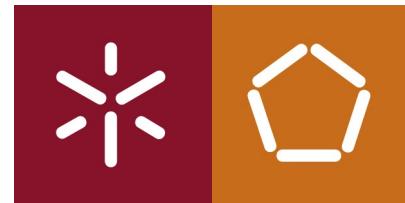


UNIVERSIDADE DO MINHO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA



Online Trading Platform

Arquiteturas de Software

Miguel Oliveira (pg41088)
Pedro Moreira (a82364)

Outubro 2019

Conteúdo

1	Introdução	2
1.1	Estrutura do relatório	2
2	Funcionalidades	3
2.1	Levantamento das funcionalidades	3
3	Modelação	4
3.1	Modelo de Domínio	4
3.2	Diagramas de comportamento	5
3.2.1	Use Cases	5
3.2.2	Diagramas de Sequência	5
3.2.3	Diagrama de Atividade	8
3.2.4	Diagrama de Máquinas de Estado	9
3.3	Atributos de qualidade	10
3.4	Diagrama de Classes	11
3.5	Design Patterns e outras estratégias	12
3.5.1	Programar para interfaces, não implementações	12
3.5.2	Princípio da responsabilidade única	13
3.5.3	Factory Method	13
3.5.4	Observer	15
3.5.5	Strategy	15
3.5.6	Facade	16
3.5.7	DAO	17
3.5.8	Singleton e Database Connection Pool	18
3.6	Modelo Lógico da Base de Dados	19
4	Implementação do novo requisito	20
5	Conclusão	23
Appendices		24
A	Atributos de Qualidade	24
B	Diagramas de Sequência	26
C	Diagrama de Classes	28

1 Introdução

O presente relatório é referente à 2^a Fase do projeto da unidade curricular Arquiteturas de Software. Neste trabalho foi desenvolvida uma arquitetura para um sistema simplificado de *Electronic Trading Platform*. Para tal foram utilizados Design Patterns e um Architectural Pattern, temas abordados ao longo da unidade curricular.

1.1 Estrutura do relatório

Neste relatório serão, primeiramente, expostos, de forma sucinta, os requisitos da aplicação e posteriormente as funcionalidades do sistema.

De seguida, será exposta a modelação UML realizada através do Modelo de Domínio, Diagramas de Comportamento como, por exemplo, Diagrama de Use Cases, entre outros. Será, também, dedicada uma secção aos atributos de qualidade esperados para a aplicação.

Na secção seguinte são fundamentados os Design Patterns usados no projeto, assim como outras estratégias para o desenvolvimento do mesmo.

Por fim, temos a resolução do requisito extra e a conclusão.

São incluídos, em anexo, as imagens relativas às decisões tomadas na 1^a Fase do projeto, para comparação com o resultado final.

2 Funcionalidades

Nesta secção serão enumeradas algumas das funcionalidades que o sistema deve implementar.

2.1 Levantamento das funcionalidades

Como já referido, o sistema deverá ser capaz de reproduzir uma plataforma de *Electronic Trading*. Para isto foram recolhidas as seguintes funcionalidades:

- O utilizador deve ser capaz de realizar um registo;
- O utilizador deve ser capaz de fazer login no sistema;
- O utilizador deve ser capaz de adicionar e levantar fundos da sua conta;
- O utilizador deve conseguir visualizar o estado atual do mercado;
- O utilizador deve poder abrir e fechar contratos (*CFDs*), dos tipos *Long* e *Short*;
- O utilizador deve ter a possibilidade de definir limites *Take Profit* (TP) e *Stop Loss* (SP) para os seus contratos;
- O utilizador deve ser capaz de visualizar o portfólio dos seus *CFDs* ativos.

Decidimos, também, que o sistema deveria ter um administrador para gerir os utilizadores e o mercado (apesar de não ser necessário no âmbito da UC).

Estas funcionalidades podem ser vistas na Secção 3.2.1, através do diagrama de Use Cases.

3 Modelação

3.1 Modelo de Domínio

Na Figura 1, podemos ver todo o domínio do nosso problema, duma forma simples e legível.

