LUCAS ANTÔNIO LOPES NEVES

DESENVOLVIMENTO DE UMA BOLSA DE VALORES UTILIZANDO SOLIDITY

LAVRAS – MG
2023

LUCAS ANTÔNIO LOPES NEVES

DESENVOLVIMENTO DE UMA BOLSA DE VALORES UTILIZANDO SOLIDITY

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

Prof. Dr. Antônio Maria Pereira de Resende
Orientador

LAVRAS – MG
2023

LUCAS ANTÔNIO LOPES NEVES

DESENVOLVIMENTO DE UMA BOLSA DE VALORES UTILIZANDO SOLIDITY
DEVELOPMENT OF A STOCK EXCHANGE USING SOLIDITY

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

APROVADA em 26 de abril de 2023.
Prof. Dr. Antônio Maria Pereira de Resende UFLA
Prof. Dr. Joaquim Quinteiro Uchôa UFLA
Prof. Dr. Maurício Ronny de Almeida Souza UFLA

Prof. Dr. Antônio Maria Pereira de Resende
Orientador

LAVRAS – MG
2023

Dedico à minha esposa Yasmin Andrade Diniz Neves e aos meus pais, Marlene Aparecida Lopes Neves e Ronaldo Pimenta Neves.

AGRADECIMENTOS

Agradeço ao meu orientador Prof. Dr. Antônio Maria Pereira de Resende que me orientou desde os projetos de Iniciação Científica e me guiou no processo de aprendizagem sobre *blockchain*.

Agradeço a minha esposa Yasmin Andrade Diniz Neves que esteve ao meu lado durante a minha trajetória na universidade e esteve me apoiando nos momentos mais desafiadores.

Agradeço aos meus pais, Marlene Aparecida Lopes Neves e Ronaldo Pimenta Neves, por sempre me apoiarem em meus estudos.

RESUMO

Pretendeu-se, neste trabalho, desenvolver uma aplicação utilizando a linguagem *Solidity* da *blockchain Ethereum*. Tal desenvolvimento possuía como objetivo entender as vantagens e desvantagens do desenvolvimento de contratos inteligentes, em especial nos contratos inteligentes para a *Ethereum*, e entender as diferenças desta linguagem para as linguagens não específicas para *blockchains*. A tecnologia *blockchain* surgiu como uma forma de armazenamento e validação da criptomoeda *Bitcoin*. Atualmente a tecnologia está sendo utilizada em outras criptomoedas e em outras áreas como contratos inteligentes autoexecutáveis que podem substituir os cartórios. A aplicação desenvolvida fornece funcionalidades necessárias para o funcionamento de uma bolsa de valores e os usuários podem criar ordens de compra e vender ativos, enquanto o sistema processa as ordens realizando as transações. Para testar as funcionalidades, testes automatizados foram desenvolvidos a partir de testes unitários e testes de casos de uso. Conclui-se que a tecnologia *blockchain* possui vantagens como a imutabilidade dos dados e a descentralização. Além disso, ela possui desvantagens também como escalabilidade reduzida quando utiliza o mecanismo de consenso *Proof-of-Work* e baixa interoperabilidade nas primeiras gerações de blockchains. Portanto, a decisão de utilizar a tecnologia *blockchain* depende das características de cada projeto, sendo necessário avaliar sua aplicação para garantir que suas vantagens atendam adequadamente ao perfil do projeto, sem sofrer, em demasia, as desvantagens da tecnologia.

Palavras-chaves: *Blockchain*; *Web3*; Contratos inteligentes.

ABSTRACT

The aim of this work was to develop an application using the Solidity language of the Ethereum blockchain. Such development aimed to understand the advantages and disadvantages of developing smart contracts, especially in smart contracts for Ethereum, and to understand the differences between this language and non-blockchain specific languages. Blockchain technology emerged as a way of storing and validating the Bitcoin cryptocurrency. The technology is currently being used in other cryptocurrencies and in other areas such as self-executing smart contracts that can replace notaries. The developed application provides functionalities necessary for the operation of a stock exchange and users can create purchase orders and sell assets, while the system processes the orders and performs the transactions. To test the functionalities, automated tests were developed from unit tests and use case tests. It is concluded that blockchain technology has advantages such as data immutability and decentralization. In addition, it also has disadvantages such as reduced scalability when using the Proof-of-Work consensus mechanism and low interoperability in the first generations of blockchains. Therefore, the decision to use blockchain technology depends on the characteristics of each project, and it is necessary to evaluate its application to ensure that its advantages adequately meet the project's profile, without suffering too much from the technology's disadvantages.

Keywords: *Blockchain*; *Web3*; *Smart contracts*.

LISTA DE FIGURAS

Figura 1 - Estrutura de uma blockchain 13
Figura 2 - Tipos de uma blockchain 15
Figura 3 - Ataque da Corrida 16
Figura 4 - Ataque de 51% 16
Figura 5 - Declaração de um array em Solidity 20
Figura 6 - Declaração e exemplo de uso de mapping em Solidity 20
Figura 7 - Pacotes NPM instalados 23
Figura 8 - Exemplo de utilização do HardHat 24
Figura 9 - Struct que representa uma ordem 27
Figura 10 - Código da transferência de valores para a taxa da criação de ordens 28
Figura 11 - Struct que representa uma transação 28
Figura 12 - Código da transferência de valores no momento da criação da transação 29
Figura 13 - Estruturas auxiliares da transferência de valores 29
Figura 14 - Código para o depósito de valores no contrato 30
Figura 15 - Código de transferência de valores 30
Figura 16 - Código de saque de valores 31
Figura 17 - Código do modificador onlyOwner 31
Figura 18 - Comando para instalação das dependências do projeto 31
Figura 19 - Comando para instalação do framework HardHat 32
Figura 20 - Comando para criar um projeto com o framework HardHat 32
Figura 21 - Comando para compilar o projeto 32
Figura 22 - Comando para executar os testes do projeto 33
Figura 23 - Código de conexão com a carteira do usuário 35
Figura 24 - Teste de criação de ordem de compra do caso 8 35
Figura 25 - Teste de criação de ordem de venda do caso 8 36
Figura 26 - Teste de criação da transação do caso 8 36
Figura 27 - Teste unitário para a funcionalidade de depositar valores monetários 37
Figura 28 - Teste unitário para a funcionalidade de verificar a igualdade entre strings 37
Figura 29 - Teste unitário para a funcionalidade de criação de ordens 38
Figura 30 - Teste unitário para a funcionalidade de criação de transações 39
Figura 31 - Teste unitário para a funcionalidade de adição de ordens 40
Figura 32 - Teste unitário para a funcionalidade de adição de transação 41
Figura 33 - Teste unitário para a funcionalidade de verificar se existe conflito na criação da transação 42
Figura 34 - Teste unitário para a ordenação das ordens de venda 43
Figura 35 - Teste unitário para a ordenação das ordens de compra 44
Figura 36 - Teste para a funcionalidade de realizar a operação de criação transação 45
Figura 37 - Diretório dos testes 45

LISTA DE TABELAS

Tabela 1 - Comparação entre as linguagens e suas características 18
Tabela 2 - Tabela com os dados do primeiro caso de teste 46
Tabela 3 - Tabela com os dados do segundo caso de teste 46
Tabela 4 - Tabela com os dados do terceiro caso de teste 46
Tabela 5 - Tabela com os dados do quarto caso de teste 47
Tabela 6 - Tabela com os dados do quinto caso de teste 47
Tabela 7 - Tabela com os dados do sexto caso de teste 47
Tabela 8 - Tabela com os dados do sétimo caso de teste 48
Tabela 9 - Tabela com os dados do oitavo caso de teste 48
Tabela 10 - Tabela com os dados do nono caso de teste 48
Tabela 11 - Tabela com os dados do décimo caso de teste 49
Tabela 12 - Tabela com os dados do décimo primeiro caso de teste 49
Tabela 13 - Tabela com os dados do décimo segundo caso de teste 49
Tabela 14 - Tabela com os dados do décimo terceiro caso de teste 50
Tabela 15 - Tabela com os dados do décimo quarto caso de teste 50
Tabela 16 - Tabela com os dados do décimo quinto caso de teste 50
Tabela 17 - Tabela com os dados do décimo sexto caso de teste 51
Tabela 18 - Tabela com os dados do décimo sétimo caso de teste 51
Tabela 19 - Tabela com os dados do décimo oitavo caso de teste 52
Tabela 20 - Tabela com os dados do décimo nono caso de teste 52
Tabela 21 - Tabela com os dados do vigésimo caso de teste 52
Tabela 22 - Tabela com os dados do vigésimo primeiro caso de teste 53
Tabela 23 - Tabela com os dados do vigésimo segundo caso de teste 53
Tabela 24 - Tabela com os dados do vigésimo terceiro caso de teste 53
Tabela 25 - Tabela com os dados do vigésimo quarto caso de teste 54
Tabela 26 - Tabela com os dados do vigésimo quinto caso de teste 54
Tabela 27 - Tabela com os dados do vigésimo sexto caso de teste 54
Tabela 28 - Tabela com os dados do vigésimo sétimo caso de teste 55
Tabela 29 - Tabela com os dados do vigésimo oitavo caso de teste 55
Tabela 30 - Tabela com os dados do vigésimo nono caso de teste 56
Tabela 31 - Tabela com os dados do trigésimo caso de teste 56
Tabela 32 - Tabela com os dados do trigésimo primeiro caso de teste 56
Tabela 33 - Tabela com os dados do trigésimo segundo caso de teste 57
Tabela 34 - Tabela com os dados do trigésimo terceiro caso de teste 57
Tabela 35 - Tabela com os dados do trigésimo quarto caso de teste 57
Tabela 36 - Tabela com os dados do trigésimo quinto caso de teste 58
Tabela 37 - Tabela com os dados do trigésimo sexto caso de teste 58
Tabela 38 - Tabela com os dados do trigésimo sétimo caso de teste 59

SUMÁRIO

1 Introdução 11
2 Referencial Teórico 13
2.1 Blockchain 13
2.2 Bitcoin 17
2.3 Ethereum 18
2.4 HardHat 21
3 Metodologia 22
4 Embasamento 23
4.1 Aprendizado sobre Blockchain 23
4.2 Aprendizado sobre Solidity 23
4.3 Aprendizado da Framework 23
5 Levantamento de Requisitos 25
5.1 Requisitos não funcionais 25
5.2 Requisitos funcionais 25
6 Desenvolvimento 27
6.1 Modelagem da aplicação 27
6.2 Implementação 31
6.3 Testes 34
7 Conclusão 60
Referências 62

1 Introdução

Blockchain surgiu junto com a criptomoeda *Bitcoin*. O *blockchain* surgiu como uma forma de armazenar as transações da criptomoeda. As principais características da tecnologia é o armazenamento distribuído e a segurança. A validação dos dados é realizada pelos algoritmos de consenso que podem realizar essa verificação por meio de cálculos matemáticos complexos ou por uma eleição pseudoaleatória. Estas características fizeram da tecnologia *blockchain* uma das mais promissoras do mercado devido à sua aplicabilidade.

A *blockchain* é uma lista encadeada em que cada bloco é ligado ao bloco anterior por meio da hash do bloco anterior. Como cada modificação no bloco altera a hash do bloco, para realizar uma modificação em um bloco, deve-se modificar todos os blocos sucessores a este bloco. Dessa forma, qualquer fraude em uma *blockchain* torna a ação inviável economicamente pois qualquer ganho será menor do que o ônus da ação.

Segundo a IBM (2023), a tecnologia *blockchain* pode ser utilizada na transparência da cadeia de suprimentos, no mercado financeiro e no abastecimento de alimentos. A partir da tecnologia, um sistema pode ser desenvolvido para compartilhar dados entre os parceiros, o que pode dar mais transparência na cadeia de suprimentos. Por não possuir grandes barreiras de entrada, a tecnologia também é um grande aliado do mercado financeiro, valores monetários podem ser transferidos para qualquer pessoa no mundo que tenha conexão à internet. Registrando a produção e a distribuição de alimentos, a *blockchain* pode tornar os alimentos mais seguros, criando confiança, aumentando a visibilidade e a responsabilidade em cada etapa do processo de fornecimento de alimentos.

O objetivo deste trabalho é desenvolver uma aplicação em *Solidity*, a linguagem de programação da *blockchain Ethereum*, para aprender sobre as vantagens e desvantagens da utilização de contratos inteligentes e aprender a usar a linguagem *Solidity*, a fim de propiciar ao autor uma experiência com a linguagem.

Para atingir o objetivo deste trabalho, um contrato inteligente que simula uma bolsa de valores foi desenvolvido utilizando *Solidity* e disponibilizado no GitHub (Neves, 2023). A aplicação é capaz de criar ordens de venda e de compra e, quando ocorrer uma compatibilidade entre as ordens, é capaz de criar transações. Além disso, toda transferência de valores ocorre por meio do contrato inteligente através da *Ethereum*.

A partir da implementação da aplicação com o auxílio do framework *HardHat*, foi possível atingir o objetivo de implementar o contrato inteligente que atendesse aos requisitos. Além da implementação do contrato inteligente, testes automatizados foram desenvolvidos para garantir o funcionamento correto da aplicação.

A partir deste trabalho foi possível concluir que uma das vantagens da tecnologia *blockchain* é a segurança porque os registros armazenados na *blockchain* são imutáveis, dessa forma uma vez que algo é registrado esse registro não pode ser alterado. Os contratos inteligentes também são seguros porque, uma vez que ele foi registrado na *blockchain*, suas condições para se executar não podem ser alteradas, por isso são autoexecutáveis. A tecnologia é ideal para transferência de valores, por não possuir barreiras de entrada e por sua segurança, mas a confirmação dos pagamentos pode demorar alguns minutos, o que atrapalha a utilização da tecnologia em larga escala. Sistemas como a bolsa de valores desenvolvida neste trabalho

possui uma grande desvantagem que a torna inviável nesse cenário, a alta latência. Logo, deve-se observar os casos de uso e analisar as vantagens e desvantagens nestes cenários.

