(false === await accessControl.isOperator(operatorAccount), "Should not be a operator");
        await accessControl.addOperator(operatorAccount, {from: adminAccount});
        assert.ok(await accessControl.isOperator(operatorAccount), "Should be a operator");
    });

    it('remove operator', async () => {
        assert.ok(false === await accessControl.isOperator(operatorAccount), "Should not be a operator");
        await accessControl.addOperator(operatorAccount, {from: adminAccount});
```

```
34          assert.ok(await accessControl.isOperator(operatorAccount), "
    Should be a operator");
35          await accessControl.removeOperator(operatorAccount, {from:
    adminAccount});
36          assert.ok(false === await accessControl.isOperator(
    operatorAccount), "Should not be a operator");
37      });
38
39      it('fail add operator', async () => {
40          assert.ok(false === await accessControl.isOperator(
    operatorAccount), "Should not be a operator");
41          try {
42              await accessControl.addOperator(userAccount, {from:
    userAccount});
43          } catch(err) {
44              return;
45          }
46          assert.fail("Should have failed to add a operator with a user
    account");
47      });
48
49  });
```

B.2 AlwaysZeroBetOracle Tests

```
1   const AlwaysZeroBetOracle = artifacts.require("AlwaysZeroBetOracle");
2
3   const assert = require('assert');
4
5   contract('AlwaysZeroBetOracle', async accounts => {
6       let oracle;
7
8       let userAccount;
9       let adminAccount;
10      let operatorAccount;
11
12      beforeEach(async function() {
13          oracle = await AlwaysZeroBetOracle.new();
14
15          adminAccount = accounts[0];
16          userAccount = accounts[1];
17          operatorAccount = accounts[5];
18      });
19
20      it('is winner 0', async () => {
21          assert.ok(await oracle.getWinnerSelection(), 0, "Winner should
    be selection with index 0");
22      });
23
24      it('winner should not be changeable', async () => {
25          try {
26              await oracle.setWinner(5);
27          } catch (err) {
28              return;
29          }
```

```
        assert.fail("Winner should not be changeable");
    });
});
```

B.3 Balances Tests

```
const Balances = artifacts.require("Balances");
const AccessControl = artifacts.require("AccessControl");
const assert = require('assert');

contract('Balances', async accounts => {
    let accessControl;

    let userAccount;
    let adminAccount;
    let operatorAccount;

    beforeEach(async function () {
        accessControl = await AccessControl.new();
        balances = await Balances.new(accessControl.address);

        adminAccount = accounts[0];
        userAccount = accounts[1];
        operatorAccount = accounts[5];
    });

    it('deposit', async () => {
        assert.equal(parseInt(await balances.getBalance(userAccount)),
    0, "Balance should be 0 wei");
        await balances.deposit({ from: userAccount, value: 100 });
        let balance = parseInt(await balances.getBalance(userAccount));

        assert.equal(balance, 100, "Balance should be 100 wei");
    });

    it('withdraw balance with available funds', async () => {
        balances.deposit({ from: userAccount, value: 100 });
        balances.withdraw(100, { from: userAccount });

        let balance = parseInt(await balances.getBalance(userAccount));

        assert.equal(balance, 0, "Balance should be 0 wei");
    });

    it('fail to withdraw balance without available funds', async () => {
        await balances.deposit({ from: userAccount, value: 100 });

        try {
            await balances.withdraw(101, { from: userAccount });
        } catch (err) {
            return;
        }
        assert.fail("Should not allow overbalance withdraws");
    });

```