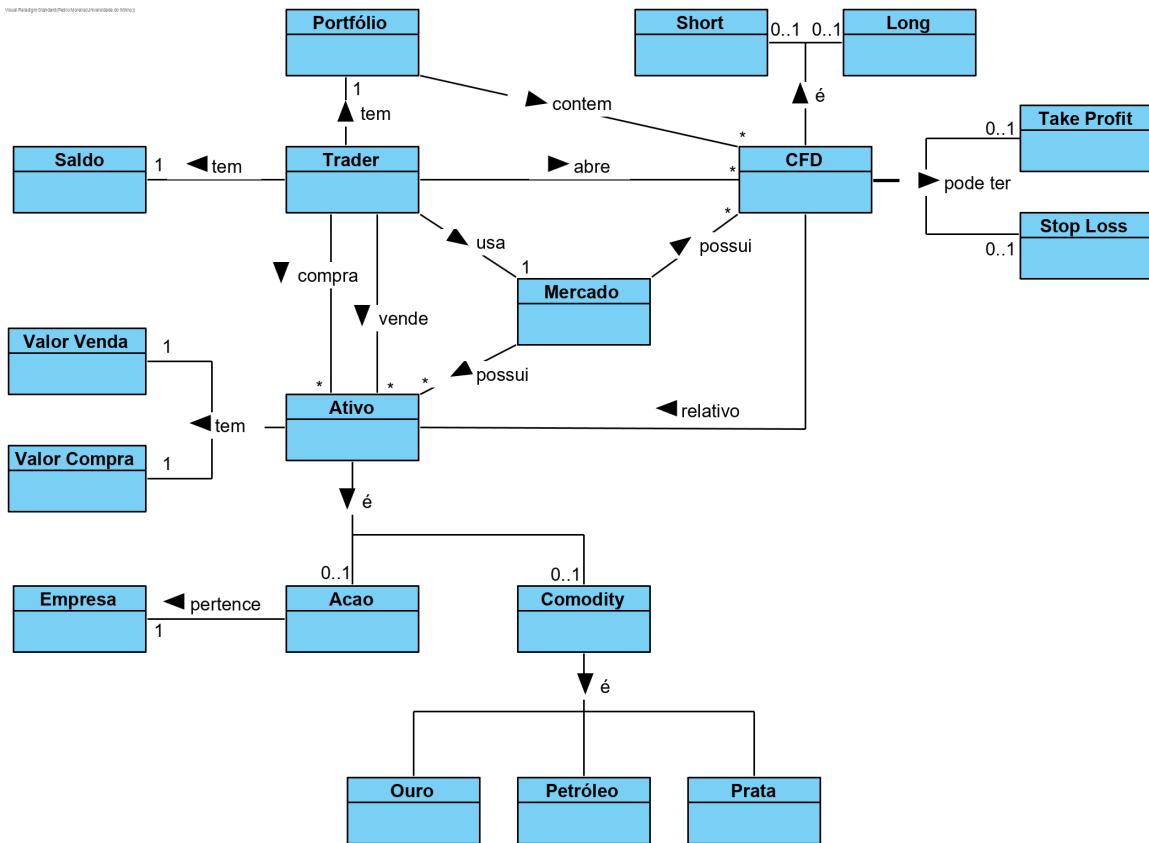


Figura 1: Modelo de Domínio

Focando no *Trader*, podemos ver que este interage com um mercado de ativos. Para tal, ele abre posições no mercado denominadas CFD (*Contract for differences*). Estas posições podem ser de dois tipos: *long* ou *short*, podendo cada uma delas ter um limite de *Stop Loss* e/ou um limite de *Take Profit*. Assim o *Trader* vai construindo o seu portfólio de CFDs. Falta apenas descrever o que é um *ativo*: um ativo pode ser uma ação ou uma *commodity*, tendo este dois valores monetários associados, valor de venda e valor de compra.

3.2 Diagramas de comportamento

3.2.1 Use Cases

Como já referido anteriormente, esta secção demonstra as funcionalidades do sistema, através do respetivo diagrama da Figura 2.

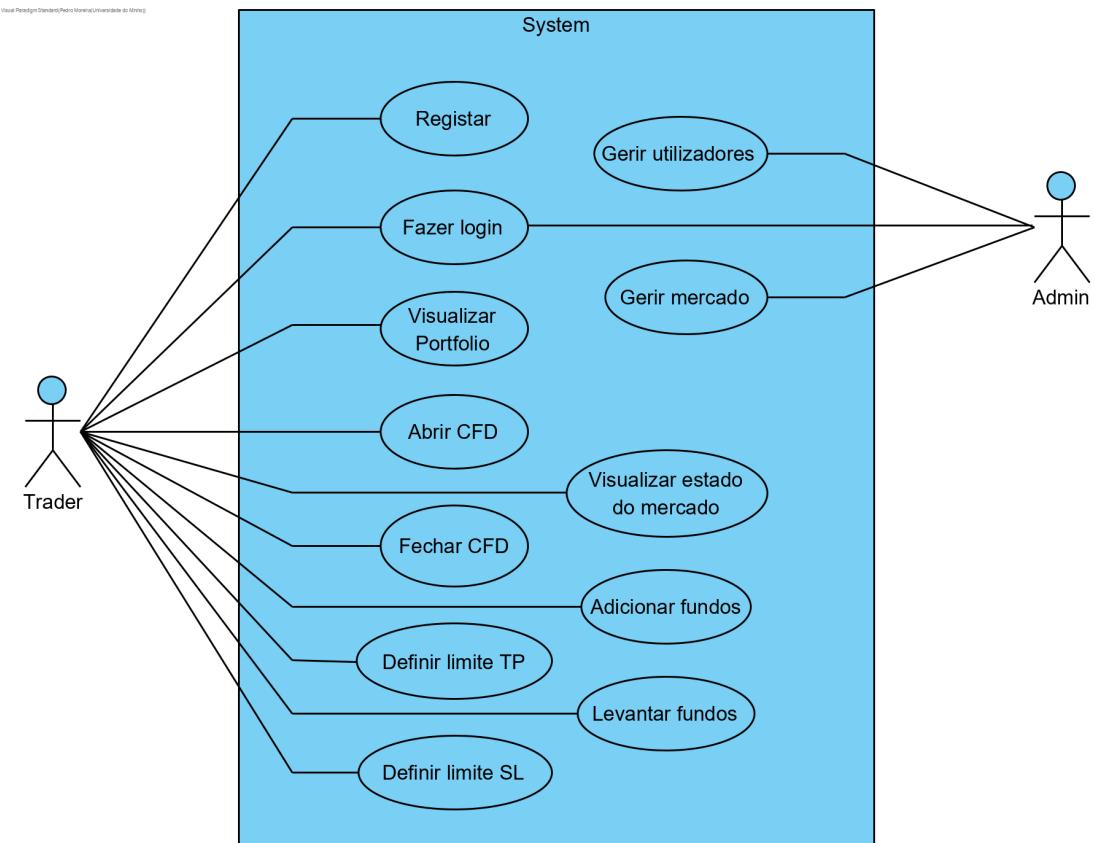


Figura 2: Diagrama de Use Cases

3.2.2 Diagramas de Sequência

Na presente secção, estão representados os diagramas de sequência para as 4 principais funcionalidades, identificadas pelo grupo. Estas são:

- Adicionar fundos, pois sem eles o utilizador não pode fazer nada;
- Ver o estado atual do mercado, para que seja possível comprar/vender ações;
- Abrir um CFD, objetivo principal da aplicação;
- Ver portfólio das posições abertas no mercado.