Este trabalho está estruturado em cinco seções. Na seção 2, trabalhos relacionados são apresentados e seus conceitos são demonstrados. Na seção 3, a metodologia do trabalho é apresentada. Na seção 4, o Desenvolvimento é apresentado, demonstrando todo o processo de desenvolvimento do trabalho. Na seção 5, as conclusões do trabalho são apresentadas.

2 Referencial Teórico

2.1 Blockchain

Em 2008, após a Crise Econômica Mundial que ocorreu neste ano, a desconfiança nas instituições financeiras motivou a criação da criptomoeda Bitcoin. A criptomoeda não foi a primeira moeda digital criada, outras moedas digitais foram criadas, mas eram incapazes de resolver o problema do gasto duplo. O gasto duplo ocorre quando uma moeda é utilizada em mais de uma transação. A solução para este problema foi a tecnologia utilizada para armazenar e validar as transações com bitcoins, a Blockchain.

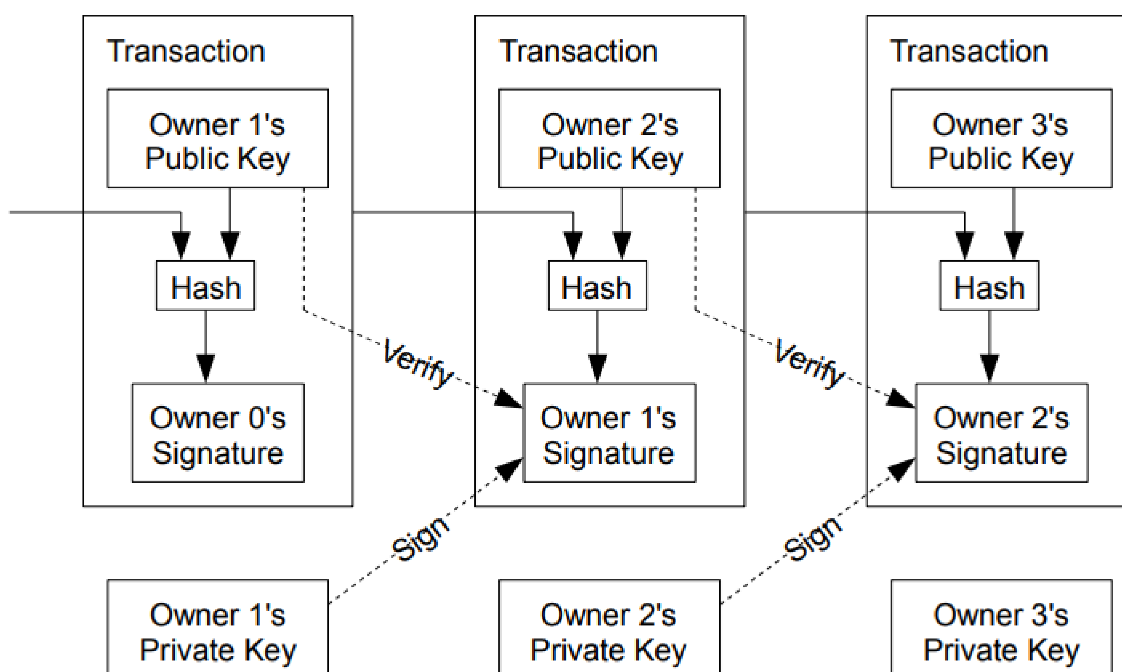

Figura 1 - Estrutura de uma blockchain

Na Figura 1, extraído do *whitepaper* do *Bitcoin* (Nakamoto, 2008), a estrutura de uma *blockchain* é exibida. A Blockchain é uma lista encadeada onde cada nó da lista é um bloco. Os blocos são encadeados entre si por meio da hash do bloco anterior, ou seja, um bloco de índice **X** possui a hash do bloco **X – 1**. A *hash* é um código alfanumérico que resume os dados do bloco, ou seja, qualquer alteração no bloco altera a sua *hash*. Os blocos também possuem o *timestamp* do momento da criação do bloco e cada bloco possui um *timestamp* maior do que o bloco anterior e um *timestamp* menor do que o *timestamp* do bloco sucessor. Por fim, cada bloco possui a transações que foram validadas no momento da criação do bloco.

Devido ao encadeamento das *hashes* e à ordem cronológica dos *timestamps* dos blocos, não é possível atualizar um bloco ou alterar a ordem dos blocos. Como as *hashes* resume os dados do bloco, se um bloco for alterado, sua *hash* também vai ser alterada e torna-se inválida porque o bloco posterior ainda possuiria a *hash* anterior. Para alterar o conteúdo de um bloco, todos os blocos anteriores a ele devem ser alterados, ou seja, quanto mais blocos possuir uma *blockchain*, mais segura será a *blockchain*. Como a ordem dos blocos também é verificada pela ordem cronológica dos *timestamps*, não é possível inserir um novo bloco no meio da *blockchain* ou alterar a ordem dos blocos.

Esse processo de validação dos blocos é realizado pelos algoritmos de consenso. Os nós da rede *blockchain* devem entrar em um consenso se o bloco é válido e, se for válido, o bloco é inserido na rede. Existem diferentes algoritmos de consenso, mas os dois principais são o algoritmo Prova de Trabalho (*Proof-of-Work*, PoW) e o algoritmo Prova de Participação (*Proof-of-Stake*, PoS).

No algoritmo *PoW*, o processo de validação é conhecido como mineração. No processo de mineração, o consenso é alcançado por meio da solução de problemas matemáticos complexos. Os nós validadores, conhecidos como mineradores, emprestam seu poder computacional para resolver os problemas matemáticos complexos. Em troca desse empréstimo, os mineradores recebem uma recompensa por participar da mineração. Essa recompensa é uma quantidade de criptomoedas e a quantidade varia de acordo com a *blockchain*.

No algoritmo *PoS*, o consenso é alcançado por meio de uma eleição pseudoaleatória. Os nós candidatos a validador devem apostar as suas moedas. Quando mais moedas apostadas, maior a chance de o nó ser escolhido como o nó validador. Para evitar que apenas nós com muitas moedas sejamos escolhidos, o algoritmo também leva em consideração a idade das moedas. Não existe uma recompensa na criação do bloco, mas o validador do bloco recebe as taxas das transações adicionadas ao bloco.

Apesar do algoritmo *PoW* ser muito seguro, o custo energético para o seu funcionamento é alto e o *PoS* tem se tornado uma alternativa (Blockgeeks, 2023). O custo energético para validar uma transação *Bitcoin* (criptomoeda que utiliza o *PoW*) é o mesmo para abastecer 1,5 lares americanos por dia. Todas as transações *Bitcoin* consomem o mesmo do que que a Dinamarca.

A rede de uma *blockchain* é composta por três tipos de nós: nós validadores, nós completos e nós leves. Os nós validadores são os nós responsáveis por validar as transações de uma *blockchain*. Os nós completos possuem uma cópia completa da rede. Por isso são responsáveis por distribuir cópias da rede e auxiliar os validadores no processo de validação da rede. Os nós leves são nós que mantêm uma cópia de apenas uma parte da rede, apenas a parte necessária para o seu funcionamento. Por exemplo, os aplicativos *mobile* de carteiras de criptomoedas são nós leves porque possuem apenas as informações necessária para seu funcionamento, como o cabeçalho do bloco de transações anteriores para validar a *blockchain* e passar as informações para outros nós.

O Trilema da Escalabilidade (Wikipédia, 2018), criado por Vitalik Buterin, sugere que uma *blockchain* pode possuir apenas duas das três seguintes características: descentralização, escalabilidade e segurança. A descentralização de uma *blockchain* se refere ao grau de diversificação em posse, influência e valor na *blockchain*. A escalabilidade de uma *blockchain* se refere à capacidade de processamento de transações da *blockchain*. Quanto maior sua capacidade de processamento, maior será sua escalabilidade. A segurança de uma *blockchain* se refere à capacidade de uma *blockchain* se defender de ataques externos. Essa limitação deve ser considerada durante o desenvolvimento e implementação de *blockchains*. Soluções como um algoritmo de consenso mais eficiente ou o uso de uma *blockchain* secundária para transações pequenas podem resolver o trilema, mas essas soluções criam suas próprias limitações.

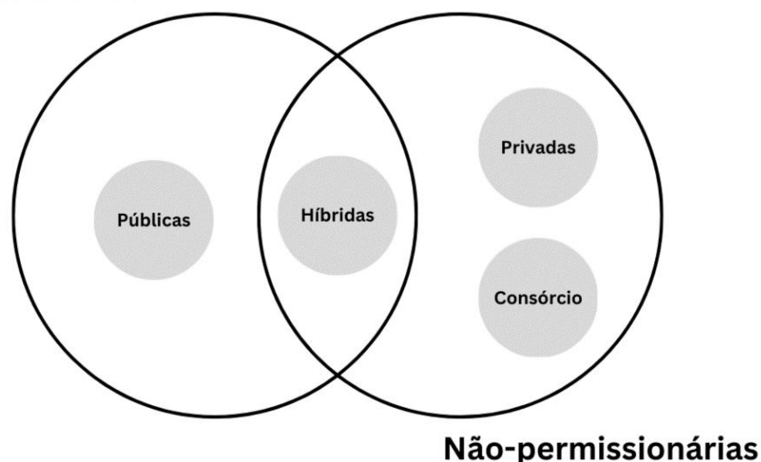

Figura 2 - Tipos de uma blockchain

Na Figura 2, os diferentes tipos de uma *blockchain* são exibidos. Uma *blockchain* pode ser classificada de diferentes formas (IBM, 2022). As *blockchains* permissionárias exigem uma permissão para um novo usuário, enquanto as *blockchains* não-permissionárias não exigem essa permissão. As *blockchains* privadas possuem uma autoridade central no controle, enquanto as *blockchains* públicas não possuem uma autoridade central no controle. As *blockchains* de consórcio são controladas por um grupo de organizações. As *blockchains* híbridas possuem uma autoridade central no controle, mas não exige uma permissão para um novo usuário.

Uma *blockchain* possui sete princípios: integridade na rede, poder distribuído, valor como incentivo, segurança, privacidade, direitos preservados e inclusão (TAPSCOTT e TAPSCOTT, 2017). Os princípios são explicados a seguir:

- Integridade na rede: uma *blockchain* deve ser confiável;
- Poder distribuído: o poder na *blockchain* deve estar distribuído entre seus nós;
- Valor como incentivo: uma *blockchain* deve retribuir os nós validadores pela validação;
- Segurança: uma *blockchain* deve ser segura.
- Privacidade: o nó de uma *blockchain* deve ser anônimo e ser identificado apenas pelo seu endereço;
- Direitos preservados: um registro armazenado na *blockchain* não pode ser alterado;
- Inclusão: a entrada de um novo usuário na *blockchain* deve ser facilitada.

Para se avaliar uma *blockchain*, existem duas métricas principais: *hash power* e *hash rate*. *Hash Power* é o custo computacional para processar todas as *hashes* de uma *blockchain*. Quanto maior o *hash power*, mais segura será a *blockchain* porque maior será o custo computacional para fraudar a rede. *Hash rate* é a quantidade de *hashes* processadas por segundo. Quanto maior o *hash rate*, mais escalável será a *blockchain* porque maior será a capacidade da *blockchain* de processar as hashes.

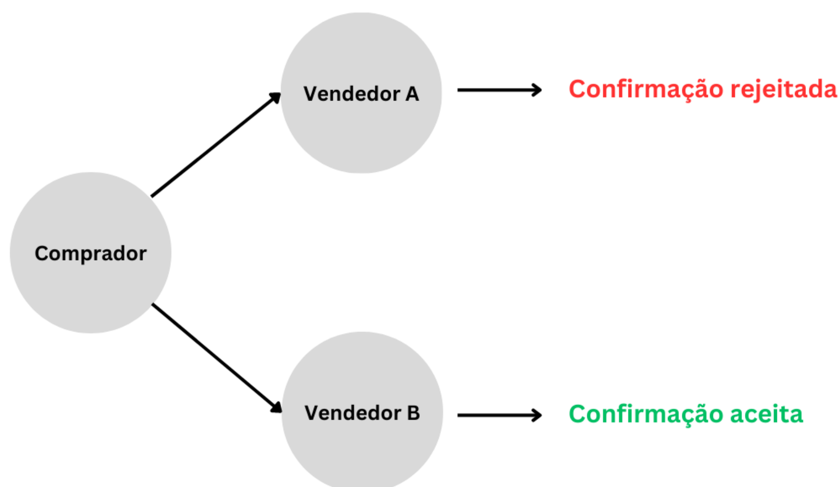

Figura 3 - Ataque da Corrida

Na Figura 3, o Ataque da Corrida é exibido. Este é um dos principais ataques que podem ocorrer a uma *blockchain*. Neste ataque, um comprador envia uma mesma moeda para diferentes vendedores esperando que os vendedores logo aprovem a transação. Se mais de um vendedor aprovar a transação, apenas um dos vendedores receberá moeda. Recomenda-se que os destinatários das moedas em transações de *blockchain* espere pelo menos seis validações (Cardoso, 2018), cada validação equivale a um bloco validado e escrito na *blockchain*. Por exemplo, a *blockchain* da *Bitcoin* valida um bloco a cada dez minutos (*Bitcoin Wiki*¸ 2019), logo os destinatários devem esperar pelo menos sessenta minutos antes de considerar a transação validada.