```
49      it('add liquidity', async () => {
50          assert.equal(parseInt(await balances.totalLiquidity()), 0, "
    Liquidity should be 0 wei");
51          await balances.addLiquidity({ from: adminAccount, value: 100 });
52          let liquidity = parseInt(await balances.totalLiquidity());
53
54          assert.equal(liquidity, 100, "Liquidity should be 100 wei");
55      });
56
57      it('withdraw liquidity with available funds', async () => {
58          await balances.addLiquidity({ from: adminAccount, value: 100 });
59          balances.withdrawLiquidity(100, { from: adminAccount });
60
61          let liquidity = parseInt(await balances.totalLiquidity());
62
63          assert.equal(liquidity, 0, "Liquidity should be 0 wei");
64      });
65
66      it('fail to withdraw liquidity without available funds', async () =>
        {
67          await balances.addLiquidity({ from: adminAccount, value: 100 });
68
69          try {
70              await balances.withdrawLiquidity(101, { from: adminAccount
    });
71          } catch (err) {
72              return;
73          }
74          assert.fail("Should not allow withdrawals without liquidity");
75      });
76
77
78      it('fail to withdraw liquidity for non admin user', async () => {
79          await balances.addLiquidity({ from: adminAccount, value: 100 });
80
81          try {
82              await balances.withdrawLiquidity(100, { from: userAccount })
    ;
83          } catch (err) {
84              return;
85          }
86          assert.fail("Should not allow withdrawals for non admin users");
87      });
88
89      it('fail to set bet holder without previous request', async () => {
90          await balances.setBetHolderAddress(adminAccount, { from:
    adminAccount });
91          try {
92              await balances.setBetHolderAddress(userAccount, { from:
    adminAccount });
93          } catch (err) {
94              return;
95          }
96          assert.fail("Should not allow to set a new bet holder address
    without a request");
97      })
98
99      it('fail to set bet holder with not enough time passed', async () =>
        {
```

```
100         await balances.setBetHolderAddress(adminAccount, { from:
    adminAccount });
101         await balances.requestUpdateBetHolderAddress(userAccount, { from
    : adminAccount });
102         try {
103             await balances.setBetHolderAddress(userAccount, { from:
    adminAccount });
104         } catch (err) {
105             return;
106         }
107         assert.fail("Should not allow to set a new bet holder address
    without a request");
108     });
109 });
```

B.4 BetOracle Tests

```
1  const BetOracle = artifacts.require("BetOracle");
2
3  const assert = require('assert');
4
5  contract('BetOracle', async accounts => {
6      let oracle;
7
8      let userAccount;
9      let adminAccount;
10     let operatorAccount;
11
12     beforeEach(async function() {
13         oracle = await BetOracle.new();
14
15         adminAccount = accounts[0];
16         userAccount = accounts[1];
17         operatorAccount = accounts[5];
18     });
19
20     it('winner should not be set', async () => {
21         assert.ok(!await oracle.isWinnerAvailable(), "Winner should not
    be available");
22         try {
23             await oracle.getWinnerSelection();
24         } catch (err){
25             return;
26         }
27         assert.fail("Should not be possible to get a winner that is not
    set");
28     });
29
30     it('fail to set winner with non admin account', async () => {
31         try {
32             await oracle.setWinner(1, {from: userAccount});
33         } catch (err){
34             return;
35         }
```

```
36          assert.fail("Should not be possible to set a winner without an
    admin account");
37      });
38
39      it('set winner', async () => {
40          oracle.setWinner(1, {from: adminAccount});
41          assert.ok(await oracle.isWinnerAvailable(), "Winner should be
    available");
42          assert.equal(parseInt(await oracle.getWinnerSelection()), 1, "
    Winner Selection ID is unexpected");
43      });
44
45      it('winner should not be changeable', async () => {
46          oracle.setWinner(1, {from: adminAccount});
47          try {
48              await oracle.setWinner(5);
49          } catch (err) {
50              return;
51          }
52          assert.fail("Winner should not be changeable");
53      });
54  });
```

B.5 Events Tests

```
1   const Events = artifacts.require("Events");
2   const AccessControl = artifacts.require("AccessControl");
3   const AlwaysZeroBetOracle = artifacts.require("AlwaysZeroBetOracle");
4   const assert = require('assert');
5
6   contract('Events', async accounts => {
7       let accessControl;
8       let events;
9       let oracle;
10
11      let userAccount;
12      let adminAccount;
13      let operatorAccount;
14
15      let event;
16      let selections;
17
18      beforeEach(async function() {
19          accessControl = await AccessControl.new();
20          events = await Events.new(accessControl.address);
21          oracle = await AlwaysZeroBetOracle.new();
22
23          adminAccount = accounts[0];
24          userAccount = accounts[1];
25          operatorAccount = accounts[5];
26
27          event = {
28              id: 14,
29              name: "testEvent",
30              betOracle: oracle.address,
```

