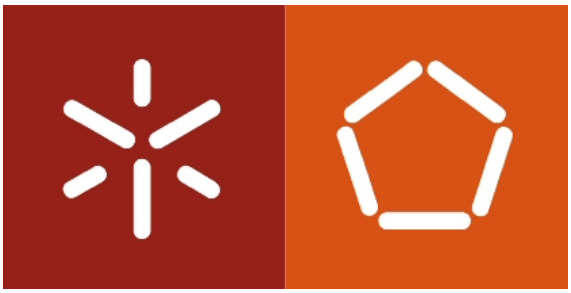

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Engenharia de Sistemas de Software

Arquiteturas de Software

ESS Trading Platform

Inês Alves (A81368)
Ricardo Caçador (A81064)

20 de Dezembro de 2019

Resumo

No âmbito da Unidade Curricular de Arquiteturas de Software, enquadrada no perfil de especialização de Engenharia de Sistemas de Software, este projeto foi desenvolvido com o objetivo de abordar e compreender melhor os conceitos abordados nas aulas desta UC, sendo que se pretende implementar eventuais progressos no seu desenvolvimento à medida que vão sendo introduzidos novos conhecimentos.

Conteúdo

1 Introdução

O grande mote desta plataforma prende-se na sustentabilidade e suporte de negociações de ações financeiras referentes não só a *commodities* (ouro, prata, moeda...), mas também a acções de empresas e organizações (Google, Apple, IBM...).

Estas negociações terão por base o sistema de CFD - *Contract For Differences*, onde um comprador e um vendedor pactuam a diferença de valor de uma ação entre o tempo em que o contrato se encontra aberto. De notar que o comprador acaba por nunca tomar posse do bem em questão. O que acontece é que este recebe as receitas (ou assume as perdas, pagando ao vendedor) provenientes das flutuações de mercado desse bem.

Assim, a plataforma terá que ser capaz de manter os valores das ações a serem lidados via CFDs, permitir que duas forças motrizes deste tipo de negociações (compradores e vendedores) possam abrir contas com um saldo inicial de investimento e possam também estipular CFDs sobre as ações disponíveis. Por fim, repare-se que, tanto os compradores como os vendedores, poderão monitorizar o conjunto de CFDs em que atuam, assim como o valor de cada ação/posição.

2 Requisitos para o Sistema

Uma plataforma de negociação permite que os seus utilizadores (compradores e vendedores) realizem negociações de CFDs. Para a implementação de todas as funcionalidades da aplicação, foram levantados os seguintes requisitos:

- Um investidor deverá registar-se no sistema através de: e-mail, password e nome;
- Uma vez registado, o investidor poderá iniciar sessão no sistema utilizando apenas o seu e-mail e a password definida aquando do seu registo;
- Ao aceder à aplicação, o investidor autenticado deverá ter disponível uma lista de eventos sobre os quais poderá agir;
- Um investidor deverá poder comprar e vender ativos;
- O sistema deverá conseguir, em tempo real, monitorizar a atividade de cada investidor;
- O sistema deverá ter uma lista com os valores mais atuais dos ativos adquiridos pelos investidores;
- Cada abertura de negociação deverá seguir o formato: valor de abertura, valor atual, quantidade negociada, data e hora de abertura e, se aplicável, valores de "Take Profit"e "Stop Loss";
- Cada fecho de negociação deverá seguir o formato: valor de fecho, resultado da negociação e data e hora do fecho da posição;
- A plataforma deverá manter uma lista de ativos com que o investidor teve interação;
- Quando um negócio é fechado, todos os investidores devem ser notificado e os respetivos saldos atualizados consoante o resultado do negócio.
- Para determinar o valor ganho/perdido num investimento, deverá ser definido, para cada evento, as odds para os possíveis resultados e, sobre essas odds, será calculado o valor ganho/perdido.

2.1 Requisitos de Qualidade

Cada requisito de qualidade é composto por 6 partes:

- **Stimulus:** condição que requer uma resposta quando chega ao sistema;
- **Stimulus source:** entidade que gera um estímulo;
- **Response:** atividade realizada como resultado da chegada do estímulo;
- **Response measure:** prova de que o requisito foi testado e que funciona;
- **Environment:** condições em que o estímulo é gerado;
- **Artifact:** o que é estimulado.

Qualidade	Motivação
Simplicidade	A plataforma deverá ser simples, permitindo que a sua utilização seja fácil e direta.
Eficiência	Os dados apresentados na plataforma devem sempre permanecer atualizados e sem falhas.

Tabela 1: Requisitos de qualidade

Tendo estes tópicos como ponto de referência, bem como os requisitos levantados anteriormente, foram desenvolvidos os seguintes requisitos concretos de qualidade:

- **Modificação**
  No caso do administrador do sistema pretender alterar, por exemplo, algo numa posição, é necessário que esta alteração ocorra sem prejudicar o sistema. Desta forma, o sistema realiza as alterações pretendidas assegurando o seu bom funcionamento, sem atrasar as restantes operações e o estado do sistema é alterado instantaneamente.