De seguida, apresentam-se os respetivos Diagramas de Sequência:

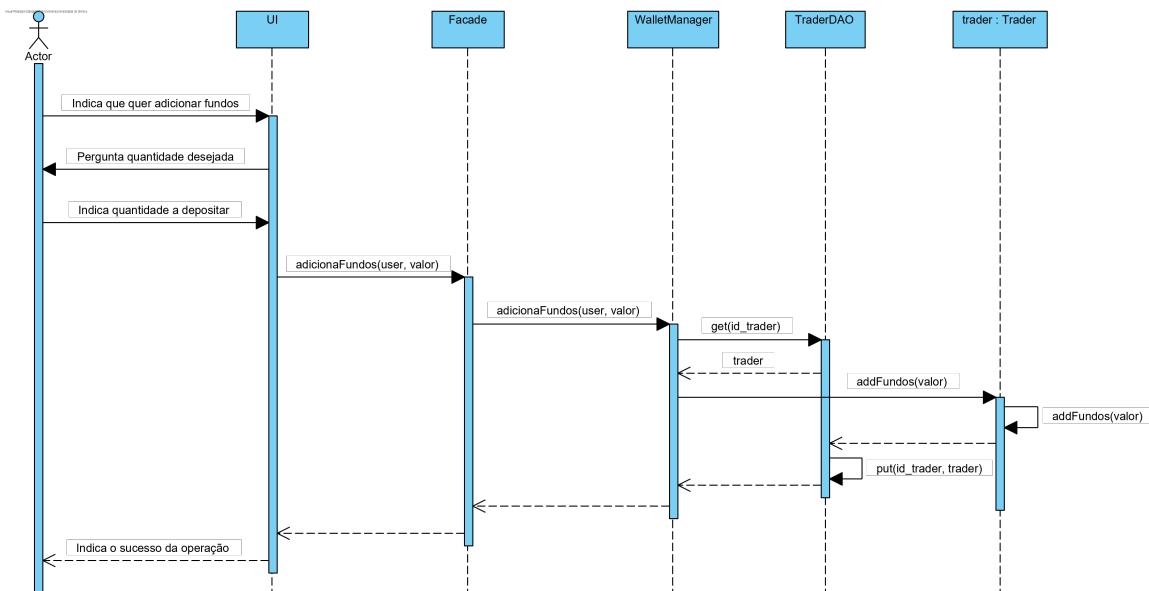


Figura 3: DS: Adicionar fundos

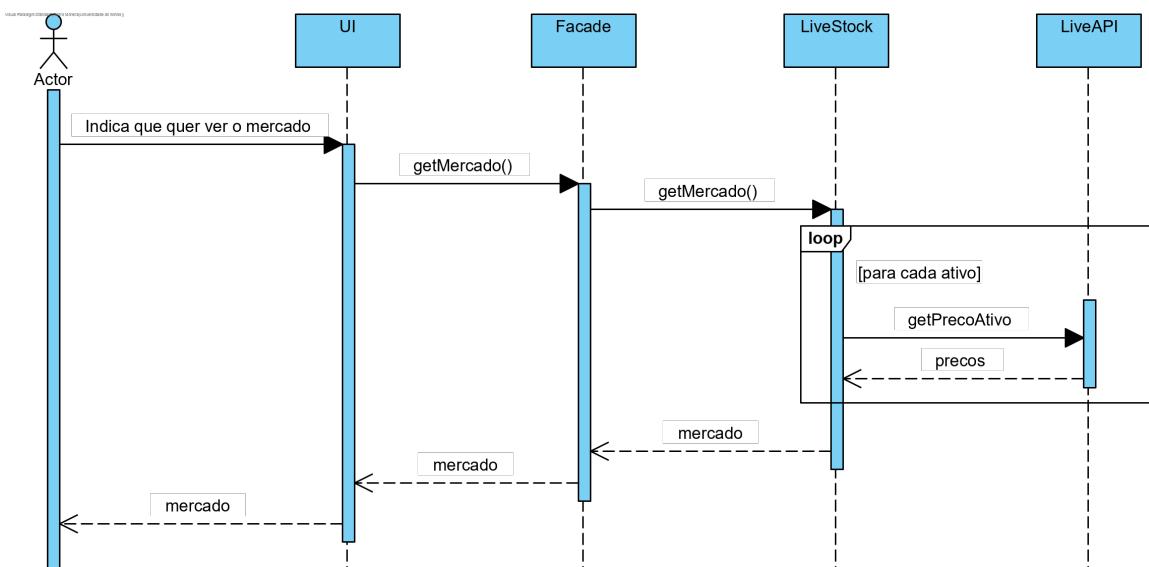


Figura 4: DS: Ver estado do mercado

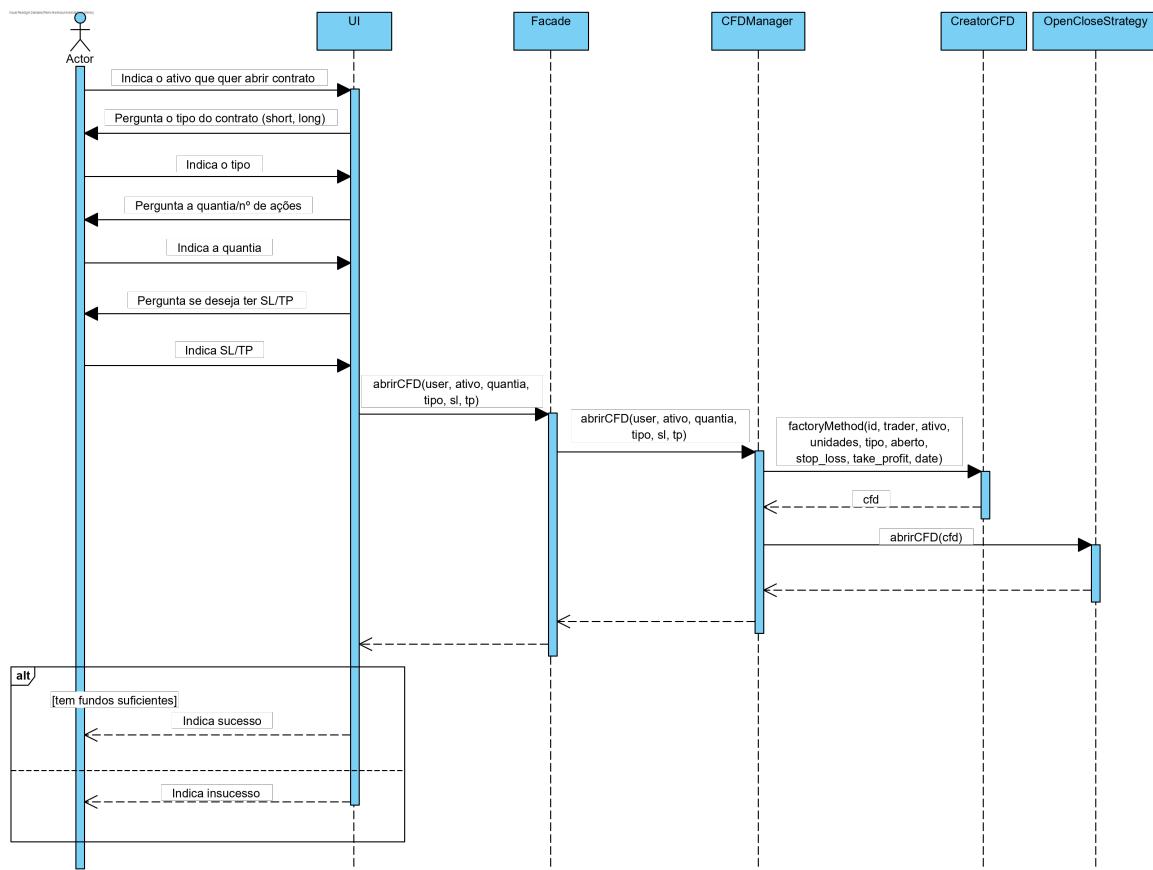


Figura 5: DS: Abrir CFD

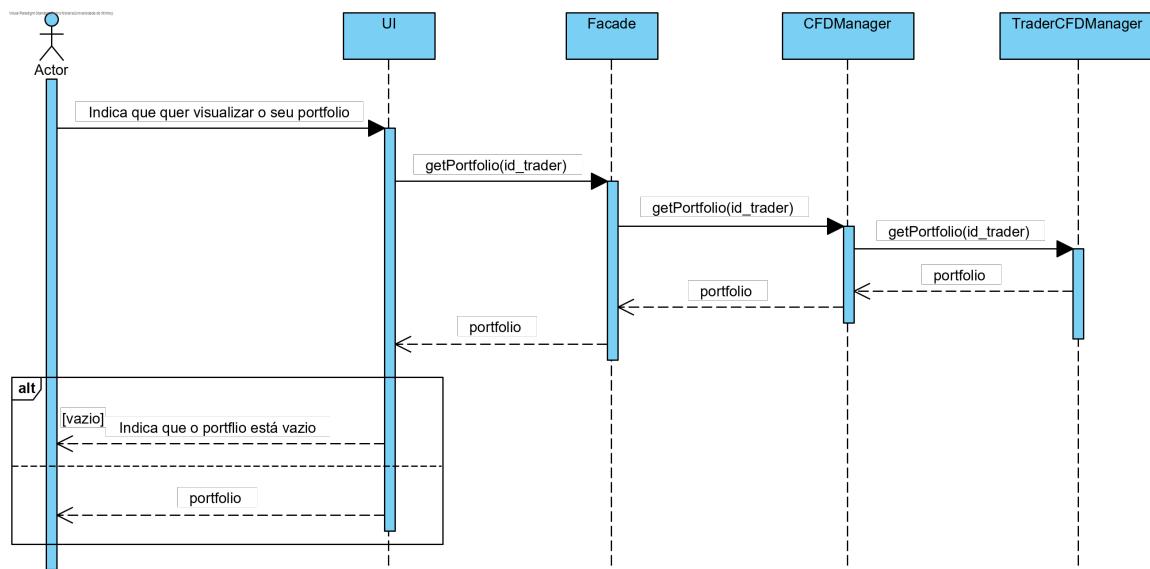


Figura 6: DS: Ver Portfolio

3.2.3 Diagrama de Atividade