```
31              isSettled: false
32          };
33
34          selections = [
35              {
36              name: "Selection 0",
37              odd: 300
38          },
39          {
40              name: "Selection 1",
41              odd: 150
42          }];
43
44          accessControl.addOperator(operatorAccount, {from: adminAccount})
    ;
45      });
46
47      it('create event', async () => {
48          await events.createActiveEvent(event, selections, {from:
    operatorAccount});
49          let activeEvent = await events.getEvent(event.id);
50          assert.equal(activeEvent.id, event.id, "Event IDs do not match")
    ;
51      });
52
53      it('create selections with event', async () => {
54          await events.createActiveEvent(event, selections, {from:
    operatorAccount});
55          let selectionsResult = await events.getSelections(event.id);
56          assert.equal(selectionsResult.length, 2, "Selection length is
    not as expected");
57      });
58
59      it('change odds', async () => {
60          let newOdds = 12341;
61          await events.createActiveEvent(event, selections, {from:
    operatorAccount});
62          await events.changeOdds(event.id, 0, newOdds, {from:
    operatorAccount});
63          let selectionsResult = await events.getSelections(event.id);
64          assert.equal(selectionsResult[0].odd, newOdds, "Event IDs do not
     match");
65      });
66
67      it('settle event', async () => {
68          await events.createActiveEvent(event, selections, {from:
    operatorAccount});
69          await events.setBetHolderAddress(adminAccount, {from:
    adminAccount});
70          await events.settleEvent(event.id, {from: adminAccount});
71          let eventShouldBeSettled = await events.getEvent(event.id);
72          assert.ok(eventShouldBeSettled.isSettled, "Event should be
    settled");
73          assert.ok(eventShouldBeSettled.isClosed, "Event should be closed
    ");
74      });
75
76      it('close event', async () => {
```

```
77          await events.createActiveEvent(event, selections, {from:
    operatorAccount});
78          await events.closeEvent(event.id, {from: operatorAccount});
79          let eventShouldBeSettled = await events.getEvent(event.id);
80          assert.ok(eventShouldBeSettled.isClosed, "Event should be closed
    ");
81      });
82  });
```

B.6 BetHolder Tests

```
1   const Balances = artifacts.require("Balances");
2   const Events = artifacts.require("Events");
3   const BetHolder = artifacts.require("BetHolder");
4   const AccessControl = artifacts.require("AccessControl");
5   const AlwaysZeroBetOracle = artifacts.require("AlwaysZeroBetOracle");
6   const BetOracle = artifacts.require("BetOracle");
7   const assert = require('assert');
8
9   contract('BetHolder', async accounts => {
10      let accessControl;
11      let events;
12      let oracle;
13      const initialLiquidity = 100;
14      const initialUserBalance = 10;
15
16      let userAccount;
17      let adminAccount;
18      let operatorAccount;
19
20      let event;
21      let selections;
22
23      beforeEach(async function() {
24          accessControl = await AccessControl.new();
25          events = await Events.new(accessControl.address);
26          oracle = await AlwaysZeroBetOracle.new();
27          balances = await Balances.new(accessControl.address);
28          betHolder = await BetHolder.new(events.address, balances.address
    , accessControl.address);
29
30          adminAccount = accounts[0];
31          userAccount = accounts[1];
32          operatorAccount = accounts[5];
33
34          event = {
35              id: 14,
36              name: "testEvent",
37              betOracle: oracle.address,
38              isSettled: false
39          };
40
41          selections = [
42              {
43              name: "Selection 0",
```