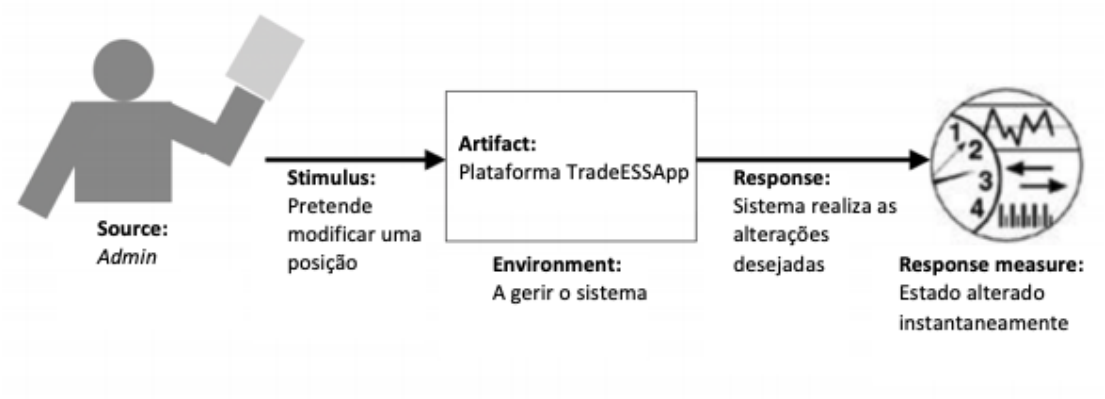

Figura 1: Requisito de Qualidade - Modificação

2.1.1 Público Alvo

Alvo	Objetivos
Docente	Para este trabalho o público alvo é o docente da UC, uma vez que esta plataforma procura ser algo simples apenas para fins de inicialização do projeto.

Tabela 2: Público alvo do sistema

2.2 Requisitos Não Funcionais

A fim de tornarmos a nossa implementação e compreensão do projeto mais lúcida, definimos alguns requisitos não funcionais que restringuem o trabalho de um arquiteto de *software*.

Restrição	Descrição
Implementação em Java	A plataforma de trading terá a sua implementação em Java, por isso é possível que surjam restrições inevitáveis para a implementação da solução. Estas restrições advêm da própria linguagem de programação.
Imparcialidade	A plataforma de trading terá de ser imparcial, isto é, independentemente do sistema operativo utilizado, deverá funcionar de igual forma e com bom desempenho.
Execução por terminal	A plataforma de trading não terá grande desenvolvimento gráfico, pelo que toda a sua execução/utilização deverá partir segundo uma linha de comandos em terminal.
Calendarização	O desenvolvimento da plataforma será fortemente influenciado pelo tempo disponível para a sua implementação.
Documentação da Arquitetura	Toda a implementação da plataforma de trading será documentada fazendo uso da ferramenta javadoc.

2.3 Contexto do Sistema

Ainda antes de partirmos para a implementação do sistema propriamente dita, é necessário fazermos referência aos elementos externos implícitos ao funcionamento da plataforma, sendo que estes interagem com ele. Consideremos, então, os seguintes fatores:

- A plataforma de *trading* terá que ser executada pelo terminal ou fazendo uso de um IDE, uma vez que não fizemos uso de nenhuma ferramenta de implementação de interfaces gráficas (ex.: Java SWING). Assim sendo, o nosso sistema possui uma interface gráfica bastante rudimentar, como iremos mostrar mais adiante;
- Quanto à base de dados que suporta o sistema, esta foi criada com a ajuda da ferramente SQLiteStudio. Inicialmente, tinhamos feito uso da ferramenta MySQL, no entanto, esta mostrou-se bastante mais complexa do que a nova implementação. Com o SQLiteStudio não temos a preocupação de verificar *passwords* para executar o programa. Uma vez que somos um grupo de 2 pessoas, era necessário estarmos a confirmar sempre se a *password* utilizada era a correta, relevando-se uma desvantagem em termos de comodidade. No entanto, a fim de executar o programa, é nessário ter a base de dados criada pelo grupo (database.db);
- Uma vez que a implementação do código foi feita na linguagem de programação Java, é necessário ter o JDK 11 instalado na máquina utilizada, a fim de compilar corretamente o código.

3 Desenho da Solução

Estando feito o levantamento de requisitos para a plataforma de negociações TradeESSApp, foram desenvolvidos, através do uso da linguagem UML (Unified Modelling Language), as seguintes vistas de estrutura, comportamento e alocação:

- **Modelo de Domínio**
- **Diagrama de Use Cases**
- **Diagramas de Sequência de Subsistema**
- **Diagrama de Classes**
- **Diagrama de Packages**
- **Diagrama de Instalação**

4 Modelo de Domínio

Numa primeira abordagem e sendo este um dos requisitos enunciados no enunciado do projeto, construímos o modelo de domínio associado à nossa plataforma de *trading*.

Assim, pela análise do enunciado, foi-nos possível identificar algumas entidades principais: **User** e **Position** e, à posteriori, todos os componentes que completavam o sistema e as entidades necessárias.

A construção do modelo de domínio facilitou-nos bastante a modelação do projeto, projetanto o que seriam as possíveis classes da nossa plataforma.