Com o Diagrama de Atividade, da Figura 7, podemos observar a sequência de accções entre o utilizador (*Trader*) e o sistema, para **Abrir um CFD**.

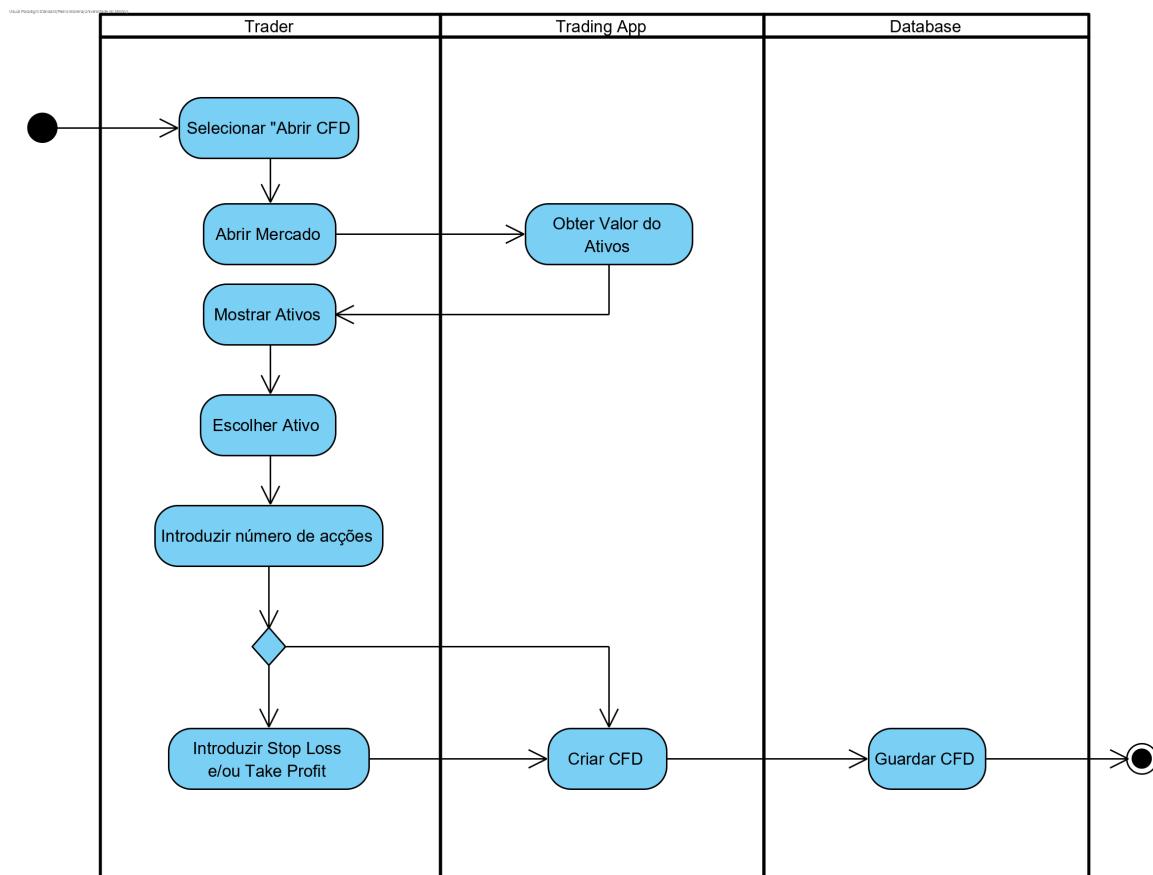


Figura 7: Diagrama de Atividade para a abertura dum CFD

3.2.4 Diagrama de Máquinas de Estado

Na Figura 8, podemos verificar as interações possíveis do utilizador com o sistema através do Diagrama de Máquina de Estados para a interface da aplicação.