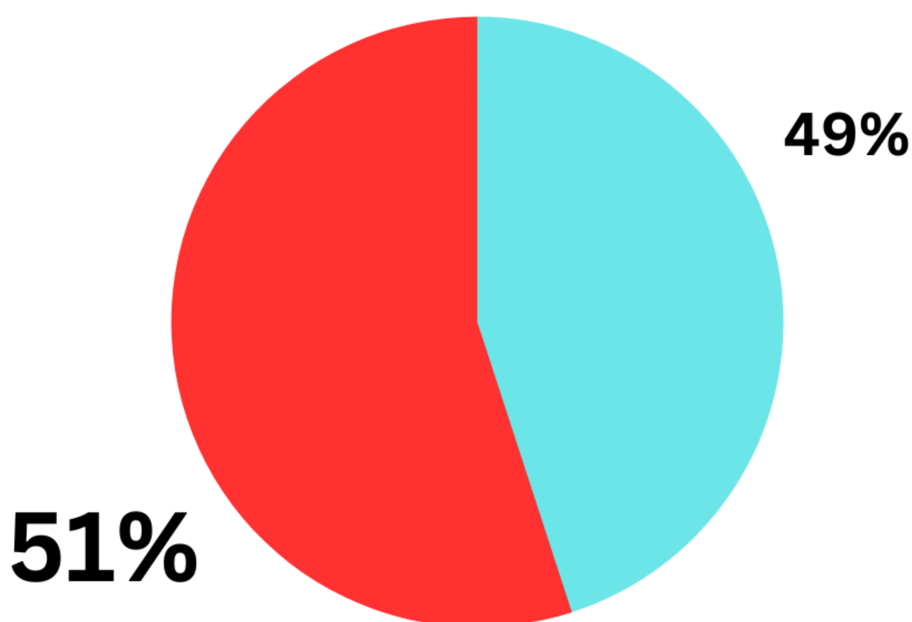

Figura 4 - Ataque de 51%

Na Figura 4, o Ataque de 51% é apresentado. Este ataque ocorre de maneiras diferentes dependendo de qual algoritmo de consenso a *blockchain* utilizar. Em *blockchains* que utilizam

o algoritmo *PoW*, o agente fraudador atinge mais da metade da capacidade de processamento da *blockchain*, ou seja, o agente fraudador teria poder computacional maior que o restante da rede. Em *blockchains* que utilizam o algoritmo PoS, o agente fraudador atinge mais da metade das moedas da *blockchain*, ou seja, o fraudador teria mais moedas do que o restante da rede. Em ambos os casos, o agente fraudador vai ser capaz de validar suas próprias transações sendo possível realizar o gasto duplo.

As *blockchains* possuem três gerações. A primeira geração é formada por *blockchains* criadas focadas em criptomoedas. A segunda geração de *blockchains* é formada por *blockchains* criadas para utilização de contratos inteligentes. A terceira geração de *blockchains* surgiu para solucionar os problemas das gerações anteriores: o custo do algoritmo PoW e a interoperabilidade entre as *blockchains*.

Para resolver o problema do custo do algoritmo *PoW*, a terceira geração de *blockchains* busca migrar as *blockchains* anteriores para o algoritmo *PoS*. Como ocorreu com a *blockchain* do *Ethereum* no dia 15 de setembro de 2022, conhecido com o *The Merge* (Ethereum, 2023). O algoritmo *PoS* possui um custo para seu funcionamento menor, mas ainda mantém a *blockchain* segura.

A interoperabilidade é a incapacidade das *blockchains* de se comunicarem entre si. A interoperabilidade reduz a adoção das *blockchains* pelo usuário comum porque o usuário em uma *blockchain* não consegue se comunicar com o usuário em outra *blockchain*. Para resolver esse problema, a terceira geração das *blockchains* busca criar um ecossistema onde as *blockchains* se comuniquem, tornando as *blockchains* mais acessíveis aos usuários.

2.2 Bitcoin

A primeira utilização de uma *blockchain* foi com a criptomoeda *Bitcoin* (Nakamoto, 2008). O *Bitcoin* é uma criptomoeda que foi criada por *Satoshi Nakamoto* em 2008, sua identidade é desconhecida e pouco se sabe se esse nome representa uma pessoa ou um grupo de pessoas. Como o *Bitcoin* foi criado logo após a Crise Econômica de 2008, a ideia por trás da criptomoeda era criar uma forma de realizar transações sem a necessidade de um terceiro. A ideia é que a transação ocorra somente entre o remetente e o destinatário da transação, ou seja, sem a necessidade de um banco ou de uma outra instituição financeira.

Como descrito no *whitepaper* do *Bitcoin* (Nakamoto, 2008), a *blockchain* do *Bitcoin* utiliza o algoritmo *Proof-of-Work*. A utilização desse algoritmo pela *blockchain* é uma de suas vantagens, mas também uma de suas desvantagens, porque o algoritmo é seguro, mas o custo para seu funcionamento é elevado, devido ao custo computacional para validar as transações. Como o *Bitcoin* é descentralizado, não existe uma organização no seu controle, ele se enquadra no Trilema da Escalabilidade (Wikipédia, 2018), ou seja, a sua *blockchain* é segura e descentralizada, mas não é escalável.

A recompensa dada aos mineradores da rede é reduzida pela metade a cada 210 mil blocos minerados em um evento chamado *halving*. No início do Bitcoin, a recompensa era 50 BTC. No primeiro *halving*, em 2012, o valor foi reduzido para 25 BTC. No segundo *halving*, em 2016, o valor foi reduzido para 12,5 BTC. No terceiro *halving*, em 2020, o valor foi reduzido para 6,25 BTC. O próximo *halving* deve acontecer em 2024.

2.3 Ethereum

A *blockchain Ethereum* foi criada por *Vitalik Buterin* em 2015 (Wikipédia, 2022). A *blockchain* surgiu com o grande diferencial utilizar contratos inteligentes. No início, a *Ethereum* utilizava o algoritmo *PoW*, mas passou a utilizar o *PoS* a partir do dia 15 de setembro de 2022, no chamado *The Merge* (Ethereum, 2023).

Os contratos inteligentes (ou *smart contracts*) são programas armazenados em uma *blockchain* que são autoexecutáveis. As partes do contrato entram em acordo sobre as condições do contrato e publicam o contrato em uma *blockchain*. Quando as condições forem atendidas, o contrato é autoexecutado independentemente da interferência de uma das partes ou de um terceiro.

Como tudo que é registrado na blockchain não pode ser alterado, os contratos inteligentes não podem ser alterados uma vez que eles são registrados. Para permitir que um contrato seja atualizado, é necessário implementar um outro contrato com o padrão de projeto Adaptador. O contrato adaptador armazena o endereço do contrato com as implementações e, a partir de uma função pública, é possível alterar este endereço. Por medidas de segurança, apenas o proprietário do contrato adaptador pode chamar esta função pública. Sempre que o contrato com as implementações for alterado, ele deverá ser implantado novamente na rede blockchain, um novo endereço para o contrato de implementações será criado e o contrato adaptador deverá ser chamado para atualizar o endereço. O usuário consultará o endereço do contrato a partir do contrato adaptador.

Para exemplificar a aplicação de contratos inteligentes, digamos que o sistema de uma imobiliária seja registrado em uma blockchain e seu funcionamento seja gerido por contratos inteligentes. Uma vez que o inquilino pague o aluguel pela blockchain, o contrato inteligente transfere a taxa da imobiliária para a carteira da imobiliária e o restante do valor para a carteira do dono do imóvel. O mesmo sistema pode ser implementado para um sistema de compras online, onde o vendedor recebe o valor da compra após o recebimento do produto por parte do comprador.

A linguagem utilizada para desenvolver os contratos inteligentes para a Ethereum é a linguagem Solidity. A linguagem Solidity foi inspirada pelas linguagens C++, Javascript e Python, logo escrever em Solidity se assemelha a escrever nessas outras linguagens.

Tabela 1 - Comparação entre as linguagens e suas características

Características	**Linguagens**			
	Solidity	**C++**	**Javascript**	**Python**
Compilação	Compilada	Compilada	Interpretada	Interpretada
Tipagem	Estática	Estática	Dinâmica	Dinâmica
Comandos de seleção	if, else	if, else	if, else	if, else, elif
Comandos de repetição	while, do/while, for	while, do/while, for	while, do/while, for, for/in, for/of	while, while/else, for, for/in
Escopo	global, local, de contrato e de função	de classe, de função, de bloco e global	global, local	global, bloco

Generics	Não	Sim	Não	Sim
Reflection	Não	Não	Sim	Sim

Na Tabela 1, uma comparação entre a Solidity e as linguagens utilizadas como inspiração é apresentada. Na tabela, as linguagens Solidity, C++, Javascript e Python são comparadas pelas seguintes características: compilação, tipagem, comandos de seleção, comandos de repetição, escopo, *generics* e *reflection*.

A linguagem compilada é aquela linguagem que o código escrito pelo programador é traduzido para um arquivo executável antes de ser executado pelo computador. A linguagem interpretada é aquela linguagem que o código escrito pelo programador é executado diretamente pelo computador, sem precisar passar por uma compilação prévia. A Solidity é uma linguagem compilada assim como C++, enquanto Javascript e Python são linguagens interpretadas.

Na tipagem estática, as variáveis têm seus tipos de dados declarados no momento do desenvolvimento e os tipos de dados não alteram. Na tipagem dinâmica, as variáveis têm seus tipos de dados declarados no momento do desenvolvimento e os tipos de dados podem alterar. A Solidity possui tipagem estática assim como C++, enquanto Javascript e Python possuem tipagem dinâmica.

Os comandos de seleção são uma estrutura de controle que permite que diferentes ações ou trechos de códigos sejam executados dependendo de uma condição específica. Os comandos de seleção são semelhantes entre as linguagens, com exceção da linguagem Python que além dos comandos que as outras linguagens possuem, também possui o comando *elif*.

Os comandos de repetição são uma estrutura de controle que permite que determinado bloco de código seja executado repetidas vezes até que uma condição seja atendida. Os comandos de repetição são semelhantes entre as linguagens, com exceção da linguagem Python que possui todos os comandos que Solidity e C++ possuem, mas também possui o comando *for/in* e com exceção da linguagem Javascript que possui todos os comandos das outras linguagens e o comando *for/of*.

O escopo se refere à região do código onde uma determinada variável ou função pode ser acessada, ou seja, o contexto em que uma variável ou função pode ser usada dentro de um programa. As linguagens se diferem com relação aos seus escopos. A Solidity possui os seguintes escopos: global, local, de contrato e de função. No escopo global, as variáveis declaradas fora de uma função podem ser acessadas por todo o contrato. No escopo local, as variáveis declaradas dentro de uma função podem ser acessadas apenas dentro da função. No escopo de contrato, as variáveis e funções declaradas no nível do contrato podem ser acessadas em todas as funções deste contrato. No escopo de função, as variáveis e funções declaradas dentro de uma função específica só podem ser acessadas dentro dessa função.

Generics são uma técnica empregada por algumas linguagens de programação para permitir a definição de classes, interfaces e métodos sem a especificação de um tipo específico (Oracle, 2023). Com isso, é possível utilizá-los com diferentes tipos de dados, o que confere maior flexibilidade e reutilização ao código. Solidity e Javascript não possuem *Generics*, enquanto C++ e Python possuem.

Reflection é uma funcionalidade presente em algumas linguagens de programação que permite que um programa possa analisar e modificar sua própria estrutura e comportamento

durante a execução (MISRA; KULKARNI, 2017). Solidity e C++ não possuem Reflection, enquanto Javascript e Python possuem.

Como os códigos em *Solidity* são executados na *Ethereum*, operação realizada na *blockchain* tem um custo, conhecido como *gas*. Segundo a documentação da *Ethereum*, o remetente de uma transação deve pagar uma quantidade de *ethers* que corresponde ao preço do *gas* multiplicado pelo consumo de *gas* da transação (Ethereum, 2023). A cobrança de *gas* é uma forma de evitar abusos na rede, uma vez que essa cobrança incentiva os desenvolvedores a desenvolverem contratos mais eficientes. Os *ethers* também podem ser utilizados para movimentações financeiras entre os usuários e os contratos. O *Solidity* também possui uma limitação quanto ao tamanho dos contratos. Os contratos não podem passar de 24576 bytes.

A linguagem Solidity possui duas estruturas de dados: mapping e array. O array em Solidity funciona semelhante aos arrays de outras linguagens, ou seja, como uma lista encadeada.

```
string [] disciplinas = ["Estruturas de dados", "Algoritmos em Grafos"]
```

Figura 5 - Declaração de um array em Solidity

Na Figura 5, a declaração de um array é apresentada. A tipagem do array deve ser declarada em sua declaração porque Solidity é uma linguagem estaticamente tipificada. Além disso, os métodos de um array em Solidity são os seguintes:

- push: para adicionar um elemento ao final do array;
- pop: para remover o último elemento e retorná-lo;
- length: para retornar quantidade de elementos do array.

Os array em *Solidity* possuem dois tipos: *array* de tamanho fixo e *array* de tamanho dinâmico. O array de tamanho fixo é declarado já com seu tamanho e com os elementos desejados e nenhum outro elemento pode ser adicionado. O array de tamanho dinâmico é declarado apenas com sua tipagem e os elementos são adicionados posteriormente com o método *push*.

```
mapping(string => string) disciplinas;
displinas["ED"] = "Estruturas de Dados";
```

Figura 6 - Declaração e exemplo de uso de mapping em Solidity

Na Figura 6, a declaração e o uso de um *mapping* em *Solidity* é apresentada. O *mapping* é uma estrutura de chave e valor, semelhante a uma tabela hash ou aos dicionários de Python. As tipagens da chave e do valor deve ser declarado no momento da declaração do *mapping* e o uso do *mapping* é inserindo o valor na sua chave como apresentado na Figura 6. Além do array e do mapping, Solidity possui alguns tipos primitivos de dados:

- **address:** endereço da carteira;
- **bool:** booleano (verdadeiro ou falso);
- **int:** números inteiros;
- **uint:** números inteiros positivos;
- **string:** texto simples;
- **bytes:** texto com tamanho específico;
- **struct:** estrutura de dados;

- **enum:** variáveis categóricas.