```
44              odd: 300
45          },
46          {
47              name: "Selection 1",
48              odd: 150
49          }];
50
51          await accessControl.addOperator(operatorAccount, {from:
    adminAccount});
52          await events.createActiveEvent(event, selections, {from:
    operatorAccount});
53          await balances.deposit({from: userAccount, value:
    initialUserBalance});
54          await balances.addLiquidity({from: adminAccount, value:
    initialLiquidity});
55          await balances.setBetHolderAddress(betHolder.address, {from:
    adminAccount});
56          await events.setBetHolderAddress(betHolder.address, {from:
    adminAccount});
57      });
58
59      it('make simple bet', async () => {
60          let valueToBet = 5;
61          let selectionId = 0;
62          await betHolder.makeBet(event.id, selectionId, valueToBet, {from
    : userAccount});
63
64          // Check bet was registered
65          let bets = await betHolder.getBetsByEventId(event.id);
66          assert.equal(bets.length, 1, "Bets length should be 1");
67
68          // Check liquidity was updated
69          let potentialWin = selections[selectionId].odd * valueToBet /
    100 - valueToBet
70          let liquidity = parseInt(await balances.totalLiquidity());
71          assert.equal(initialLiquidity - potentialWin, liquidity, "
    Liquidity is different than expected");
72
73          // Check balance was updated
74          let balance = parseInt(await balances.getBalance(userAccount));
75          assert.equal(initialUserBalance - valueToBet, balance, "User
    balance is different than expected");
76      });
77
78      it('fail to make bet in unexisting selection', async () => {
79          let valueToBet = 5;
80          let selectionId = 5;
81          try {
82              await betHolder.makeBet(event.id, selectionId,
    initialUserBalance + 1, {from: userAccount});
83          } catch (err) {
84              return;
85          }
86          assert.fail("Should fail to make a bet");
87      });
88
89      it('fail to make bet in unexisting event', async () => {
90          let valueToBet = 5;
91          let selectionId = 0;
```

```
92          try {
93              await betHolder.makeBet(event.id + 1, selectionId,
    valueToBet, {from: userAccount});
94          } catch (err) {
95              return;
96          }
97          assert.fail("Should fail to make a bet");
98      });
99
100     it('fail to make bet in closed event', async () => {
101         let valueToBet = 5;
102         let selectionId = 0;
103
104         await events.closeEvent(event.id, {from: operatorAccount});
105         try {
106             await betHolder.makeBet(event.id, selectionId, valueToBet, {
    from: userAccount});
107         } catch (err) {
108             return;
109         }
110         assert.fail("Should fail to make a bet");
111     });
112
113     it('fail to make bet without enough user balance', async () => {
114         let selectionId = 0;
115         try {
116             await betHolder.makeBet(event.id, selectionId,
    initialUserBalance + 1, {from: userAccount});
117         } catch (err) {
118             return;
119         }
120         assert.fail("Should fail to make a bet");
121     });
122
123     it('fail to make bet without liquidity', async () => {
124         let valueToBet = 5;
125         let selectionId = 0;
126         await events.changeOdds(event.id, selectionId, 500000, {from:
    operatorAccount}); // Ensure there is no liquidity
127         try {
128             await betHolder.makeBet(event.id, selectionId, valueToBet, {
    from: userAccount});
129         } catch (err) {
130             return;
131         }
132         assert.fail("Should fail to make a bet");
133     });
134
135     it('fail to make bet with value 0', async () => {
136         let valueToBet = 0;
137         let selectionId = 0;
138         try {
139             await betHolder.makeBet(event.id, selectionId, valueToBet, {
    from: userAccount});
140         } catch (err) {
141             return;
142         }
143         assert.fail("Should fail to make a bet");
144     });
```

```
145
146     it('fail to make bet in stopped contract', async () => {
147         let valueToBet = 5;
148         let selectionId = 5;
149         await betHolder.setStopped(true, {from: adminAccount});
150         try {
151             await betHolder.makeBet(event.id, selectionId, valueToBet, {
    from: userAccount});
152         } catch (err) {
153            return;
154         }
155         assert.fail("Should fail to make a bet");
156     });
157
158     it('settle simple bet (user won)', async () => {
159         let valueToBet = 5;
160         let selectionId = 0;
161         await betHolder.makeBet(event.id, selectionId, valueToBet, {from
    : userAccount});
162         await betHolder.setWinnerForEvent(event.id);
163         await betHolder.settleEvent(event.id, {from: userAccount});
164
165         // Check bet was registered
166         let eventFound = await events.getEvent(event.id);
167         assert.ok(eventFound.isSettled, "Event should be settled");
168
169         // Check liquidity was set correctly (bet was won by the user)
170         let potentialWin = selections[selectionId].odd * valueToBet /
    100 - valueToBet
171         let liquidity = parseInt(await balances.totalLiquidity());
172         assert.equal(initialLiquidity - potentialWin, liquidity, "
    Liquidity is different than expected");
173
174         // Check balance was updated
175         let balance = parseInt(await balances.getBalance(userAccount));
176         assert.equal(initialUserBalance + potentialWin, balance, "User
    balance is different than expected");
177     });
178
179     it('settle simple bet (user lost)', async () => {
180         let valueToBet = 5;
181         let selectionId = 1;
182         await betHolder.makeBet(event.id, selectionId, valueToBet, {from
    : userAccount});
183         await betHolder.setWinnerForEvent(event.id);
184         await betHolder.settleEvent(event.id, {from: userAccount});
185
186         // Check bet was registered
187         let eventFound = await events.getEvent(event.id);
188         assert.ok(eventFound.isSettled, "Event should be settled");
189
190         // Check liquidity was set correctly (bet was lost by the user)
191         let liquidity = parseInt(await balances.totalLiquidity());
192         assert.equal(initialLiquidity + valueToBet, liquidity, "
    Liquidity is different than expected");
193
194         // Check balance was updated
195         let balance = parseInt(await balances.getBalance(userAccount));
```