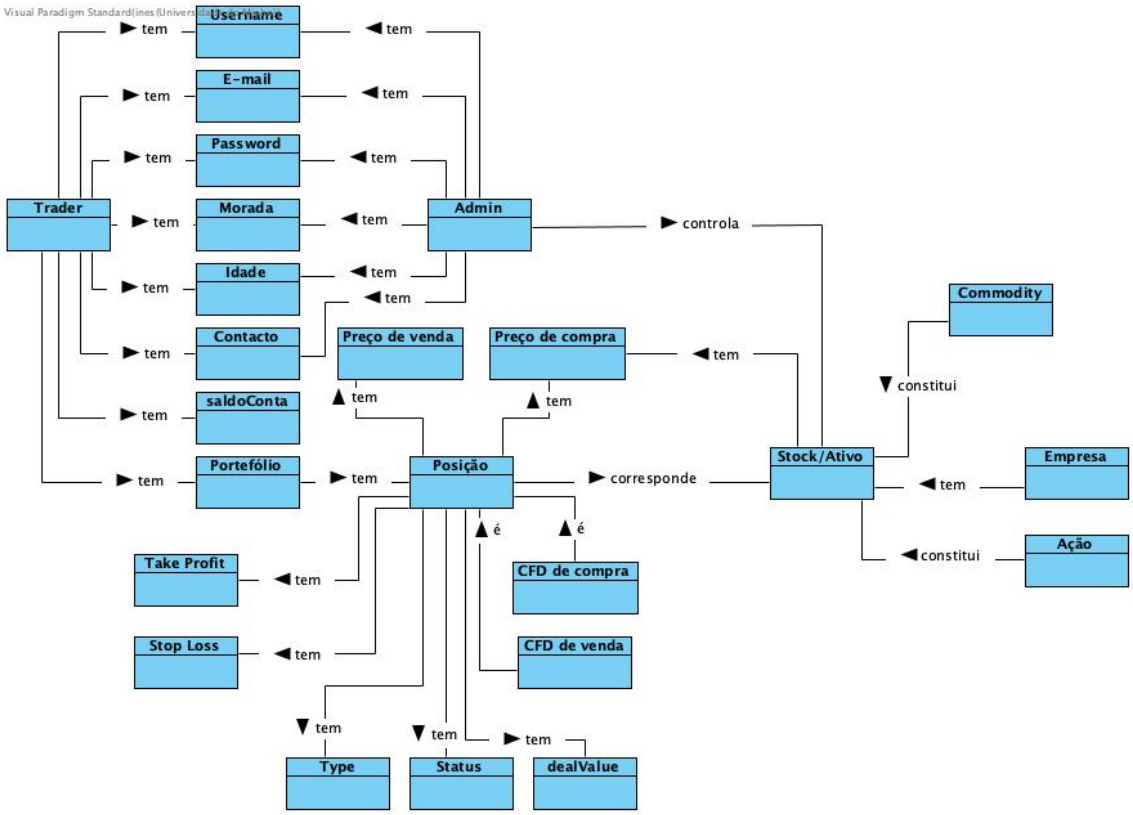

Figura 2: Modelo de Domínio

5 Diagrama de Use Cases

A partir do modelo de domínio apresentado na secção anterior, conseguimos, com maior facilidade, modelar o nosso modelo de Use Cases para o sistema em questão. Para este efeito, decidimos implementar apenas um ator no sistema: ***User***. Como podemos perceber, o *User*, isto é, o investidor, é, no nosso entender, o principal utilizador do sistema, uma vez que será ele que irá dar-lhe o maior uso. Ainda que na versão anterior deste trabalho tenha sido implementado um ator *Admin* que se tratava de um administrador responsável pela criação e manipulação de dados do sistema, a decisão de o absolver foi unânime.