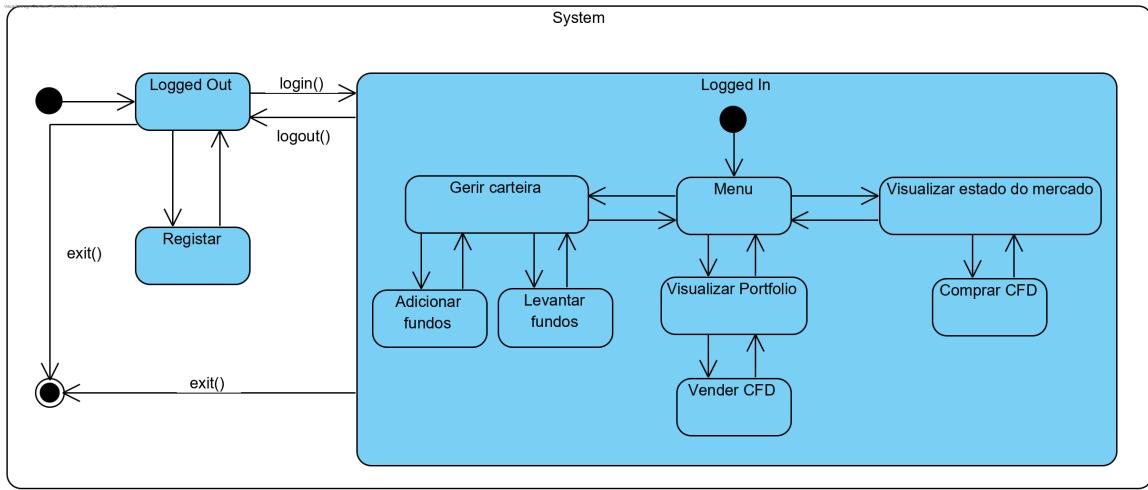


Figura 8: Máquina de Estados da interface do sistema

3.3 Atributos de qualidade

De seguida, apresentam-se os atributos de qualidade esperados para a aplicação:

- **Reusability**

Com a evolução das tecnologias pode surgir, por vezes, necessidade de fazer alterações em certas camadas da aplicação. Ora, se uma camada necessita de ser mudada, seria péssimo ter que mudar todas as outras. Com este atributo de qualidade garantimos que apenas é necessário mudar a camada pretendida, reutilizando as restantes camadas. Segue-se um exemplo deste atributo:

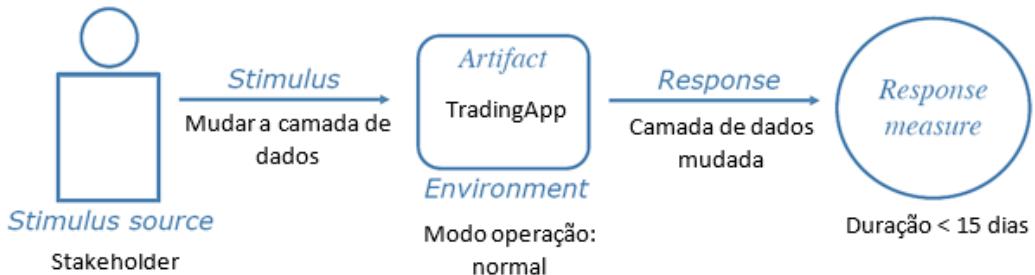


Figura 9: QA: Reusability

- **Extensability**

Aquando do momento do design da arquitetura não é possível prever todas

as possíveis situações novas que podem surgir e que levem a incluir novas funcionalidades no sistema. Devido a isto, o sistema deve suportar essas possíveis futuras alterações.

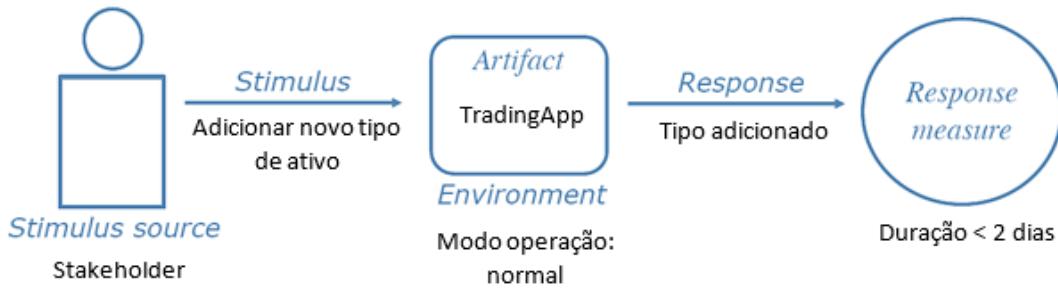


Figura 10: QA: Extensability

- **Modifiability**

Por fim, temos como um atributo de qualidade a *Modifiability* do sistema. Este é inicialmente concebido para ser usado num terminal, mas poderá ter que ser rapidamente adaptado para uma versão web, por exemplo.