Os valores inteiros (*int*) e os valores inteiros positivos (*uint*) também podem variar quanto ao tamanho em bits que o dado pode armazenar, podendo ser de 8 bits a 256 bits.

2.4 HardHat

Antes de iniciar o desenvolvimento em Solidity, como em qualquer outra linguagem, deve-se pensar em formas de executar os testes e de realizar a implantação do contrato (o *deployment*). Além disso, no desenvolvimento para *blockchains*, existe a necessidade de utilizar *blockchains* de teste, como carteira de testes com saldo de criptomoedas positivo. Para realizar essas tarefas, existem duas ferramentas (*frameworks*) no ambiente Solidity, os *frameworks HardHat* e *Truffle*.

O *HardHat* é mais vantajoso que o *Truffle* porque o *HardHat* possui uma comunidade maior e mais utilizado pelo mercado de trabalho. Essa preferência pelo *HardHat* pode ser comprovada considerando a quantidade de downloads dos pacotes dos *frameworks* no site NPM. No dia 3 de outubro de 2022, o pacote do HardHat possuía 142.049 downloads semanais enquanto o pacote do Truffle possuía 29.413 downloads semanais (NPMJS, 2023).

3 Metodologia

Para consecução do objetivo, este trabalho foi dividido em três fases. Na primeira fase, denominada Embasamento, onde o autor buscou aprender mais sobre as tecnologias e ferramentas que seriam utilizadas. Na segunda fase, denominada Levantamento de Requisitos, o autor mapeou todos os requisitos necessários para a aplicação funcionar corretamente. Na terceira fase, denominada Desenvolvimento, o autor modelou a aplicação, implementou utilizando Solidity e realizou os testes para garantir o funcionamento da aplicação.

A fase Embasamento foi dividida em três etapas descritas a seguir:

- **Aprendizado sobre *Blockchain*:** Nesta fase, o autor leu artigos e livros sobre *blockchain* objetivando aprender mais sobre a tecnologia e entender suas vantagens e desvantagens.
- **Aprendizado sobre *Solidity*:** Nesta fase, o autor instalou o *Solidity*, aprendeu os conceitos mínimos, necessários e suficientes da linguagem para poder implementar a aplicação proposta.
- **Aprendizado de *Framework*:** Nesta fase, o autor aprendeu como utilizar o *framework HardHat* para auxiliar o desenvolvimento da aplicação.

A fase de Levantamento de Requisitos foi dividida em X etapas descritas a seguir:

- **Requisitos não funcionais:** Nesta etapa, o autor buscou mapear todos os requisitos não funcionais necessários para o funcionamento da aplicação.
- **Requisitos funcionais:** Nesta etapa o autor buscou mapear todos os requisitos funcionais necessários para o funcionamento da aplicação.

A fase Desenvolvimento foi dividida em três etapas descritas a seguir:

- **Requisitos da aplicação:** Nesta fase, o autor mapeou os requisitos da aplicação.
- **Modelagem da aplicação:** Nesta fase, o autor modelou a aplicação de acordo com as estruturas de dados do *Solidity*.
- **Implementação:** Nesta fase, o autor implementou os requisitos mapeados utilizando a linguagem *Solidity*.
- **Teste:** Nesta fase, o autor desenvolveu testes automatizados para verificar o correto funcionamento da aplicação.

4 Embasamento

4.1 Aprendizado sobre Blockchain

O aprendizado sobre *blockchain* se iniciou nos dois projetos de iniciação científica que o autor participou. O primeiro projeto consistia em avaliar a qualidade de um sistema *blockchain*, a fim de verificar possíveis gargalos em sua execução e dificuldades de desempenho e controle de qualidade. O segundo projeto consistia em criar critérios e comparar duas ferramentas *blockchain* para fornecer mecanismos para avaliação e seleção de ferramentas *blockchain*.

Para realizar este trabalho, o autor apoiou-se no livro *Blockchain Revolution* dos autores *Don Tapscott* e *Alex Tapscott* (TAPSCOTT e TAPSCOTT, 2017), na documentação da *Ethereum* (*Ethereum Foundation*, 2023) e da *Solidity* (Solidity, 2023).

Os principais resultados dessa fase estão descritos no referencial teórico, onde os principais conceitos sobre *blockchain* foram apresentados.

4.2 Aprendizado sobre Solidity

A linguagem *Solidity* foi inspirada pelas linguagens *C++*, *Javascript* e *Python*, logo escrever em *Solidity* se assemelha a escrever nessas outras linguagens. Para aprender sobre a linguagem *Solidity*, o autor se baseou na documentação da linguagem (Solidity, 2023) e nos exemplos que a documentação apresenta. Uma vez que já conhecia as outras linguagens, o processo de aprendizagem da linguagem demorou poucas semanas.

```
"@nomiclabs/hardhat-ethers": "^2.0.0",
"@nomiclabs/hardhat-waffle": "^2.0.0",
"chai": "^4.2.0",
"chance": "^1.1.9",
"ethereum-waffle": "^3.0.0",
"ethers": "^5.0.0",
"hardhat": "^2.9.9"
```

Figura 7 - Pacotes NPM instalados

Alguns pacotes do NPM foram instalados para auxiliar o desenvolvimento da aplicação neste projeto. Pacotes para testes e pacotes relacionados ao *framework* de desenvolvimento foram instalados. Os pacotes *chai* e *chance* são pacotes relacionados a testes e os outros são relacionados ao framework de desenvolvimento.

4.3 Aprendizado do Framework

O aprendizado sobre o *framework HardHat* se baseou em sua documentação (HardHat, 2023). Os exemplos contidos na documentação serviram como base para uma maior compreensão sobre o funcionamento do framework.

```
const { ethers, waffle } = require('hardhat');
const provider = waffle.provider;
async function getBalancesFromAddress(accounts) {
  for (const account of accounts) {
    const balance0ETH = await provider.getBalance(account.address);

    console.log(account.address, '\t', +balance0ETH);
  }
}
await this.exchange.connect(this.accounts[i + 1]).depositMoney({
  value: ethers.utils.formatUnits(
    String(
      order.isPassive
        ? Math.ceil(order.value * 1.01)
        : Math.ceil(order.value * 1.02)
    ),
    'wei'
  ),
});
```

Figura 8 - Exemplo de utilização do HardHat

Na Figura 8, um exemplo sobre a utilização do HardHat durante os testes é apresentado. No código apresentado, o *framework* é utilizado para consultar os saldos das carteiras da *Ethereum* e para movimentação financeira entre as carteiras.

5 Levantamento de Requisitos

5.1 Requisitos não funcionais

Os requisitos não funcionais são identificados pelo código RNF. A seguir são descritos os requisites não funcionais:

- RNF1 – A aplicação deve ser desenvolvida em Solidity.
- RNF2 – O projeto deve utilizar o Yarn.
- RNF3 – O projeto deve ser compatível com os sistemas operacionais Windows 11 e Ubuntu 20.
- RNF4 – A aplicação deve ter uma latência de até um segundo.

5.2 Requisitos funcionais

Para contextualizar os requisitos funcionais alguns conceitos utilizados no projeto precisam ser descritos. Esses conceitos são descritos a seguir:

- Ordens: instrução dada por um usuário para comprar ou vender um ativo financeiro.
- Ordens de compra: ordem com o objetivo de compra de um ativo financeiro.
- Ordens de venda: ordem com o objetivo de compra de um ativo financeiro.
- Ordens fracionadas: ordem que podem ser realizada de forma fracionada, ou seja, se a ordem tiver compatibilidade com uma outra ordem com maior ou menor número de cotas, a operação deve ser realizada com as cotas com compatibilidade e uma ordem com o restante das cotas deve ser criada.
- Ordens ativas: ordem que executa a rotina de verificação de compatibilidade no momento da criação da ordem.
- Ordens passivas: ordem que não executa a rotina de verificação de compatibilidade.
- Transação: operação criada após uma ordem de compra e uma ordem de venda tiverem compatibilidade.

Os requisitos funcionais são identificados pelo código RF. A seguir são descritos os requisitos funcionais:

- RF1 – Deve ser possível cadastrar ordens de compra.
- RF2 – Deve ser possível cadastrar ordens de venda.
- RF3 – Deve ser possível selecionar, no momento da criação da ordem, se a ordem será fracionada.
- RF4 – Deve ser possível selecionar, no momento da criação da ordem, se a ordem será ativa ou passiva.
- RF5 – Não deve ser possível excluir uma ordem.
- RF6 – Não deve ser possível atualizar uma ordem.
- RF7 – A ordem apenas deve ser alterada para dizer que a ordem está desativada quando ela passar pelo processo de criação da transação.
- RF8 – Deve ser possível consultar as ordens.
- RF9 – Deve ser possível criar uma transação.

- RF10 – A movimentação financeira, relacionada às ordens ou às transações, deve ocorrer apenas por meio da *Ethereum*.

6 Desenvolvimento

6.1 Modelagem da aplicação

Para a aplicação funcionar como uma bolsa de valores utilizando o *Solidity*, na aplicação existe ordens de compra e de venda. As ordens se dividem em três estruturas de dados: um *array* com todas as ordens e dois *mapping* apenas com as posições das ordens no *array* principal sendo um *mapping* para as ordens de venda e outro *mapping* com as ordens de compra. Os *mapping* servem também como uma lista encadeada, sendo que as ordens estão organizadas cronologicamente. Os valores das ordens estão em *wei*, a unidade que é a menor divisão da moeda nativa do *Ethereum* (ETH).

```
struct Order {
    uint256 index;
    bool isSale;
    address userAddress;
    uint256 createdAt;
    string asset;
    uint256 value;
    uint256 numberOfShares;
    bool acceptsFragmenting;
    bool isActive;
    bool isPassive;
}
```

Figura 9 - Struct que representa uma ordem

Na Figura 9, o *struct* que representa uma ordem é apresentado. Na estrutura de dados, os seguintes dados são registrados:

- index (uint256): inteiro positivo com tamanho de até 256 bits que representa o índice da ordem;
- isSale (bool): booleano que representa se a ordem é uma ordem de venda ou não (ou seja, é uma ordem de compra);
- userAddress (address): endereço de carteira que representa a carteira do usuário dono da ordem;
- createdAt (uint256): inteiro positivo com tamanho de até 256 bits que representa o momento em que a ordem foi criada;
- asset (string): cadeira de caracteres que representa o nome do ativo;
- value (uint256): inteiro positivo com tamanho de até 256 bits que representa o valor em wei do ativo negociado na ordem;
- numberOfShares (uint256): inteiro positivo com tamanho de até 256 bits que representa a quantidade de cotas que vão ser negociadas na ordem;
- acceptsFragmenting (bool): booleano que representa se usuário aceita fragmentar a ordem;
- isActive (bool): booleano que representa se a ordem está ativa ou não;
- isPassive (bool): booleano que representa se a ordem é uma ordem ativa ou não (ou seja, é uma ordem passiva).

Quando o usuário vai criar uma ordem, ele pode escolher se vai criar a ordem com possibilidade de fragmentação. A fragmentação ocorre quando existe ordens compatíveis no ativo e no valor, mas não são compatíveis na quantidade de cotas. Neste caso, é criada uma transação com o menor valor de cotas entre as ordens e criada uma ordem com o valor restante de cotas da ordem com maior quantidade de cotas.

No momento de criação das ordens, o usuário também pode escolher se vai criar uma ordem ativa ou passiva. A ordem ativa no momento de sua criação verifica se existe uma ordem compatível com seus valores podendo ou não resultar em uma transação, enquanto a ordem passiva não realiza a verificação.

```
if (isPassive) {
    sendMoney(owner, userAddress, (value * 1) / 100);
} else {
    sendMoney(owner, userAddress, (value * 2) / 100);
}
```

Figura 10 - Código da transferência de valores para a taxa da criação de ordens

A aplicação realiza a cobrança do proprietário de forma diferente dependendo se a ordem é passiva ou ativa. Como existe a verificação de compatibilidade de ordens para as ordens ativas, a aplicação cobra uma taxa de dois por cento do valor da ordem para as ordens ativas e um por cento para as ordens passivas.

```
struct Transaction {
    uint256 index;
    address seller;
    address buyer;
    string asset;
    uint256 createdAt;
    uint256 value;
    uint256 numberOfShares;
    uint256 saleOrderIndex;
    uint256 purchasedOrderIndex;
}
```

Figura 11 - Struct que representa uma transação

Na Figura 11, o *struct* que representa uma transação é apresentado. Quando uma ordem de compra e uma ordem de venda se tornam compatíveis, as ordens passam se tornarem desativadas e uma transação com o valor das ordens é criado no *array* que representa as transações. Os valores monetários presentes nas transações também estão em wei. Na estrutura de dados que representa a transação, os seguintes dados são registrados:

- index (uint256): inteiro positivo com tamanho de até 256 bits que representa o índice da transação;
- seller (address): endereço de carteira que representa a carteira do usuário dono da ordem de venda;
- buyer (address): endereço de carteira que representa a carteira do usuário dono da ordem de compra;
- asset (string): cadeira de caracteres que representa o ativo negociado;