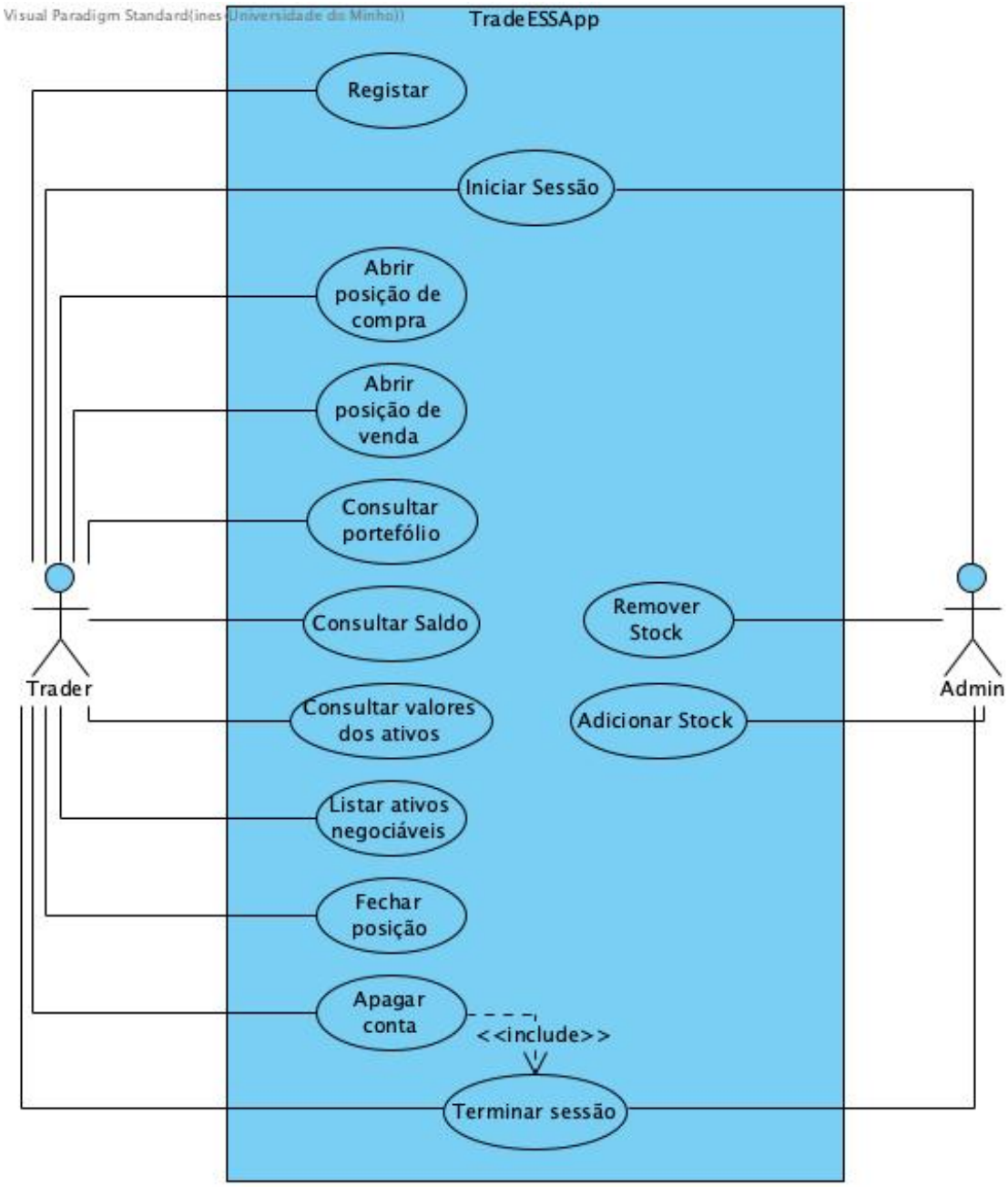

Figura 3: Diagrama de Use Cases

6 Diagrama de Classes

A partir do modelo de domínio, do diagrama de Use Cases apresentados anteriormente, foram identificadas as principais entidades do sistema e, consequentemente, potenciais candidatas a serem classes no desenvolvimento desta plataforma de *trading*. Como tal, passamos agora para a construção do Diagrama de Classes, dando mais um passo no desenvolvimento deste sistema de software.

Para realizar a correta construção deste diagrama foi necessário ter em atenção as classes que o nosso sistema iria possuir, bem como os métodos de que as mesmas iriam fazer uso.

Um diagrama de classes consiste na representação estrutural das classes que servem de modelo para os objetos do sistema, explicitando também as interações entre elas e o papel que cada uma tem na realização das operações solicitadas pelos utilizadores do mesmo.

Cada classe representa um grupo de objetos que partilham o mesmo tipo de estrutura (atributos e relacionamentos) e as mesmas operações, sendo que estas podem restar relacionadas umas com as outras através de associações.

Assim, construimos um modelo da aplicação onde é implementada a persistência através das diversas classes DAOs, ou seja, guardando os dados numa base de dados. De notar que todas as classes possuem os respetivos Getters e Setters das variáveis de instância, apesar de não estarem representadas no diagrama de classes abaixo apresentado, a fim de não prejudicar a perceção deste.

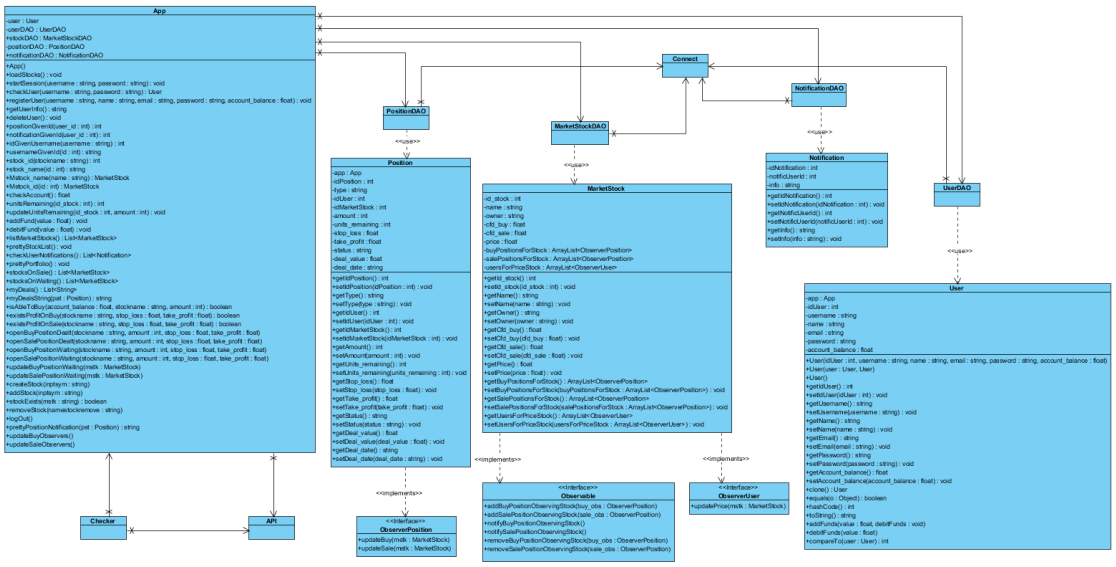

Figura 4: Diagrama de Classes

Foram necesárias diversas alterações à solução anterior, a fim de satisfazer os requisitos inicialmente impostos e ainda implementar o modelo arquitetura de *Sofware Observer Pattern*.

Para além disso, foi necessário implementar uma espécie de monitorização sobre as variações dos preços dos ativos no mercado. Para isso, foi desenvolvida uma classe *Checker* que extende *Thread*. Assim, o método *run* inserido nesta classe permite que, de 2 em 2 minutos, seja feita uma verificação sobre os preços dos ativos, bem como a sua atualização. Isto significa que, desta forma, os preços são atualizados na base de dados e as posições que se encontrem abertas para determinados ativos são notificadas.

Por fim, de modo a que o programa cumprisse as normas de desenho do *Observer Pattern*, foi necessário permitir ao utilizador ser notificado acerca de eventuais alterações. Sendo assim, foi implementada uma classe *Notification* e o seu respetivo DAO (*NotificationDAO*), de modo a permitir a persistência dos dados. É ainda importante notar que existem dois tipos de notificação:

- notificação gerada sempre que o utilizador possua posições (de qualquer tipo) cujo estado esteja em "Waiting"e a compra ou venda tenha sido efetuada (graças à verificação dos preços por parte da classe *Checker*);
- notificação enviada sempre que algum ativo novo é adicionado ao mercado, possibilitando os utilizadores a abrirem posições com o novo ativo.