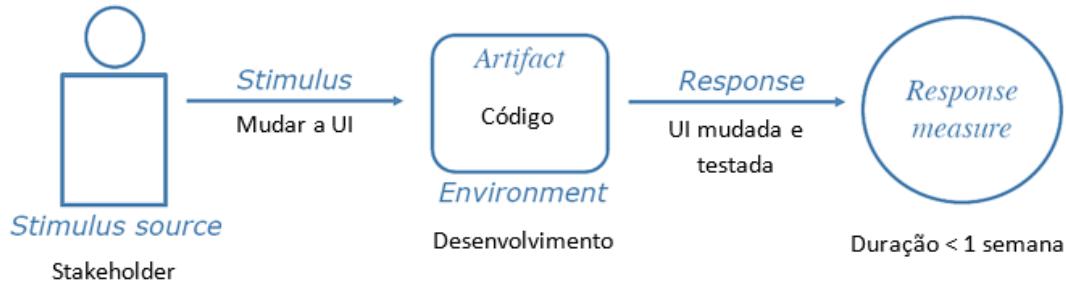


Figura 11: QA: Modifiability

3.4 Diagrama de Classes

Na Figura 12, podemos visualizar o estado interno da plataforma e como as classes interagem entre si, para assegurar o funcionamento esperado do sistema.

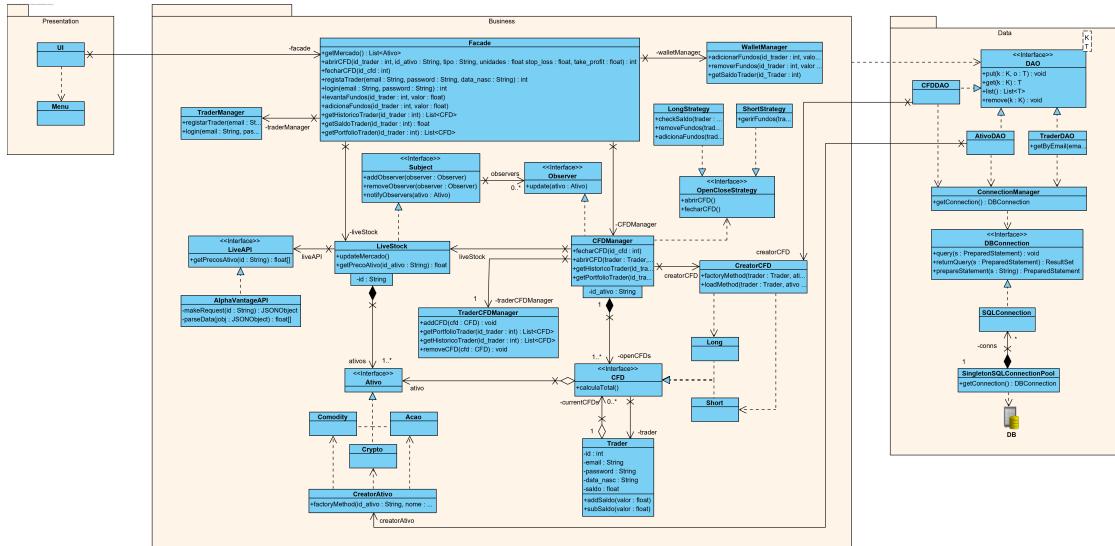


Figura 12: Diagrama de Classes

No diagrama estão, também, presentes três packages (*Presentation*, *Business*, *Data*). Pela sua disposição, é de fácil percepção que a arquitetura adotada é a arquitetura de camadas. Esta escolha deveu-se também aos atributos de qualidade descritos na Secção 3.3

A explicação em detalhe deste diagrama de classes é descrita na secção seguinte, juntamente com a explicação dos *design patterns* adotados, assim como outras estratégias úteis para o desenvolvimento do sistema.

3.5 Design Patterns e outras estratégias

3.5.1 Programar para interfaces, não implementações

Uma das boas técnicas da programação orientada ao objetos é programar para interfaces e não para implementações concretas.

Tendo isto em conta, foram criadas as interfaces Ativo e CFD, com o intuito de ter várias implementações de cada uma, para além de que esta técnica permite no futuro adicionar novas implementações sem mudar o restante código/arquitetura.

Ambas as interfaces têm métodos como *getters* e *setters* para obrigar as implementações a ter as variáveis de instância necessárias. Como implementações da interface **Ativo** temos as classes **Comodity**, **Acao** e **Crypto**. Como implementações da interface **CFD** temos as classes **Long** e **Short**.

Uma outra interface é a `LiveAPI`. Esta possui um único método, `getPrecosAtivo`, que tem como argumento o id do ativo na bolsa de valores. Retorna um *array* de dois *floats*, o preço de venda e o preço de compra do respetivo ativo. Deste modo, garantimos que no futuro podemos trocar a API à qual vamos buscar estes preços

sem que essa mudança seja vista pelo restante sistema. Como implementação desta interface, contamos com a classe `AlphaVantageAPI`, que utiliza a API com o mesmo nome.

3.5.2 Princípio da responsabilidade única

Numa primeira versão do projeto, existia uma “superclasse”, que detinha a responsabilidade para tudo o que a aplicação fazia. Nesta nova implementação, a referida classe foi substituída por classes mais pequenas, cada uma com um objetivo específico. De seguida, enumeram-se essas classes:

- `TraderManager` - responsável pelo registo e *login* dos utilizadores;
- `WalletManager` - responsável pela gestão dos fundos de um utilizador;
- `CFDManager` - responsável por gerir os *CFDs* (abertos) da aplicação;
- `LiveStock` - responsável pela gestão em “tempo real” dos preços dos vários ativos;
- `TraderCFDManager` - responsável pelo histórico e portfólio dum utilizador em particular.

3.5.3 Factory Method

Para abstrair a criação dos objetos que implementam as interfaces `Ativo` e `CFD` foram criadas, respetivamente, as classes `CreatorAtivo` e `CreatorCFD`. As implementações deste design pattern seguem-se de seguida:

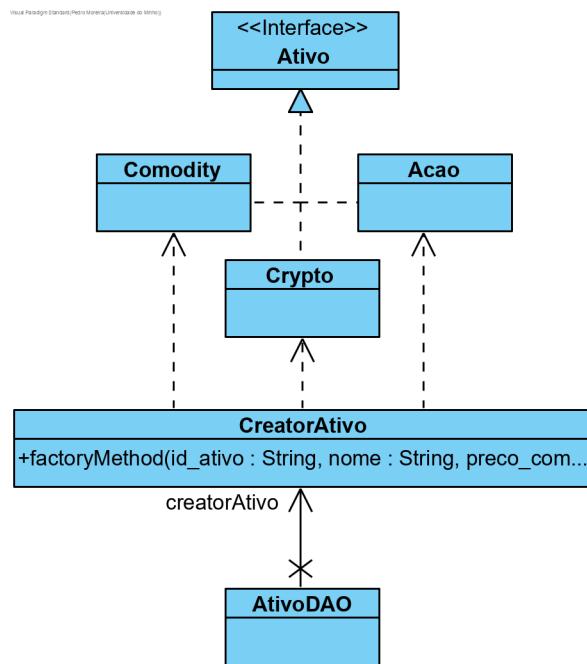


Figura 13: Factory Method (Ativo)