```
196         assert.equal(initialUserBalance - valueToBet, balance, "User
    balance is different than expected");
197     });
198
199     it('fail to settle bet without result available', async () => {
200         noResultOracle = await BetOracle.new();
201         event.id = 151;
202         event.betOracle = noResultOracle.address; // The oracle doesn't
    have a winner set yet
203         await events.createActiveEvent(event, selections, {from:
    operatorAccount});
204
205         await betHolder.makeBet(event.id, 0, 1, {from: userAccount});
206         try {
207             await betHolder.setWinnerForEvent(event.id);
208             await betHolder.settleEvent(event.id, {from: userAccount});
209         } catch(err) {
210            return;
211         }
212         assert.fail("It should not be possible to settle a event");
213     });
214
215     it('fail to settle bet without setting result', async () => {
216         await betHolder.makeBet(event.id, 0, 1, {from: userAccount});
217         try {
218             //await betHolder.setWinnerForEvent(event.id);
219             await betHolder.settleEvent(event.id, {from: userAccount});
220         } catch(err) {
221            return;
222         }
223         assert.fail("It should not be possible to settle a event");
224     });
225
226     it('stop contract and resume it', async () => {
227         assert.ok(!await betHolder.isStopped(), "Contract should not be
    stopped");
228
229         await betHolder.setStopped(true, {from: adminAccount});
230         assert.ok(await betHolder.isStopped(), "Contract should be
    stopped");
231
232         await betHolder.setStopped(false, {from: adminAccount});
233         assert.ok(!await betHolder.isStopped(), "Contract should be
    resumed");
234     });
235
236     it('fail to stop contract for non admin account', async () => {
237         try {
238             await betHolder.setStopped(true, {from: userAccount});
239         } catch(err) {
240             assert.ok(!await betHolder.isStopped(), "Contract should be
    resumed");
241             return;
242         }
243         assert.fail("It should not be possible to stop the contract");
244     });
245
246     it('get user bets', async () => {
247         let valueToBet = 1;
```

```
248         let selectionId = 0;
249         await betHolder.makeBet(event.id, selectionId, valueToBet, {from
    : userAccount});
250         await betHolder.makeBet(event.id, selectionId, valueToBet, {from
    : userAccount});
251         await betHolder.makeBet(event.id, selectionId, valueToBet, {from
    : userAccount});
252         await balances.deposit({from: operatorAccount, value: valueToBet
    });
253         await betHolder.makeBet(event.id, selectionId, valueToBet, {from
    : operatorAccount}); // should not be returned
254
255         let userBetsNotSettled = await betHolder.getBetsByUser(
    userAccount);
256
257         assert.equal(userBetsNotSettled.length, 3, "User should have 3
    bets");
258         assert.equal(!userBetsNotSettled[0].isSettled, true, "User
    should not have settled bets");
259
260         await betHolder.setWinnerForEvent(event.id);
261         await betHolder.settleEvent(event.id, {from: userAccount});
262
263         let userBetsSettled = await betHolder.getBetsByUser(userAccount)
    ;
264         assert.equal(userBetsSettled.length, 3, "User should have 3 bets
    ");
265         assert.equal(userBetsSettled[0].isSettled, true, "User should
    have settled bets");
266     });
267
268
269 });
```