Como já foi referido, o foco do grupo passou pela implementação da arquitetura de *software* de *Observer-Observable*, passando, então, a existir uma *interface Observer* e uma *interface Observable*.

A classe *MarketStock* implementou a interface Observable, pelo que todos os ativos passam a ser "observados"face a alterações, mais concretamente no que diz respeito aos preços de compra, venda e preço real.

A classe *Position* implementou a interface *ObserverPosition*, ou seja, as posições cujo estado é igual a "*Waiting*"passam a observer as variações nos preços de venda e compra dos ativos que abrangem.

Note-se porem que, face ao novo requisito, a classe User passaria a ser um ObserverUser (implementando a interface Observer) mas tal requisito nao foi totalmente implementado e sera abordado mais a frente neste documento.

Atraves da implementacao do *Design Pattern Observer-Observable*, a aplicacao assumiu uma nova dinamica. Sendo assim, foram considerados que seriam *Observers* todas as posicoes abertas pelos utilizadores e que nao tivessem sido negociadas na hora (ou seja, toda a posição com o status igual a "Waiting"). Sempre que uma posicao deste tipo fosse criada e inserida na base de dados, ela seria adicionada tambem a um determinado array existente no Observable MarketStock:

- se for uma posição de compra (*type = "Buy"*) e com *status = "Waiting"*, ela deve ser adicionada à lista de posições de compra em espera do ativo referenciado pela posição;
- se for uma posição de venda (*type = "Sale"*) e com *status = "Waiting"*, ela deve ser adicionada à lista de posições de venda em espera do ativo referenciado pela posição;

Assim, sempre que houver uma alteracao em algum dos precos de um ativo, a lista de posicoes respetiva sera notificada, ou seja, para o caso em que haja uma alteracao na posição de compra de um ativo, todas as posicoes de compra que estejam em espera serao notificadas (notifyBuyPositionObservingStock) acerca da alteracao da posição de compra, sendo que depois a aplicacao tem a capacidade de discernir se para esse valor a compra do ativo pode ser feita e a posicao de compra correspondente podera ser fechada, sendo que, em caso afirmativo, a posicao sera removida da lista de posicoes em espera. O raciocínio é análogo para as posições de venda.

6.1 Classes

6.1.1 *User*

Classe que representa um utilizador geral da plataforma, contendo as suas características principais como o nome, e-mail, *password*, *username*, *account_balance* entre outras...

Todos estes parametros criados aquando o registo de um utilizador sao registados na base de dados e permanecem imutaveis, com excecao do *account_balance* que vai sendo atualizado de acordo com as compras e vendas do utilizador. Este tem ainda a hipótese de adicionar fundos à sua conta.

6.1.2 *Position*

Esta classe contém toda a informação que suporta a abertura de uma posição de compra ou venda por parte do utilizador da aplicação. Desta forma, se o utilizador decidir abrir uma posição de compra, será criada uma nova posição (*Position*) com *"type = Buy"*, indicando, desta forma, que se trata de uma posição de compra, com o ID na base de dados da ação que foi solicitada para

compra, da quantidade desejada para compra, do *stop loss* e *take profit* definidos pelo *trader* para a compra e do valor dessa mesma. Existe também um parâmetro *status* que serve para verificar se, no momento da abertura da posição de compra, foi possível, de imediato, fazer o negócio. Para isto, o parâmetro pode assumir os valores "Dealt"se a compra for realizada; ou "Waiting", no caso do pedido de compra ficar em fila de espera, pois pode que não ser um negócio rentável naquele momento.

Posto isto, caso o utilizador faça um pedido de venda à plataforma, o raciocínio é o mesmo, sendo que a única diferença é que a posição criada terá *type="Sale"*.

Note-se que nas classes referidas anteriormente existem também a si associados identificadores únicos (IDs) utilizados, maioritariamente, para efetuar pedidos à base de dados, visto que as pesquisas são feitas fazendo uso dos mesmos.

6.1.3 *MarketStock*

Representa o mercado de *stocks*/ações existente. Esta classe contém toda a informação que suporta a existência de uma ação na plataforma. Como tal, foi considerado que um *MarketStock* teria:

- um ID;
- um *name*, que corresponde ao símbolo da ação na API;
- um *owner*, que corresponde ao nome da companhia que possui a ação na API;
- um *cfdBuy*, corrrespondendo ao *bid* da ação na API;
- um *cfdSale*, correspondente ao *ask* da ação na API.

Os valores de *cfdBuy* e *cfdSale* vão sendo atualizados pela API em certos períodos de tempo através de threads de monitorização.

6.1.4 API

Entidade representativa da API utilizada;

6.1.5 App

Classe principal desta plataforma. É aqui que se realizam, efetivamente, todas as operações de registo, início de sessão, término de sessão, criação de *stocks*, etc...

Todas estas classes encontram-se na camada de negócio - *Business Package*. Foram ainda implementados outros *packages*, utilizados para guardar dados na base de dados e desenvolvimento de interfaces do sistema.

7 Diagrama de Packages

Uma vez que todas as classes já se encontram definidas, podemos, agora, visualizar a integração e relacionamento entre elas através de um Diagrama de *Packages*.