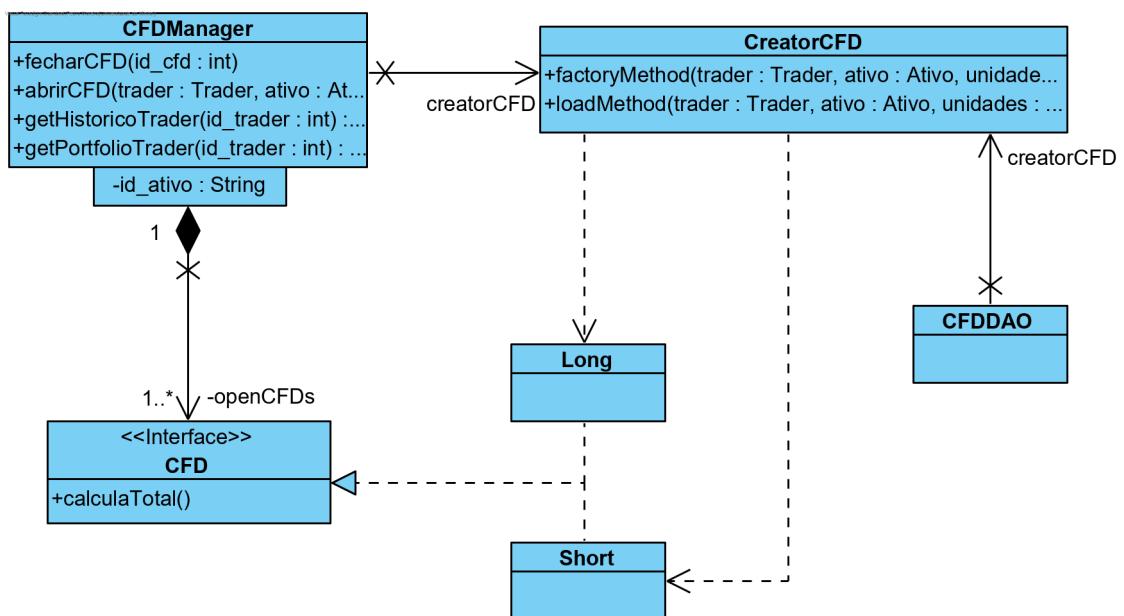


Figura 14: Factory Method (CFD)

De notar que nesta última implementação do padrão foi incluído um novo método, `loadMethod`, pois quando um *CFD* é carregado da base de dados, a sua

criação é realizada de forma diferente aquando da criação no momento da abertura do *CFD*.

3.5.4 Observer

Tratando-se da bolsa de valores, os preços das ações estão em constante mudança. O padrão *Observer* pode, assim, ser aplicado, para que seja possível os *CFDs* serem fechados automaticamente, tendo em conta os valores de *Stop Loss* e *Take Profit*, sem que o utilizador tenha que fazer algo para isso.

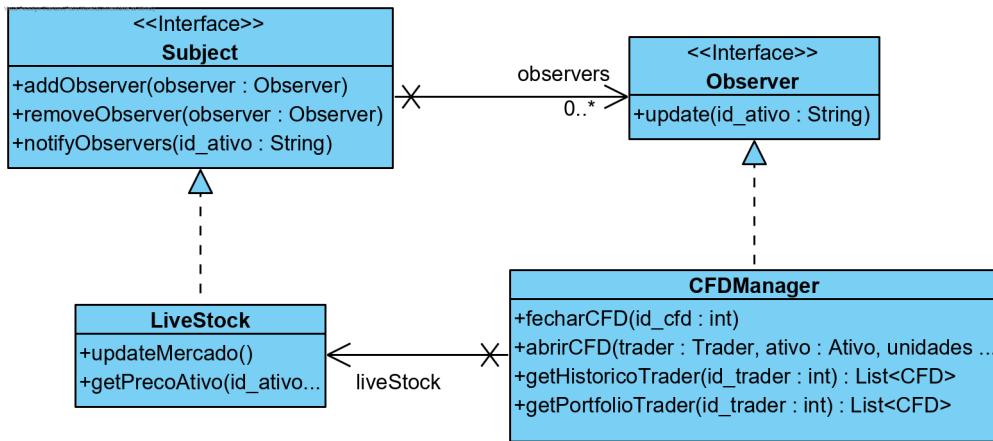


Figura 15: Observer

Sempre que os preços dum ativo são atualizados, o *CFDManager*, que possui um Map `<String, List <CFD>>` – um mapa cuja chave é o id do ativo e o valor é a lista dos *CFDs* abertos relativos a essa ativo –, percorre a lista dos *CFDs* abertos desse ativo e verifica se estão reunidas as condições necessárias para o seu fecho. Caso isso se verifique, os respetivos *CFDs* são fechados e o *trader* recebe o saldo correspondente.

(Nota: Nesta implementação da classe *LiveStock* decidimos atualizar o mercado com uma frequência de 75 segundos, devido a limitações da versão gratuita da API utilizada.)

3.5.5 Strategy

O modo como é realizada a abertura dum *CFD* do tipo *Long* é diferente da abertura do tipo *Short*, assim como aquando do fecho. O grupo interpretou o processo da seguinte maneira: quando um *Long* é aberto, o *trader* fica imediatamente sem o valor respetivo à compra, recuperando (ou não) o investimento aquando do fecho do *CFD*; quando um *Short* é aberto, o *trader* necessita apenas de ter saldo superior a 0, não ficando sem o valor da abertura. O ganho/perda é calculado aquando do fecho

do *CFD*, fazendo a diferença entre o preço atual e o preço na altura da abertura. Outra diferença entre estes dois tipos de *CFD* é que o valor utilizado para calcular o valor total é diferente: *Long* utiliza o preço de compra do ativo, enquanto que o *Short* usa o preço de venda.