- createdAt (uint256): inteiro positivo com tamanho de até 256 bits que representa o momento em que a transação foi criada;
- value (uint256): inteiro positivo com tamanho de até 256 bits que representa o valor em wei do ativo negociado;
- numberOfShares (uint256): inteiro positivo com tamanho de até 256 bits que representa o número de cotas do ativo negociado na transação;
- salesOrderIndex (uint256): inteiro positivo com tamanho de até 256 bits que representa o índice da ordem de venda no array de ordens;
- purchaseOrderIndex (uint256): inteiro positivo com tamanho de até 256 bits que representa o índice da ordem de compra no array de ordens.

```
sendMoney(
    purchasedOrder.userAddress,
    saleOrder.userAddress,
    saleOrder.value
);
```

Figura 12 - Código da transferência de valores no momento da criação da transação

Na Figura 12, o código que realiza a transferência de valores no momento da criação da transação é apresentado. O valor da ordem é transferido do comprador da ordem para o vendedor da ordem. A transferência de valores ocorre em três momentos: na criação da ordem, na criação da transação e no saque por parte do proprietário do contrato das taxas cobradas pelo contrato.

```
mapping(address => uint256) private balances;
address[] private addressFromBalances;
uint256 private numberOfAddress;
```

Figura 13 - Estruturas auxiliares da transferência de valores

Na Figura 13, as estruturas auxiliares para a transferência de valores são apresentadas. O *mapping balances* armazena o saldo dos usuários dentro do contrato. O *array addressFromBalances* e o inteiro positivo *numberOfAddress* são estruturas secundárias utilizadas para controlar e iterar os endereços com saldo no contrato.

```
event Deposited(address from, uint256 amount);

function depositMoney() public payable returns (uint256) {
    balances[msg.sender] += msg.value;

    if (!existsAddress(msg.sender)) {
        numberOfAddress += 1;
        addressFromBalances.push(msg.sender);
    }

    emit Deposited(msg.sender, msg.value);

    return balances[msg.sender];
}
```

Figura 14 - Código para o depósito de valores no contrato

Na Figura 14, a função de depósito está sendo exibida. Quando o usuário interage com o contrato, dentro de *msg.sender* estará o endereço da carteira do usuário e dentro de *msg.value* estará o valor depositado pelo usuário. Na condicional *if* da função, é verificado se o usuário já existe no contrato. Se não existir, o contador de endereços é incrementado e o endereço do usuário é adicionado ao *array* de endereços para facilitar a iteração dos endereços. Por fim, em evento de depósito é emitido e a função retorna o saldo do usuário.

```
function sendMoney(
    address receiver,
    address sender,
    uint256 amount
) private returns (bool) {
    require(balances[sender] >= amount, "Balance not sufficient");

    balances[sender] -= amount;
    balances[receiver] += amount;

    payable(receiver).transfer(amount);
}
```

Figura 15 - Código de transferência de valores

Na Figura 15, a função de transferência de valores é exibida. Essa é uma função privada, ou seja, ela só pode ser chamada pelo contrato. Primeiro, o saldo do usuário é verificado, se o saldo for suficiente, a transferência de valores ocorre, caso contrário, um erro é exibido com a mensagem de *"Balance not sufficient"*. O valor da transferência é removido do saldo do usuário remetente e enviado para o saldo do usuário destinatário. Por fim, a transferência ocorre.

```
function withdraw(uint256 amount) public onlyOwner {
    require(balances[msg.sender] >= amount, "Balance not sufficient");

    balances[msg.sender] -= amount;

    payable(msg.sender).transfer(amount);
}
```

Figura 16 - Código de saque de valores

Na Figura 16, a função de saque de valores é exibida. Essa função é uma função pública, mas como existe o modificado *onlyOwner,* apenas o proprietário do contrato pode utilizá-la. Primeiramente, a função verifica o saldo do proprietário do contrato, se tiver saldo, o saque vai ocorrer, caso contrário um erro com a mensagem *"Balance not sufficient"* é exibido. Esses valores são o somatório de todas as taxas cobradas pela utilização do contrato. Antes do saque, o valor do saque é removido do saldo do proprietário do contrato na estrutura de dados do contrato que armazena os saldos e, por fim, o saque é realizado.

```
modifier onlyOwner() {
    require(msg.sender == owner, "This is not the owner");
    _; // it means run the code.
}
```

Figura 17 - Código do modificador onlyOwner

Na Figura 17, o modificado que verifica se o usuário é o proprietário do contrato é exibido. Caso o usuário não for o proprietário do contrato, um erro com a mensagem *"This is not the owner"* é exibido.

6.2 Implementação

```
neves1-ubuntu20@neves:~/projects/lucas54neves/stock-exchange-using-solidity$ yarn
yarn install v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
warning "ethereum-waffle > @ethereum-waffle/compiler > typechain > ts-essentials@6.0.7"
[4/4] Building fresh packages...
[-/22] ⠂ waiting...
[9/22] ⠂ sqlite3
```

Figura 18 - Comando para instalação das dependências do projeto

Na Figura 18, a execução do comando *yarn* é apresentada. Este comando instala as dependências dos projetos, instalando todos os pacotes em JavaScript utilizados no projeto. O comando deve ser executado na raiz do projeto.

```
neves1-ubuntu20@neves:~/projects/lucas54neves/stock-exchange-using-solidity$ yarn add -D hardhat
yarn add v1.22.19
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
warning "ethereum-waffle > @ethereum-waffle/compiler > typechain > ts-essentials@6.0.7" has unmet
[4/4] Building fresh packages...
[-/5] ⠂ waiting...
[2/5] ⠂ sqlite3
```

Figura 19 - Comando para instalação do framework HardHat

Na Figura 19, a execução do comando para instalação do *framework HardHat* é apresentada. O *HardHat* já será instalado no comando da Figura 18, mas, para novos projetos, este é o comando necessário para instalar o *framework*.

```
neves1-ubuntu20@neves:~/projects/lucas54neves/new-project-solidity$ npx hardhat
888    888                      888 888               888
888    888                      888 888               888
888    888                      888 888               888
8888888888  8888b.  888d888 .d88888 88888b.   8888b.  888888
888    888     "88b 888P"  d88" 888 888 "88b     "88b 888
888    888 .d888888 888    888  888 888  888 .d888888 888
888    888 888  888 888    Y88b 888 888  888 888  888 Y88b.
888    888 "Y888888 888     "Y88888 888  888 "Y888888  "Y888

Welcome to Hardhat v2.14.0

✓ What do you want to do? · Create a JavaScript project
✓ Hardhat project root: · /home/neves1-ubuntu20/projects/lucas54neves/new-project-solidity
✓ Do you want to add a .gitignore? (Y/n) · y
✓ Do you want to install this sample project's dependencies with npm (hardhat @nomicfoundat
icfoundation/hardhat-chai-matchers @nomiclabs/hardhat-ethers @nomiclabs/hardhat-etherscan c
hat typechain @typechain/ethers-v5 @ethersproject/abi @ethersproject/providers)? (Y/n) · y
```

Figura 20 - Comando para criar um projeto com o framework HardHat

Na Figura 20, a execução do comando que criar um projeto para desenvolver em *Solidity* com as configurações básicas do *framework HardHat* é apresentada. Neste comando, é possível escolher se a linguagem do projeto será em *TypeScript* ou *JavaScript*, escolher a raiz do projeto, escolher se um arquivo *gitignore* padrão será criado e escolher se algumas dependências comuns em projetos com *HardHat* serão instaladas.

```
neves1-ubuntu20@neves:~/projects/lucas54neves/stock-exchange-using-solidity$ yarn hardhat compile
yarn run v1.22.19
$ /home/neves1-ubuntu20/projects/lucas54neves/stock-exchange-using-solidity/node_modules/.bin/hardh
Compiled 1 Solidity file successfully
Done in 1.53s.
```

Figura 21 - Comando para compilar o projeto

Na Figura 21, a execução do comando para compilar o projeto é apresentada. O comando "hardhat compile" é usado durante o desenvolvimento de contratos inteligentes com o framework *Hardhat*. Ele compila os contratos inteligentes presentes no diretório "contracts/" e gera arquivos *JavaScript* que representam as classes dos contratos compilados. Esses arquivos são armazenados no diretório *artifacts/*. Além disso, o comando executa os plugins de compilação configurados, como o *hardhat-deploy* que pode gerar arquivos de configuração para implantação de contratos em diferentes redes. Em resumo, o comando *hardhat compile* é fundamental para transformar o código-fonte dos contratos em *bytecode* e prepará-los para serem implantados em uma rede blockchain (HardHat, 2023).

```
neves1-ubuntu20@neves:~/projects/lucas54neves/stock-exchange-using-solidity$ yarn hardhat test
yarn run v1.22.19
$ /home/neves1-ubuntu20/projects/lucas54neves/stock-exchange-using-solidity/node_modules/.bin/h


  Case 1

Before test
Asset: ABC123
Qtd     Compra  Venda   Qtd
576     894828  258292  19
14      745675  356851  173
207     656983  486400  669
405     587962  572877  299
```

Figura 22 - Comando para executar os testes do projeto

Na Figura 22, a execução do comando para executar os testes do projeto é apresentada. Este comando vai executar todos os testes presentes no diretório *src/test/* do projeto.

A fim de sempre manter a aplicação funcionando, foi desenvolvido um pipeline de CI/CD no repositório da aplicação no GitHub. Primeiro, a *branch* principal *main* foi bloqueada para aceitar apenas modificações vinda de uma *pull request*. Por fim, foi desenvolvido um script que testar o código de todas *pull requests*, executando todos os testes automatizados.

Figura 13 - Bloqueio da branch main

Na Figura 13, o bloqueio da *branch main* é apresentado. O bloqueio permitirá que apenas *commits* originados de uma *pull request* sejam aceitos. Para criar bloqueios de *branch* ou outro tipo de regras para uma *branch*, deve-se entrar no repositório do GitHub desejado, entrar na aba *Settings*, entrar na aba *Branches* e selecionar a regra desejada.

```yaml
name: Test Exchange

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:

    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [12.x, 14.x, 16.x]

    steps:
    - uses: actions/checkout@v3
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v3
      with:
        node-version: ${{ matrix.node-version }}
        cache: 'npm'
    - name: Install dependencies
      run: npm install
    - name: Run tests
      run: npm run test
```

Figura 14 - Script que executa os testes automatizados

Na Figura 14, o *script* que executa os testes automatizados é apresentado. O script executa em três versões do *NodeJS* (12, 14 e 16) os testes automatizados toda vez que uma *pull request* for criada.

6.3 Testes

Testes automatizados foram desenvolvidos para garantir o funcionamento do projeto. Os testes são divididos em duas partes: nove testes unitários e trinta e sete testes de casos. Os testes unitários têm como função apenas testar os métodos do contrato do projeto. Os casos de testes, por sua vez, têm como objetivo testar diferentes cenários para garantir que o projeto funcione de ponta-a-ponta. Uma base de dados pré-teste é gerada aleatoriamente para a realização dos testes. A geração aleatória é feita pelo framework Chance.

```
await this.exchange.connect(this.accounts[i + 1]).depositMoney({
  value: ethers.utils.formatUnits(
    String(
      order.isPassive
        ? Math.ceil(order.value * 1.01)
        : Math.ceil(order.value * 1.02)
    ),
    'wei'
  ),
});
```

Figura 23 - Código de conexão com a carteira do usuário

Na Figura 23, um exemplo de código de como se conectar com a carteira do usuário durante os testes. Para que os testes ocorram da maneira correta, deve-se assinar as chamadas das funções com o endereço da carteira do usuário que vai executar a chamada. Chama-se o método *connect* do objeto do contrato, passando como parâmetro o endereço da carteira do usuário, e depois chama-se a função desejada passando seus parâmetros.

```
Before test
Asset: ABC123
Qtd     Compra  Venda   Qtd
292     992065  181030  395
884     865862  278110  828
359     601297  378551  778
285     399389  399078  949
753     48463   962320  719
874     30304
    ✓ should be enable to create a purchased order (204ms)
After test
Asset: ABC123
Qtd     Compra  Venda   Qtd
292     992065  181030  395
884     865862  278110  828
359     601297  378551  778
285     399389  399078  949
753     48463   962320  719
874     30304
100     78
```

Figura 24 - Teste de criação de ordem de compra do caso 8

Na Figura 24, o teste de criação de ordens de compra é apresentado. A listagem das ordens é exibida em seu estado anterior à criação da ordem e no seu estado posterior à criação da ordem. O teste exibe a criação correta da ordem de compra com valor 78 e quantidade de cotas de 100.

```
Before test
Asset: ABC123
Qtd     Compra  Venda   Qtd
292     992065  181030  395
884     865862  278110  828
359     601297  378551  778
285     399389  399078  949
753     48463   962320  719
874     30304
100     78
    ✓ should be enable to create a sale order (310ms)
After test
Asset: ABC123
Qtd     Compra  Venda   Qtd
292     992065  78      150
884     865862  181030  395
359     601297  278110  828
285     399389  378551  778
753     48463   399078  949
874     30304   962320  719
100     78
```

Figura 25 - Teste de criação de ordem de venda do caso 8

Na Figura 25, o teste de criação de ordens de venda é exibido. A listagem das ordens é exibida em seu estado anterior à criação da ordem e no seu estado posterior à criação da ordem. O teste exibe a criação correta da ordem de venda com o valor 78 e quantidade de cotas de 150.

```
    ✔ should not create transaction (272ms)
After test
Asset: ABC123
Qtd     Compra  Venda   Qtd
56      921689  78      150
196     657402  351227  109
721     645839  372827  874
36      511818  792308  436
233     267666  867881  684
284     145168  899900  227
714     126672  923723  634
100     78
  Case 9
Before test
Asset: ABC123
Qtd     Compra  Venda   Qtd
646     960588  122899  140
896     876715  672885  978
607     767259  775120  778
675     451527  798178  245
909     379190  824964  538
840     311213  926616  700
357     301925  973672  278
250     168989
```