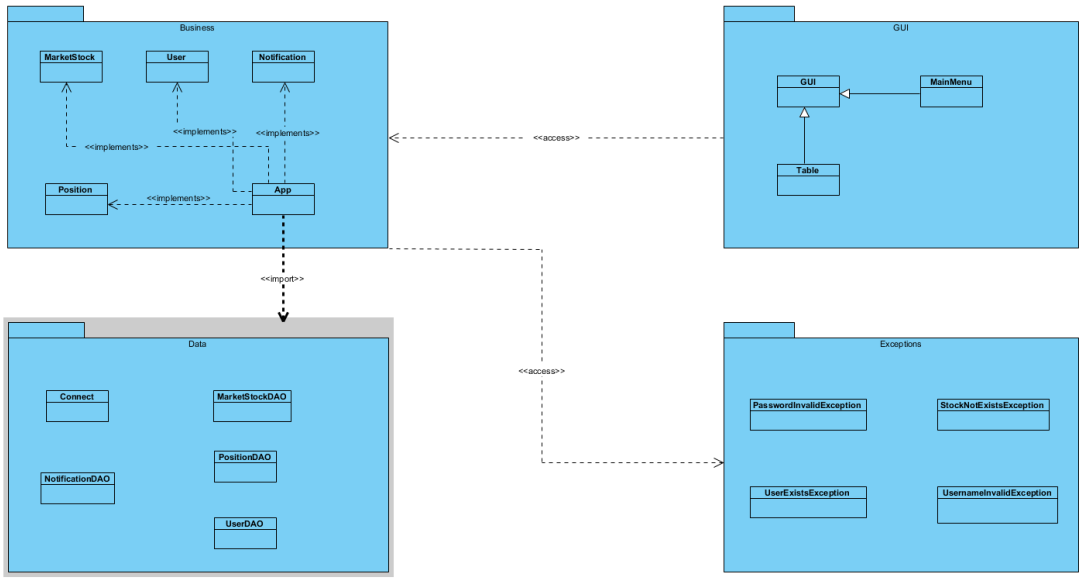

Figura 5: Diagrama de Packages

Este diagrama "resume"a modelação realizada até ao momento, traduzindo a arquitetura usada, bem como o padrão principal que trata da comunicação entre os diversos componentes do sistema. Cada *package* corresponde a um mecanismo de agrupamento genérico com elementos relacionados, sendo que estes podem ser classes, interfaces, entre outros...

É importante notarmos ainda que, neste tipo de diagramas, os elementos de um determinado *package* podem ser, ou não, vísiveis para outros fora dele. Os *packages* podem também apresentar dependências entre si, caso a alterção de um afete outro.

7.1 *Business Package*

Package relativo à lógica do negócio, responsável pela comunicação entre a camada de persistência e a camada de apresentação.

7.2 *GUI Package*

Contém todas as classes responsáveis pela interface com o utilizador, fazendo a comunicação entre o utilizador e a plataforma.

7.3 *Data Package*

Responsável pela persistência de todos os dados e informação sobre a plataforma, estabelecendo comunicação com a camada de negócio através do *facade*. Então, aqui são implementados todos os DAOs.

8 Diagramas de Sequência de Subsistema

Definidos os subsistemas através do Diagrama de *Packages*, foram definidos também alguns diagramas de subsistema relativos a alguns *use cases*.