Figura 26 - Teste de criação da transação do caso 8

Na Figura 26, o teste de criação de transações é apresentado. A listagem das ordens é exibida em seu estado anterior à criação da transação e no seu estado posterior à criação da transação.

```
it('should send money', async () => {
  const accountId = chance.integer({ min: 1, max: this.accounts.length - 1 });
  const value = chance.integer({ min: 1000, max: 1_000_000 });

  const balanceBefore = await this.exchange.getSmartContractBalance();

  await this.exchange.connect(this.accounts[accountId]).depositMoney({
    value: ethers.utils.formatUnits(String(value), 'wei'),
  });

  const balanceAfter = await this.exchange.getSmartContractBalance();

  expect(convertToNumber(balanceAfter)).to.equal(
    convertToNumber(balanceBefore) + value
  );
});
```

Figura 27 - Teste unitário para a funcionalidade de depositar valores monetários

Na Figura 27, o teste unitário para o depósito de valores monetários é apresentado. Os testes unitários da aplicação têm como função testar cada uma das funcionalidades do contrato. O teste da Figura 27 consiste em primeiro verificar o saldo anterior do destinatário dos valores. Após a verificação, o valor monetário é enviado. Depois do envio, o saldo atual do destinatário é verificado. Por vim, o teste espera que o saldo depois da movimentação seja o saldo anterior à movimentação mais o valor movimentado. Caso os valores sejam compatíveis, o teste é aprovado.

```
it('should compare assets', async () => {
  const sameAssets = await this.exchange.compareStrings('PETRO', 'PETRO');

  expect(sameAssets).to.equal(true);

  const differentAssets = await this.exchange.compareStrings('PETRO', 'VALE');

  expect(differentAssets).to.equal(false);
});
```

Figura 28 - Teste unitário para a funcionalidade de verificar a igualdade entre strings

Na Figura 28, o teste unitário para comparar a igualdade entre duas *strings* é apresentado. O teste consiste na verificação dos bytes das strings, uma vez que a comparação direta entre duas strings não é possível. Caso a função retorne verdadeiro quando as strings sejam iguais e falso quando as strings sejam diferentes, o teste é aprovado, caso contrário é recusado.

```
it('should create a order', async () => {
  const orderData = {
    isSale: false,
    userAddress: this.accounts[1].address,
    asset: 'VALE',
    value: 1235,
    numberOfShares: 4,
    acceptsFragmenting: true,
    isPassive: true,
  };

  const order = await this.exchange.createOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    orderData.value,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  expect(orderData.isSale).to.equal(order.isSale);
  expect(orderData.userAddress).to.equal(order.userAddress);
  expect(orderData.asset).to.equal(order.asset);
  expect(orderData.value).to.equal(order.value);
  expect(orderData.numberOfShares).to.equal(order.numberOfShares);
  expect(orderData.acceptsFragmenting).to.equal(order.acceptsFragmenting);
  expect(orderData.isPassive).to.equal(order.isPassive);
});
```

Figura 29 - Teste unitário para a funcionalidade de criação de ordens

Na Figura 29, o teste da funcionalidade de criação de ordens é apresentado. O teste consiste em criar uma ordem e verificar se os atributos da ordem criada têm os mesmos parâmetros passados na função que cria a ordem. Caso os atributos sejam iguais aos parâmetros, o teste é aprovado, caso contrário é recusado.

```
it('should create a transaction', async () => {
  const transactionData = {
    seller: this.accounts[2].address,
    buyer: this.accounts[1].address,
    asset: 'ITAUSA',
    value: 22,
    numberOfShares: 1,
    saleOrderIndex: 4,
    purchasedOrderIndex: 3,
  };

  const transaction = await this.exchange.createTransaction(
    transactionData.seller,
    transactionData.buyer,
    transactionData.asset,
    transactionData.value,
    transactionData.numberOfShares,
    transactionData.saleOrderIndex,
    transactionData.purchasedOrderIndex
  );

  expect(transactionData.seller).to.equal(transaction.seller);
  expect(transactionData.buyer).to.equal(transaction.buyer);
  expect(transactionData.asset).to.equal(transaction.asset);
  expect(transactionData.value).to.equal(transaction.value);
  expect(transactionData.numberOfShares).to.equal(transaction.numberOfShares);
  expect(transactionData.saleOrderIndex).to.equal(transaction.saleOrderIndex);
  expect(transactionData.purchasedOrderIndex).to.equal(
    transaction.purchasedOrderIndex
  );
});
```

Figura 30 - Teste unitário para a funcionalidade de criação de transações

Na Figura 30, o teste unitário para a funcionalidade de criação de transações é apresentado. O teste consiste em criar uma transação e verificar se os atributos da transação criada são os mesmos dos parâmetros passados na função de criação de transação. Caso os atributos sejam iguais aos parâmetros, o teste é aprovado, caso contrário é recusado.

```
it('should add orders', async () => {
  const orderData = {
    isSale: false,
    userAddress: this.accounts[1].address,
    asset: 'VALE',
    value: 1235,
    numberOfShares: 4,
    acceptsFragmenting: true,
    isPassive: true,
  };

  const order = await this.exchange.createOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    orderData.value,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  let orders;

  for (let i = 0; i <= 10; i++) {
    orders = await this.exchange.returnOrders();

    expect(orders.length).to.equal(i);

    await this.exchange.addOrder(order);
  }

  expect(orders.length).to.equal(10);
});
```

Figura 31 - Teste unitário para a funcionalidade de adição de ordens

Na Figura 31, o teste unitário para a funcionalidade de adição de ordens é apresentado. O teste consiste em criar várias ordens e verificar o tamanho do array de ordens após a adição. Caso o tamanho do array de ordens seja sempre igual à quantidade de ordens adicionadas o teste é aprovado, caso contrário é recusado.

```
it('should add transaction', async () => {
  const transactionData = {
    seller: this.accounts[2].address,
    buyer: this.accounts[1].address,
    asset: 'ITAUSA',
    value: 22,
    numberOfShares: 1,
    saleOrderIndex: 4,
    purchaseOrderIndex: 3,
  };

  const transaction = await this.exchange.createTransaction(
    transactionData.seller,
    transactionData.buyer,
    transactionData.asset,
    transactionData.value,
    transactionData.numberOfShares,
    transactionData.saleOrderIndex,
    transactionData.purchaseOrderIndex
  );

  let transactions;

  for (let i = 0; i <= 10; i++) {
    transactions = await this.exchange.returnTransactions();

    expect(transactions.length).to.equal(i);

    await this.exchange.addTransaction(transaction);
  }

  expect(transactions.length).to.equal(10);
});
```

Figura 32 - Teste unitário para a funcionalidade de adição de transação

Na Figura 32, o teste unitário para a funcionalidade de adição de transação é apresentado. O teste consiste em criar dez transações e verificar a cada adição de transação se o tamanho do array de transações está compatível com a quantidade de transações adicionadas.

```
it('should check transaction conflict', async () => {
  // sellerAcceptsToFragment,
  // buyerAcceptsToFragment,
  // sellerHasTheMostQuantity,
  // numberOfSharesIsDifferent
  expect(
    await this.exchange.checkTransactionConflict(false, true, true, true)
  ).to.equal(true);
  expect(
    await this.exchange.checkTransactionConflict(false, false, true, true)
  ).to.equal(true);
  expect(
    await this.exchange.checkTransactionConflict(false, false, false, true)
  ).to.equal(true);
  expect(
    await this.exchange.checkTransactionConflict(true, false, false, true)
  ).to.equal(true);
  expect(
    await this.exchange.checkTransactionConflict(true, true, true, true)
  ).to.equal(false);
  expect(
    await this.exchange.checkTransactionConflict(true, true, false, true)
  ).to.equal(false);
  expect(
    await this.exchange.checkTransactionConflict(false, true, false, true)
  ).to.equal(false);
  expect(
    await this.exchange.checkTransactionConflict(true, false, true, true)
  ).to.equal(false);
});
```

Figura 33 - Teste unitário para a funcionalidade de verificar se existe conflito na criação da transação

Na Figura 33, o teste unitário para a funcionalidade de verificar se existe conflito na criação da transação é apresentado. O teste consiste em verificar se o vendedor aceita fragmentar a transação, se o comprador aceita fragmentar a transação, se o vendedor possui maior quantidade de cotas do ativo e por fim se a quantidade de cotas do ativo é diferente entre as duas ordens. Se o vendedor não aceitar fragmentar a transação e se o vendedor possuir maior quantidade de cotas do ativo, a função vai retornar verdadeiro. Se o comprador não aceitar fragmentar e se o comprador possuir maior quantidade de cotas do ativo, a função vai retornar verdadeiro. Caso contrário, a função vai retornar falso.

```javascript
it('should return sale orders', async () => {
  const orderData = {
    isSale: true,
    userAddress: this.accounts[1].address,
    asset: 'VALE',
    numberOfShares: 4,
    acceptsFragmenting: true,
    isPassive: true,
  };

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    56,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    58,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    52,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    47,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    69,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    53,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    51,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  const orders = await this.exchange.returnSaleOrders(orderData.asset);

  expect(orders.length).to.equal(7);
  expect(orders[0].value).to.equal(47);
  expect(orders[1].value).to.equal(51);
  expect(orders[2].value).to.equal(52);
  expect(orders[3].value).to.equal(53);
  expect(orders[4].value).to.equal(56);
  expect(orders[5].value).to.equal(58);
  expect(orders[6].value).to.equal(69);
});
```

Figura 34 - Teste unitário para a ordenação das ordens de venda

Na Figura 34, o teste unitário que testa a ordenação das ordens de venda após as adições sucessivas de ordem é apresentado. O teste consiste em adicionar sete ordens e verificar se as ordens se mantiveram ordenadas de acordo com o seu valor. Se a ordenação estiver correta, o teste será aprovado, caso contrário será reprovado.

```
it('should return purchased orders', async () => {
  const orderData = {
    isSale: false,
    userAddress: this.accounts[1].address,
    asset: 'VALE',
    numberOfShares: 4,
    acceptsFragmenting: true,
    isPassive: true,
  };

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    56,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    58,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    52,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    47,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    69,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    53,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    orderData.isSale,
    orderData.userAddress,
    orderData.asset,
    51,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  const orders = await this.exchange.returnPurchasedOrders(orderData.asset);

  expect(orders.length).to.equal(7);
  expect(orders[0].value).to.equal(69);
  expect(orders[1].value).to.equal(58);
  expect(orders[2].value).to.equal(56);
  expect(orders[3].value).to.equal(53);
  expect(orders[4].value).to.equal(52);
  expect(orders[5].value).to.equal(51);
  expect(orders[6].value).to.equal(47);
});
```

Figura 35 - Teste unitário para a ordenação das ordens de compra

Na Figura 35, o teste unitário que testa a ordenação das ordens de compra após as adições sucessivas de ordem é apresentado. O teste consiste em adicionar sete ordens e verificar se as ordens se mantiveram ordenadas de acordo com o seu valor. Se a ordenação estiver correta, o teste será aprovado, caso contrário será reprovado.

```
it('should realize a operation', async () => {
  const orderData = {
    userAddress: this.accounts[1].address,
    asset: 'VALE',
    numberOfShares: 4,
    acceptsFragmenting: true,
    isPassive: true,
  };

  await this.exchange.realizeOperationOfCreationOfOrder(
    true,
    orderData.userAddress,
    orderData.asset,
    56,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  let saleOrders = await this.exchange.returnSaleOrders(orderData.asset);

  expect(saleOrders.length).to.equal(1);

  const balanceBefore = await this.exchange.getSmartContractBalance();

  await this.exchange.connect(this.accounts[1]).depositMoney({
    value: ethers.utils.formatUnits(String(56), 'wei'),
  });

  const balanceAfter = await this.exchange.getSmartContractBalance();
  expect(convertToNumber(balanceAfter)).to.equal(
    convertToNumber(balanceBefore) + 56
  );

  await this.exchange.realizeOperationOfCreationOfOrder(
    false,
    orderData.userAddress,
    orderData.asset,
    56,
    orderData.numberOfShares,
    orderData.acceptsFragmenting,
    orderData.isPassive
  );

  let purchasedOrders = await this.exchange.returnPurchasedOrders(
    orderData.asset
  );

  expect(purchasedOrders.length).to.equal(1);

  await this.exchange.realizeOperationOfCreationOfTransaction(
    orderData.asset,
    saleOrders[0]
  );

  const transactions = await this.exchange.returnTransactions();

  expect(transactions.length).to.equal(1);

  saleOrders = await this.exchange.returnSaleOrders(orderData.asset);

  expect(saleOrders.length).to.equal(0);

  purchasedOrders = await this.exchange.returnPurchasedOrders(
    orderData.asset
  );

  expect(purchasedOrders.length).to.equal(0);
  });
});
```

Figura 36 - Teste para a funcionalidade de realizar a operação de criação transação

Na Figura 36, o teste para a funcionalidade de realizar a operação de criação de transação é apresentado. A realizar a operação significa criar as ordens de compra e venda e, posteriormente, criar a transação. Caso todo processo ocorra como esperado, ou seja, que as ordens e a transação sejam criadas, o teste será aprovado, caso contrário será reprovado.

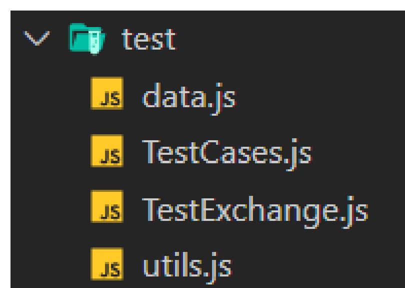

Figura 37 - Diretório dos testes

Na Figura 37, o diretório dos testes é apresentado. No arquivo *data.js*, todos os dados de testes para os casos de testes estão presentes. No arquivo *TesteCases.js*, os testes para casos de testes estão presentes. No arquivo *TestExchange.js*, os testes unitários para o contrato estão presentes. No arquivo *utils.js*, funções usadas com frequência nos testes estão presentes.

O script que testa os casos de teste se divide em três testes menores. No primeiro ocorre o teste que verifica se as ordens de compra foram criadas corretamente, ou seja, o teste apenas será aprovado se as ordens forem criadas corretamente, caso contrário o teste será recusado. No segundo ocorre o teste que verifica se as ordens de venda foram criadas corretamente, ou

seja, o teste apenas será aprovado se as ordens forem criadas corretamente, caso contrário o teste será recusado. Por fim, no terceiro, ocorre o teste relacionado a criação da transação. Se no caso de teste foi mapeado que a transação deva ocorrer, o teste será aprovado apenas se a transação ocorrer, caso contrário o teste será recusado. Se no caso de teste foi mapeado que a transação não deva ocorrer, o teste será aprovado apenas se transações não forem criadas, caso contrário o teste será recusado.

Tabela 2 - Tabela com os dados do primeiro caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**
Ativo	ABC123	ABC123
Quantidade de ações	100	100
Valor por cada ação	78	78
Aceita fragmentar	Sim	Sim
Ordem passiva	Sim	Sim

Na Tabela 2, os dados do primeiro caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O vendedor vende cem ações do ativo por 78 wei cada ação. O comprador e o vendedor aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com as cem ações por 78 wei.

Tabela 3 - Tabela com os dados do segundo caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**
Ativo	ABC123	ABC123
Quantidade de ações	100	100
Valor por cada ação	78	78
Aceita fragmentar	Não	Sim
Ordem passiva	Sim	Sim

Na Tabela 3, os dados do segundo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O vendedor vende cem ações do ativo por 78 wei cada ação. O comprador não aceita fragmentar e o vendedor aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com as cem ações por 78 wei.

Tabela 4 - Tabela com os dados do terceiro caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**
Ativo	ABC123	ABC123
Quantidade de ações	100	100
Valor por cada ação	78	78
Aceita fragmentar	Sim	Não
Ordem passiva	Sim	Sim

Na Tabela 4, os dados do terceiro caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O vendedor vende cem ações do ativo por 78 wei cada ação. O comprador aceita fragmentar e o vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com as cem ações por 78 wei.

Tabela 5 - Tabela com os dados do quarto caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**
Ativo	ABC123	ABC123
Quantidade de ações	100	100
Valor por cada ação	78	78
Aceita fragmentar	Não	Não
Ordem passiva	Sim	Sim

Na Tabela 5, os dados do quarto caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O vendedor vende cem ações do ativo por 78 wei cada ação. O comprador e o vendedor não aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com as cem ações por 78 wei.

Tabela 6 - Tabela com os dados do quinto caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**
Ativo	ABC123	ABC123
Quantidade de ações	100	150
Valor por cada ação	78	78
Aceita fragmentar	Sim	Sim
Ordem passiva	Sim	Sim

Na Tabela 6, os dados do quinto caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O vendedor vende cento e cinquenta ações do ativo por 78 wei cada ação. O comprador e o vendedor aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com as cem ações por 78 wei e uma nova ordem de venda com cinquenta ações por 78 wei cada ação.

Tabela 7 - Tabela com os dados do sexto caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**
Ativo	ABC123	ABC123
Quantidade de ações	100	150
Valor por cada ação	78	78
Aceita fragmentar	Não	Sim
Ordem passiva	Sim	Sim

Na Tabela 7, os dados do sexto caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O vendedor vende cento e cinquenta ações do ativo por 78 wei cada ação. O comprador não aceita fragmentar e o vendedor aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com as cem ações por 78 wei e uma nova ordem de venda com cinquenta ações por 78 wei cada ação.

Tabela 8 - Tabela com os dados do sétimo caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**
Ativo	ABC123	ABC123
Quantidade de ações	100	150
Valor por cada ação	78	78
Aceita fragmentar	Sim	Não
Ordem passiva	Sim	Sim

Na Tabela 8, os dados do sétimo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O vendedor vende cento e cinquenta ações do ativo por 78 wei cada ação. O comprador aceita fragmentar e o vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, não se deve criar uma transação e as ordens de venda e de compra ainda devem estar abertas.

Tabela 9 - Tabela com os dados do oitavo caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**
Ativo	ABC123	ABC123
Quantidade de ações	100	150
Valor por cada ação	78	78
Aceita fragmentar	Não	Não
Ordem passiva	Sim	Sim

Na Tabela 9, os dados do oitavo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O vendedor vende cento e cinquenta ações do ativo por 78 wei cada ação. O comprador e o vendedor não aceitam fragmentar a transação. Para que o teste seja aprovado, não se deve criar uma transação e as ordens de venda e de compra ainda devem estar abertas.

Tabela 10 - Tabela com os dados do nono caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**
Ativo	ABC123	ABC123
Quantidade de ações	150	100
Valor por cada ação	78	78
Aceita fragmentar	Sim	Sim
Ordem passiva	Sim	Sim

Na Tabela 10, os dados do nono caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cento e cinquenta ações do ativo por 78 wei cada ação. O vendedor vende cem ações do ativo por 78 wei cada ação. O comprador e o vendedor aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com as cem ações por 78 wei e uma nova ordem de compra com cinquenta ações por 78 wei cada ação.

Tabela 11 - Tabela com os dados do décimo caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1
Ativo	ABC123	ABC123
Quantidade de ações	150	100
Valor por cada ação	78	78
Aceita fragmentar	Não	Sim
Ordem passiva	Sim	Sim

Na Tabela 11, os dados do décimo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O vendedor vende cento e cinquenta ações do ativo por 78 wei cada ação. O comprador não aceita fragmentar e o vendedor aceita fragmentar a transação. Para que o teste seja aprovado, não se deve criar uma transação e as ordens de venda e de compra ainda devem estar abertas.

Tabela 12 - Tabela com os dados do décimo primeiro caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1
Ativo	ABC123	ABC123
Quantidade de ações	150	100
Valor por cada ação	78	78
Aceita fragmentar	Sim	Não
Ordem passiva	Sim	Sim

Tabela 13 - Tabela com os dados do décimo segundo caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1
Ativo	ABC123	ABC123
Quantidade de ações	150	100
Valor por cada ação	78	78
Aceita fragmentar	Não	Não
Ordem passiva	Sim	Sim

Na Tabela 12, os dados do décimo primeiro caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cento e cinquenta ações do ativo por 78 wei cada ação. O vendedor vende cem ações do ativo por 78 wei cada ação. O comprador aceita fragmentar e o vendedor não aceita fragmentar a transação. Para que o teste seja

aprovado, deve-se criar uma transação com as cem ações por 78 wei e uma nova ordem de compra com cinquenta ações por 78 wei cada ação.

Na Tabela 13, os dados do décimo segundo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e um vendedor. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O vendedor vende cento e cinquenta ações do ativo por 78 wei cada ação. O comprador e o vendedor não aceitam fragmentar a transação. Para que o teste seja aprovado, não se deve criar uma transação e as ordens de venda e de compra ainda devem estar abertas.

Tabela 14 - Tabela com os dados do décimo terceiro caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1	Ordem de venda 2
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	60	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Sim	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 14, os dados do décimo terceiro caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e os vendedores aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações e outra transação com quarenta ações.

Tabela 15 - Tabela com os dados do décimo quarto caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1	Ordem de venda 2
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	60	40
Valor por cada ação	78	78	78
Aceita fragmentar	Não	Sim	Sim
Ordem passiva	Sim	Sim	Sim

Tabela 16 - Tabela com os dados do décimo quinto caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1	Ordem de venda 2
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	60	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Não	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 15, os dados do décimo quarto caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei

cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador não aceita fragmentar e os vendedores aceitam fragmentar a transação. Para que o teste seja aprovado, não se deve criar uma transação e as ordens de venda e de compra ainda devem estar abertas.

Na Tabela 16, os dados do décimo quinto caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e o segundo vendedor aceitam fragmentar e o primeiro vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações e outra transação com quarenta ações.

Tabela 17 - Tabela com os dados do décimo sexto caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	60	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Sim	Não
Ordem passiva	Sim	Sim	Sim

Na Tabela 17, os dados do décimo sexto caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e o primeiro vendedor aceitam fragmentar e o segundo vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações e outra transação com quarenta ações.

Tabela 18 - Tabela com os dados do décimo sétimo caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	60	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Não	Não
Ordem passiva	Sim	Sim	Sim

Na Tabela 18, os dados do décimo sétimo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador aceita fragmentar e os vendedores não aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações e outra transação com quarenta ações.

Tabela 19 - Tabela com os dados do décimo oitavo caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1	Ordem de venda 2
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	130	60	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Sim	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 19, os dados do décimo oitavo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cento e trinta ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e os vendedores aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações e outra transação com quarenta ações e criar uma outra ordem de compra com trinta ações por 78 wei.

Tabela 20 - Tabela com os dados do décimo nono caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1	Ordem de venda 2
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	130	60	40
Valor por cada ação	78	78	78
Aceita fragmentar	Não	Sim	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 20, os dados do décimo nono caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cento e trinta ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador não aceita fragmentar e os vendedores aceitam fragmentar a transação. Para que o teste seja aprovado, não se deve criar uma transação e as ordens ainda devem estar ativas.

Tabela 21 - Tabela com os dados do vigésimo caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1	Ordem de venda 2
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	130	60	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Não	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 21, os dados do vigésimo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cento e trinta ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e o segundo vendedor aceitam fragmentar e o primeiro

vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações e outra transação com quarenta ações e criar uma outra ordem de compra com trinta ações por 78 wei.

Tabela 22 - Tabela com os dados do vigésimo primeiro caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1	Ordem de venda 2
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	130	60	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Sim	Não
Ordem passiva	Sim	Sim	Sim

Na Tabela 22, os dados do vigésimo primeiro caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cento e trinta ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e o primeiro vendedor aceitam fragmentar e o segundo vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações e outra transação com quarenta ações e criar uma outra ordem de compra com trinta ações por 78 wei.

Tabela 23 - Tabela com os dados do vigésimo segundo caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1	Ordem de venda 2
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	130	60	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Não	Não
Ordem passiva	Sim	Sim	Sim

Na Tabela 23, os dados do vigésimo segundo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cento e trinta ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador aceita fragmentar e os vendedores não aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações e outra transação com quarenta ações e criar uma outra ordem de compra com trinta ações por 78 wei.

Tabela 24 - Tabela com os dados do vigésimo terceiro caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1	Ordem de venda 2
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	60	50
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Sim	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 24, os dados do vigésimo terceiro caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende cinquenta ações por 78 wei. O comprador e os vendedores aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações e outra transação com quarenta ações e criar uma outra ordem de venda com dez ações por 78 wei.

Tabela 25 - Tabela com os dados do vigésimo quarto caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	60	50
Valor por cada ação	78	78	78
Aceita fragmentar	Não	Sim	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 25, os dados do vigésimo quarto caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende cinquenta ações por 78 wei. O comprador não aceita fragmentar e os vendedores aceitam fragmentar a transação. Para que o teste seja aprovado, não se deve criar uma transação e as ordens devem ainda estar ativas.

Tabela 26 - Tabela com os dados do vigésimo quinto caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	60	50
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Não	Sim
Ordem passiva	Sim	Sim	Sim

Tabela 27 - Tabela com os dados do vigésimo sexto caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	60	50
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Sim	Não
Ordem passiva	Sim	Sim	Sim

Na Tabela 26, os dados do vigésimo quinto caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende cinquenta ações por 78 wei. O comprador e o segundo vendedor aceitam fragmentar e o

primeiro vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações e outra transação com quarenta ações e criar uma outra ordem de venda com dez ações por 78 wei.

Na Tabela 27, os dados do vigésimo sexto caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende cinquenta ações por 78 wei. O comprador e o primeiro vendedor aceitam fragmentar e o segundo vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações, a ordem de venda com cinquenta ações ainda deve estar ativa e uma ordem de compra com as quarenta ações remanescentes deve ser criada.

Tabela 28 - Tabela com os dados do vigésimo sétimo caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1	Ordem de venda 2
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	60	50
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Não	Não
Ordem passiva	Sim	Sim	Sim

Na Tabela 28, os dados do vigésimo sétimo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende sessenta ações por 78 wei e o segundo vendedor vende cinquenta ações por 78 wei. O comprador aceita fragmentar e os vendedores não aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com sessenta ações, a ordem de venda com cinquenta ações ainda deve estar ativa e uma ordem de compra com as quarenta ações remanescentes deve ser criada.

Tabela 29 - Tabela com os dados do vigésimo oitavo caso de teste

Propriedades	Ordem de compra 1	Ordem de venda 1	Ordem de venda 2
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	100	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Sim	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 29, os dados do vigésimo oitavo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende cem ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e os vendedores aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com cem ações e a segunda ordem de venda ainda deve estar ativa.

Tabela 30 - Tabela com os dados do vigésimo nono caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	100	40
Valor por cada ação	78	78	78
Aceita fragmentar	Não	Sim	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 30, os dados do vigésimo nono caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende cem ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador não aceita fragmentar e os vendedores aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com cem ações e a segunda ordem de venda ainda deve estar ativa.

Tabela 31 - Tabela com os dados do trigésimo caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	100	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Não	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 31, os dados do trigésimo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende cem ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e o segundo vendedor aceitam fragmentar e o primeiro vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com cem ações e a segunda ordem de venda ainda deve estar ativa.