8.1 Iniciar Sessão

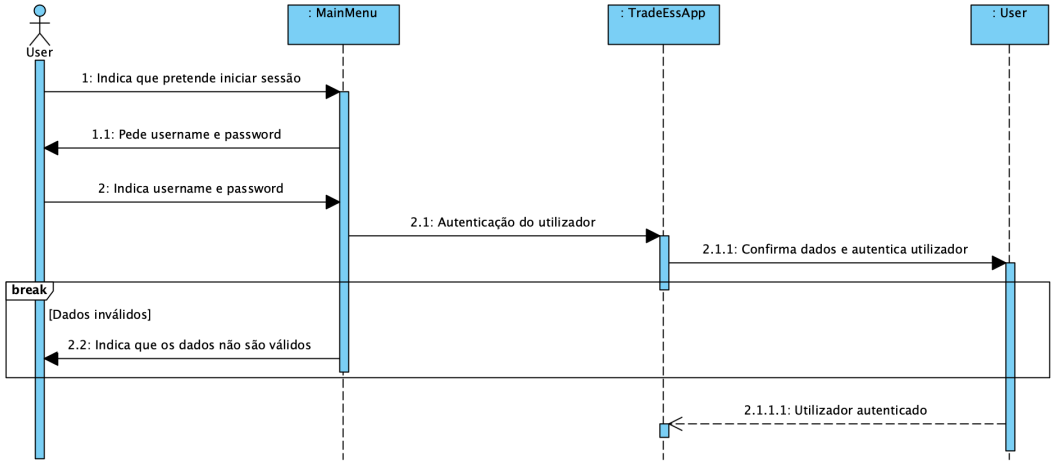

Figura 6: Diagrama de Sequência de Subsistema: Iniciar Sessão

8.2 Registar Utilizador

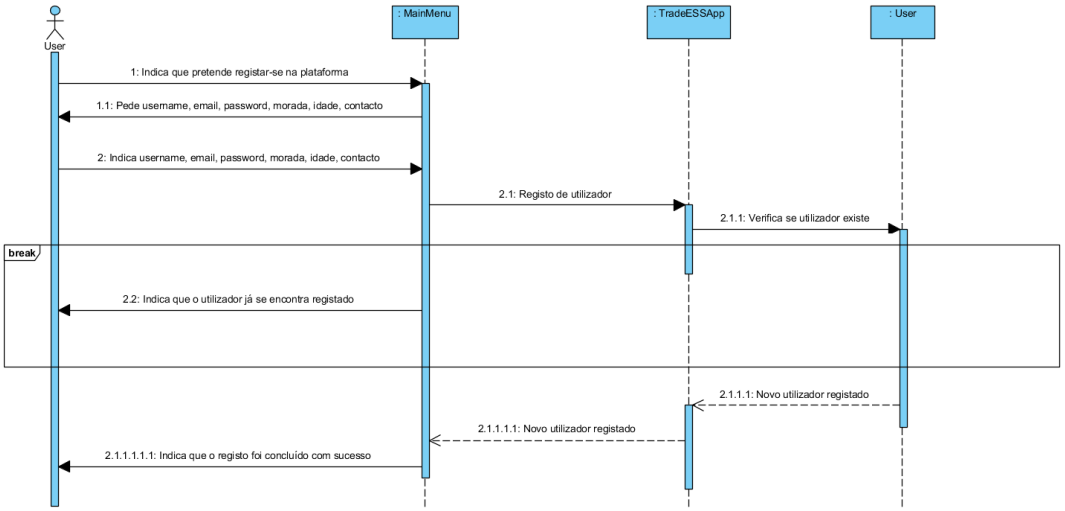

Figura 7: Diagrama de Sequência de Subsistema: Registar Utilizador

8.3 Abrir posição de compra

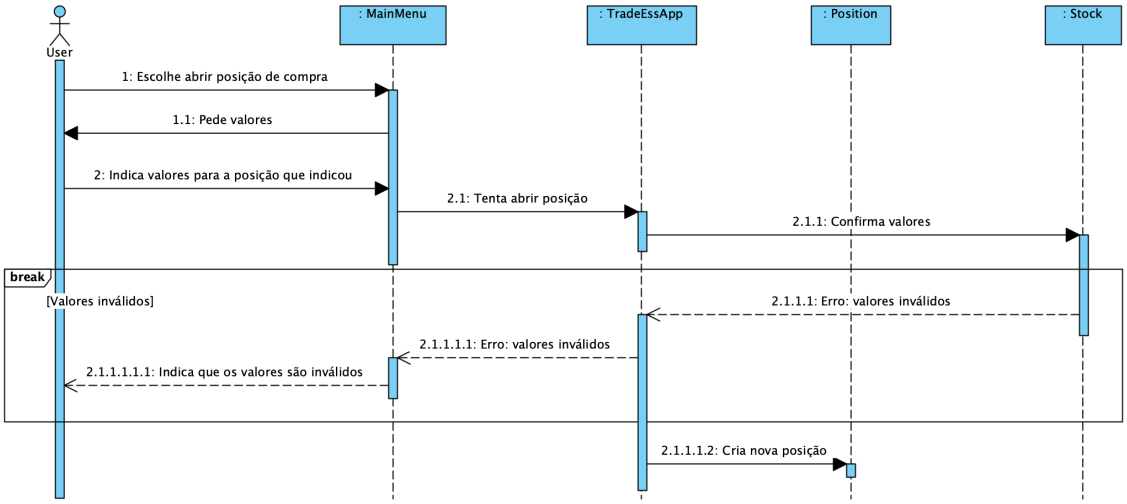

Figura 8: Diagrama de Sequência de Subsistema: Buy Position

9 Diagrama de Instalação

Nesta fase, procedemos à esquematização do sistema de *software* ao nível dos componentes físicos que ele abrange, modelando, por isso, a sua topologia de *hardware*. Apesar do ponto fulcral do UML ser a estruturação e o comportamento do *software* de um sistema, este tipo de diagramas aborda também a porta da aplicação a ser desenvolvida, demonstrando os componentes *hardware* nos quais o *software* desenvolvido se encontra implementado.

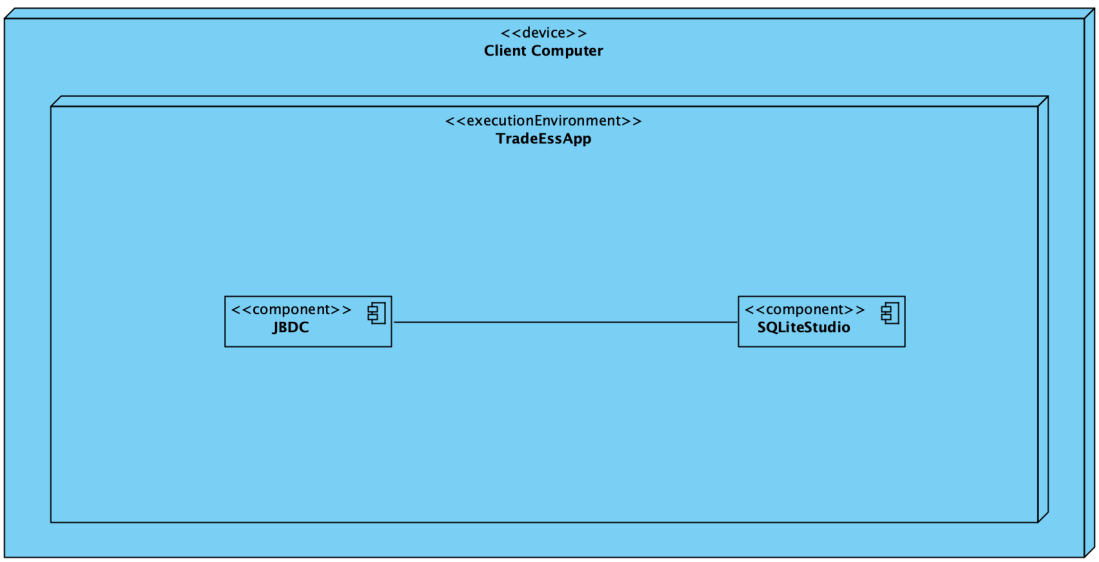

Figura 9: Diagrama de Instalação

10 Alterações relativas ao trabalho anterior

1. **Inclusão de *threads* para verificação de valores**;
2. **Inclusão do novo requisito**;
3. **Inclusão do *Observer Pattern***

Como podemos observar, ambas as implementações se regem por uma arquitetura em camadas, uma vez que cada camada contem objetos relacionados com o conceito particular que representam. Para além disso, nota-se ainda de forma evidente um aumento do nivel de dependencia entre as camadas a medida que nos aproximamos da camada onde os dados sao guardados. Por outras palavras, podemos verificar que a camada *GUI* e extremamente dependente das que se encontram "abaixo"dela (camada *Business* e camada *Data*) uma vez que a interacao com o utilizador so e plenamente concretizada gracas a execucao correta da aplicacao que e assegurada pelas outras camadas subjacentes.

Por outro lado, a camada *Data* (onde se encontram todos os DAOs necessários) não apresenta qualquer dependência relativamente às outras camadas, uma vez que o seu objetivo e garantir a persistencia dos dados.

Assim, podemos concluir que o raciocínio se manteve constante, sendo apenas necessário fazer algumas modificações para a implementação do *Design Pattern Observer-Observable*.

11 Novo Requisito

Para realizar a correta implementação do novo requisito, foram feitas algumas modificações no programa. Para suportar este novo requisito a ideia passou por diferenciar o tipo de *Observers* em 2: *ObserverPosition* e *ObserverUser*, pelo que, para além de termos observadores atentos sobre as variações nos CFDs de compra e venda de cada ativo, temos também observadores interessados em serem notificados aquando da variação do preço dos CFDs.

Assim, nesta solução, as posições que se encontram abertas estarão interessadas em variações nos CFDs, enquanto que os utilizadores da plataforma estarão interessados apenas no preço dos mesmos. Assim, a classe *MarketStock* passa a fazer uso de um *ArrayList¡ObserverUser¿ usersForPriceStock* responsável por armazenar os utilizadores que estão a observar o preço de um CFD.

12 Interface

12.1 Menu Principal

Quando iniciamos a nossa plataforma podemos encontrar o seguinte menu:

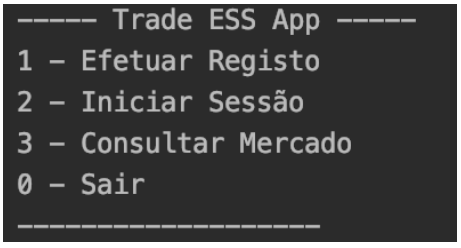

Figura 10: Menu Principal da plataforma

De seguida, podemos efetuar o registo de um utilizador:

```
---- Efetuar Registo ----
Insira o seu email:
a81368@alunos.uminho.pt
Insira o seu username:
inês
Insira a sua password:
password
Insira a sua morada:
Rua da Universidade
Insira a sua idade:
20
Idade válida!
Insira o seu contacto telefónico:
910123456
```

Figura 11: Menu de Registo de um utilizador

E, então, se informarmos o sistema que queremos iniciar sessão, é-nos apresentado o seguinte menu:

```
Deseja iniciar sessão? (Sim | Não)
Sim
Sessão iniciada com sucesso.
---- Bem-vindo, inês! ----
Nome: inês
Saldo:0.0
--------------------
1 - Consultar e gerir saldo
2 - Abrir Buy Position
3 - Abrir Sale Position
4 - Consultar Portefólio
5 - Apagar Conta
6 - Terminar sessão
--------------------
```

Figura 12: Menu de funcionalidades de um utilizador

A partir daqui, podemos usufrir de todas as funcionalidades deste sistema, como é o caso da consulta e gestão do saldo:

```
--- Gerir Saldo ---
1 - Ver saldo
2 - Depositar Dinheiro
0 - Sair
-------------------
2
Qual a quantia que quer depositar?
10
10.0
```

Figura 13: Menu referente à consulta e gestão de saldo

De notar ainda que, caso iniciemos sessão com as credencidais de administrador, obtemos o seguinte resultado:

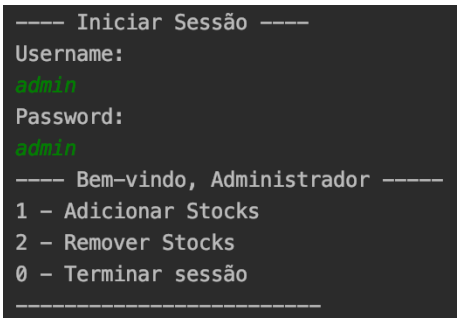

Figura 14: Menu de início de sessão de um Administrador

13 Conclusão e Análise Crítica

Realizado todo o planeamento e modelação, podemos afirmar que o trabalho aqui apresentado respeita os objetivos mencionados no enunciado do projeto. Apesar de não ser algo necessário nesta fase do trabalho (que só pedia um esqueleto do código) a nossa plataforma já possui algumas funcionalidades que se aproximam mais de uma plataforma real para *trading*.Estabelece uma conexão à base de dados e processa a informação relativa aos vários utilizadores, negócios/contratos e posições.

A fim de termos uma plaforma de *trading* fiável, tentámos fazer uso de uma API (Yahoo Finance), conseguindo, desta forma, dados reais e de confiança. O objetivo seria serem introduzidas posições na base de dados, posições de compra e de venda, introdução correta de certos valores das ações provenientes da API, consultar posições que o utilizador tenha na sua posse, entre outras...

No entanto, infelizmente, esta implementação não conseguiu ser finalizada, pelo que não foi implementada a parte relativa à utilização da API. Apesar disso, a estratégia de solução para este problema já foi discutida, passando pela implementação de *threads*, atualizando os valores que pretendemos. Estudaremos ainda a viabilidade de realizar um programa Cliente-Servidor, possibilitando o acesso de vários clientes ao servidor e, mais uma vez, recorrendo a *threads*.

Ainda assim, apesar de algumas limitações desta implementação, fazemos um balanço positivo deste trabalho. Contamos ainda, como já referido, numa implementação futura, conseguir atribuir a real dinâmica ao sistema, conseguindo, por isso, uma monitorização real dos valores das posições e o aperfeiçoamento desta solução.