Tabela 32 - Tabela com os dados do trigésimo primeiro caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	100	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Sim	Não
Ordem passiva	Sim	Sim	Sim

Na Tabela 32, os dados do trigésimo primeiro caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende cem ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e o primeiro vendedor aceitam fragmentar e o segundo

vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com cem ações e a segunda ordem de venda ainda deve estar ativa.

Tabela 33 - Tabela com os dados do trigésimo segundo caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	100	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Não	Não
Ordem passiva	Sim	Sim	Sim

Na Tabela 33, os dados do trigésimo segundo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende cem ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador aceita fragmentar e os vendedores não aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com cem ações e a segunda ordem de venda ainda deve estar ativa.

Tabela 34 - Tabela com os dados do trigésimo terceiro caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	140	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Sim	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 34, os dados do trigésimo terceiro caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende cento e quarenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e os vendedores aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com cem ações e uma ordem de venda com as quarenta ações remanescentes da ordem do primeiro comprador, além disso a ordem de venda do segundo vendedor ainda deve estar ativa.

Tabela 35 - Tabela com os dados do trigésimo quarto caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	140	40
Valor por cada ação	78	78	78
Aceita fragmentar	Não	Sim	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 35, os dados do trigésimo quarto caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende cento e quarenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador não aceita fragmentar e os vendedores aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com cem ações e uma ordem de venda com as quarenta ações remanescentes da ordem do primeiro vendedor, além disso a ordem de venda do segundo vendedor ainda deve estar ativa.

Tabela 36 - Tabela com os dados do trigésimo quinto caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	140	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Não	Sim
Ordem passiva	Sim	Sim	Sim

Na Tabela 36, os dados do trigésimo cinco caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende cento e quarenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e o segundo vendedor aceitam fragmentar e primeiro vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com quarenta ações e uma ordem de compra com as sessenta ações remanescentes da ordem do segundo vendedor, além disso a ordem de venda do primeiro vendedor ainda deve estar ativa.

Tabela 37 - Tabela com os dados do trigésimo sexto caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	140	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Sim	Não
Ordem passiva	Sim	Sim	Sim

Na Tabela 37, os dados do trigésimo sexto caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende cento e quarenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador e o primeiro vendedor aceitam fragmentar e segundo vendedor não aceita fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com cem ações e uma ordem de compra com as quarenta ações remanescentes da ordem do primeiro vendedor, além disso a ordem de venda do segundo vendedor ainda deve estar ativa.

Tabela 38 - Tabela com os dados do trigésimo sétimo caso de teste

Propriedades	**Ordem de compra 1**	**Ordem de venda 1**	**Ordem de venda 2**
Ativo	ABC123	ABC123	ABC123
Quantidade de ações	100	140	40
Valor por cada ação	78	78	78
Aceita fragmentar	Sim	Não	Não
Ordem passiva	Sim	Sim	Sim

Na Tabela 38, os dados do trigésimo sétimo caso de teste são apresentados. No teste, a operação é realizada entre um comprador e dois vendedores. O ativo a ser negociado é o ativo *ABC123* e ambas as ordens são passivas. O comprador compra cem ações do ativo por 78 wei cada ação. O primeiro vendedor vende cento e quarenta ações por 78 wei e o segundo vendedor vende quarenta ações por 78 wei. O comprador aceita fragmentar e os vendedores não aceitam fragmentar a transação. Para que o teste seja aprovado, deve-se criar uma transação com quarenta ações e uma ordem de compra com as sessenta ações remanescentes da ordem do primeiro vendedor, além disso a ordem de venda do segundo vendedor ainda deve estar ativa.

7 Análise e Discussões

A linguagem *Solidity* é semelhante às linguagens C++ e JavaScript. A forma como as linguagens são escritas e suas estruturas de dados, de seleção e de repetição são semelhantes entre as linguagens. Desenvolvedores de C++ e JavaScript podem migrar para o *Solidity* sem grandes dificuldades.

No ambiente de desenvolvimento *Solidity*, o *framework HardHat* é necessário porque ele otimiza o tempo do desenvolvedor e facilita os testes automatizados. O *framework* possui carteiras de testes que são utilizadas nos testes automatizados e o *deployment* pode ser feito pelo framework.

A aplicação desenvolvida para o projeto cumpriu com todos os requisitos mapeados, com exceção do requisito de baixa latência para a aplicação. A aplicação não cumpriu o requisito de baixa latência porque a tecnologia *blockchain* é uma tecnologia de alta latência. A alta latência ocorre porque, para uma transação realizada em uma *blockchain* ser considerada realmente válida, deve-se esperar seis confirmações, ou seja, seis aprovações de blocos (Cardoso, 2018). No caso do *Bitcoin* que aprova um bloco a cada dez minutos (*Bitcoin Wiki*, 2019), o tempo de espera deve ser sessenta minutos e, no caso da *Ethereum* que aprova um bloco a cada quinze segundos (*Ethereum Foundation*, 2023), o tempo de espera deve ser de um minuto e meio.

A alta latência da tecnologia também foi um problema para a Bolsa de Valores Australiana (*Investing.com*, 2022). A instituição australiana cancelou um projeto de adotar a *blockchain* em sua infraestrutura devido à alta latência da tecnologia. Em operações do mercado financeiro, a baixa latência é indispensável. Em sistemas de compra e venda de ações, a aprovação das operações deve ocorrer com uma latência extremante baixa. Caso contrário, o sistema causará falhas ou lentidão.

Os testes automáticos desenvolvidos para a aplicação foram todos aprovados. Os testes testaram as funções do contrato inteligente e diferentes casos de uso, para garantir o funcionamento completo da aplicação. Os testes foram executados em um computador com um processador *11th Gen Intel(R) Core (TM) i5-11400H @ 2.70GHz 2.69 GHz* e 32 GB de memória RAM e a execução dos testes teve uma duração de cinco minutos.

A tecnologia *blockchain* possui um grande potencial considerando criptomoedas e contratos inteligentes. Considerando as criptomoedas, a tecnologia já foi validada pela *Bitcoin*. Por meio da *blockchain*¸ a *Bitcoin* consegue realizar transações financeiras sem a necessidade de um terceiro, como um banco ou outra instituição financeira, e consegue evitar o gasto duplo mesmo sendo uma moeda digital. Considerando os contratos inteligentes, a *Ethereum* consegue criar contratos que são autoexecutáveis, acabando com a necessidade de cartórios.

Tanto a *Bitcoin* quanto a *Ethereum* conseguem realizar essas ações por meio dos algoritmos de consenso. Os algoritmos conseguem tornar as *blockchains* descentralizadas, seguras e imutáveis. A descentralização é que retira a necessidade de um terceiro em transações financeiras e evita o gasto duplo, no caso do *Bitcoin*. A imutabilidade é que garante que os contratos inteligentes da *Ethereum* são autoexecutáveis, porque uma vez que as partes entram em acordo sobre as condições, não é possível alterar essas condições.

8 Conclusão

Neste trabalho objetivou-se aplicar os conceitos de *blockchain* e desenvolver um contrato inteligente utilizando *Solidity*, a linguagem de programação da *blockchain Ethereum*. Os conceitos sobre *blockchain* e contratos inteligentes foram aprendidos e colocados em prática na aplicação do projeto.

As vantagens da linguagem *Solidity* são a otimização da linguagem para desenvolver os contratos inteligentes na *Ethereum* e o incentivo da linguagem para os desenvolvedores otimizarem seus códigos. A desvantagem da linguagem *Solidity* é que a linguagem não é otimizada para o desenvolvimento fora do ambiente *blockchain*. Desenvolver em *Solidity* para outros ambientes não é eficiente porque a linguagem possui limitações como poucas estruturas de dados e a alta latência.

Um contrato inteligente que simula uma bolsa de valores foi desenvolvido. Todos os requisitos mapeados durante o projeto foram cumpridos, exceção do requisito não funcional de latência baixa. Assim como a bolsa de valores australiana encerrou o seu projeto que buscava adotar a tecnologia *blockchain* porque a tecnologia possui uma alta latência (Investing.com, 2022), projetos que precisam de soluções com baixa latência não devem adotar a tecnologia *blockchain*. Para casos de uso como esses, uma solução seria usar a tecnologia apenas para parte da aplicação. Considerando a aplicação desse projeto, a solução poderia ser armazenar apenas os registros de transação na *blockchain*, deixando todo processamento de ordens em uma solução de baixa latência.

Testes automatizados para garantir o funcionamento da aplicação foram desenvolvidos. Os testes foram executados em um computador com um processador *11th Gen Intel(R) Core (TM) i5-11400H @ 2.70GHz 2.69 GHz* e 32 GB de memória RAM. Todos os testes foram aprovados e a execução dos testes tiveram uma duração de cinco minutos.

A partir desse trabalho, foi possível colocar em prática todos os conceitos aprendidos sobre a tecnologia *blockchain*. Em trabalhos futuros, propõe-se desenvolver o *front-end* a aplicação e integrar o *front-end* ao contrato inteligente.

Referências

NAKAMOTO, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System.* Bitcoin.org, 2008, https://bitcoin.org/bitcoin.pdf. Acesso em: 28 abr 2023.

BLOCKGEEKS. Proof of Work vs Proof of Stake: What's the Difference? Disponível em: https://blockgeeks.com/guides/proof-of-work-vs-proof-of-stake/. Acesso em: 30 abr 2023.

TRILEMA DA ESCALABILIDADE. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikipedia Foundation, 2018. Disponível em: <https://pt.wikipedia.org/w/index.php?title=Trilema_da_escalabilidade&oldid=53870237>. Acesso em: 21 dez. 2018.

ANÚNCIO DE FUSÃO DA MAINNET. In: *Ethereum* Foundation Blog. *Ethereum* Foundation, 2022. Disponível em: <https://blog.ethereum.org/pt-br/2022/08/24/mainnet-merge-announcement>. Acesso em: 28 ago. 2022.

TAPSCOTT, Don; TAPSCOTT, Alex. *BLOCKCHAIN* REVOLUTION: Como a tecnologia por trás do *Bitcoin* está mudando o dinheiro, os negócios e o mundo. São Paulo: SEBAI-SP editora, 2017.

10 PAÍSES COM MAIS PESSOAS SEM CONTA BANCÁRIA. In: FORBES, 2018. Disponível em: <https://forbes.com.br/listas/2018/06/10-paises-com-mais-pessoas-sem-conta-bancaria>. Acesso em: 26 nov. 2022.

CARDOSO, Bruno. O QUE É "GASTO DUPLO" E COMO O BITCOIN É CAPAZ DE EVITÁ-LO. JusBrasil, 2018. Disponível em: <https://brunonc.jusbrasil.com.br/artigos/584812107/o-que-e-gasto-duplo-e-como-o-bitcoin-e-capaz-de-evita-lo>. Acesso em: 3 dez. 2022.

WHAT IS *BLOCKCHAIN*. IBM, 2022. Disponível em: <https://www.ibm.com/br-pt/topics/what-is-*blockchain*>. Acesso em: 3 dez. 2022.

WHAT ARE SMART CONTRACTS ON *BLOCKCHAIN*. IBM, 2023. Disponível em: <https://www.ibm.com/topics/smart-contracts>. Acesso em: 26 jan. 2022.

NEVES, Lucas. Stock exchange contract using Solidity from Ethereum. GitHub. Disponível em: https://github.com/lucas54neves/stock-exchange-using-solidity. Acesso em: 10 de abril de 2023.

ETHEREUM. Ethereum Roadmap: The Merge. Disponível em: https://ethereum.org/en/roadmap/merge/. Acesso em: 06 mai. 2023.

ETHEREUM. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation, 2022. Disponível em: <https://pt.wikipedia.org/w/index.php?title=Ethereum&oldid=64819076>. Acesso em: 27 nov. 2022.

Oracle. Java Tutorials - Generic Types. Disponível em: https://docs.oracle.com/javase/tutorial/java/generics/types.html. Acesso em: 06 de maio de 2023.

MISRA, S.; KULKARNI, S. Reflection in Java. International Journal of Computer Applications, v. 171, n. 2, p. 33-36, jul. 2017. Disponível em: https://www.ijcaonline.org/archives/volume171/number2/misra-2017-ijca-911266.pdf. Acesso em: 06 maio 2023.

Ethereum. Gas. Disponível em: https://ethereum.org/pt/developers/docs/gas/. Acesso em: 06 maio 2023.

NPMJS. Truffle Teams. Disponível em: https://www.npmjs.com/package/truffle. Acesso em: 07 de maio de 2023.

NPMJS. HardHat. Disponível em: https://www.npmjs.com/package/hardhat. Acesso em: 07 maio 2023.

Ethereum Foundation. Ethereum Documentation. Disponível em: https://ethereum.org/en/developers/docs/. Acesso em: 07 de maio de 2023.

SOLIDITY. Solidity documentation. Disponível em: https://docs.soliditylang.org/. Acesso em: 06 maio 2023.

HARDHAT. Hardhat Documentation. Disponível em: https://hardhat.org/getting-started/. Acesso em: 07 maio 2023.

Hardhat. "Getting Started - Compiling Your Contracts." Disponível em: https://hardhat.org/getting-started/#compiling-your-contracts. Acesso em: 07 de maio de 2023.

INVESTING.COM. Bolsa de Valores Australiana encerra projeto blockchain e demite funcionários. Investing.com, publicado em 03/12/2022. Disponível em: https://br.investing.com/news/cryptocurrency-news/bolsa-de-valores-australiana-encerra-projeto-blockchain-e-demite-funcionarios-1063868. Acesso em: 09/05/2023.

Ethereum Foundation. Ethereum Whitepaper. Disponível em: https://ethereum.org/whitepaper/. Acesso em: 09 de maio de 2023.

Bitcoin Wiki, 2019. Disponível em: https://en.bitcoin.it/wiki/Block. Acesso em: 09 de maio de 2023.