Deste modo, ao aplicarmos o padrão *Strategy* permitimos que a abertura e o fecho de *CFDs* tenham uma API comum, mas o modo como são feitos varia consoante o seu tipo.

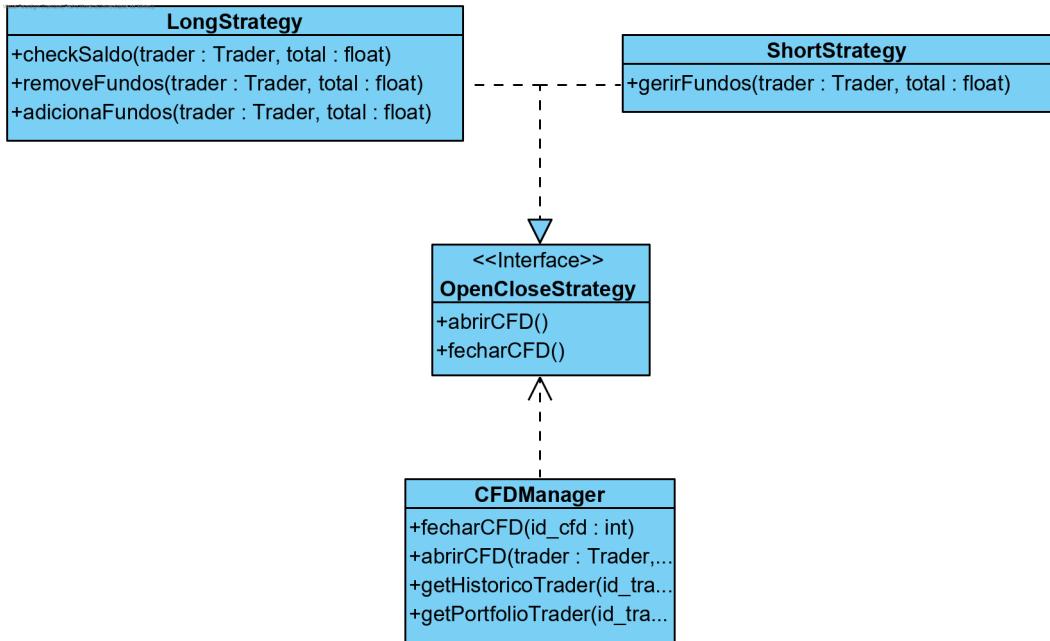


Figura 16: Strategy

3.5.6 Facade

De modo a que a UI do sistema não tenha conhecimento de tudo o que está na camada de Business, o padrão **Facade** permite que a UI apenas conheça uma pequena parte da implementação. Por outras palavras, com este design pattern fornecemos à UI uma API reduzida para que esta possa apresentar as informações ao utilizador.

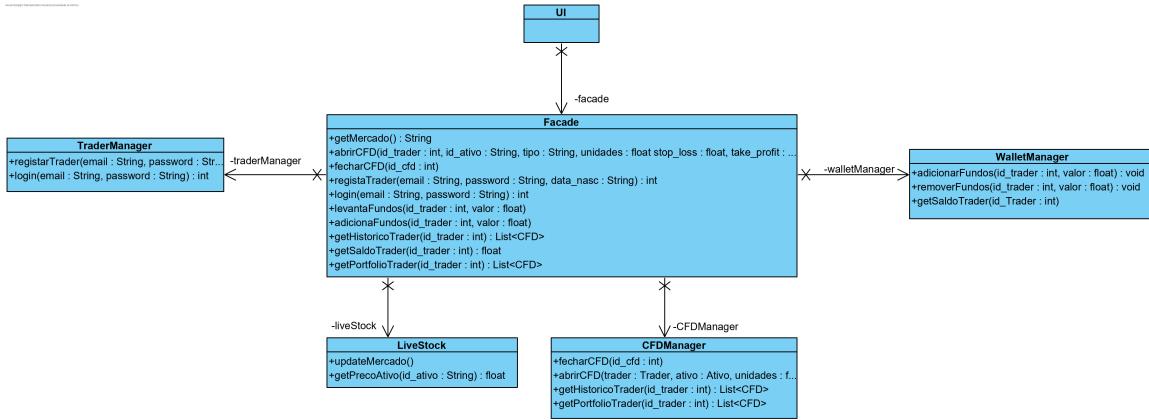


Figura 17: Facade

Como podemos ver pela Figura 17, a classe **Facade** tem uma instância das classes enumeradas na Secção 3.5.2. Deste modo, o *facade* pode realizar as instruções dadas pelo utilizador, sem expor mais do que necessário.

3.5.7 DAO

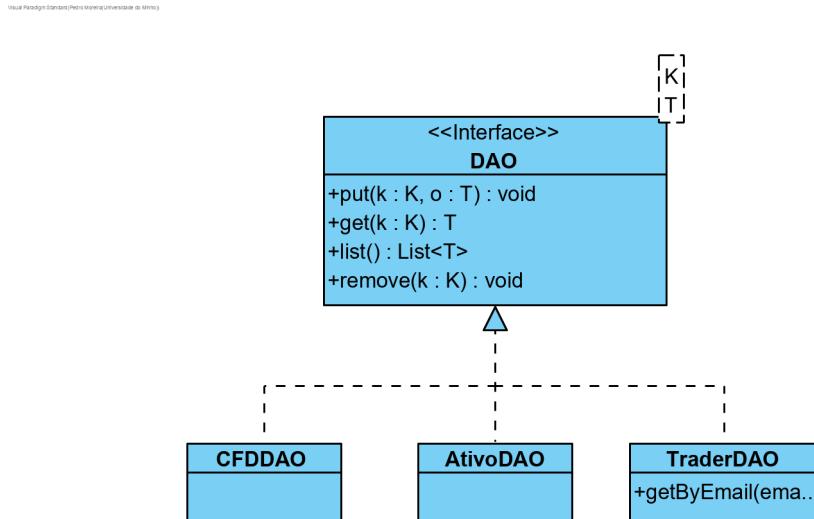


Figura 18: DAO

Para isolar o acesso aos dados pela camada *Business* optamos por aplicar o padrão *DAO*, que permite encapsular as implementações das operações na base de dados, expondo apenas os respetivos métodos. Para além disso, usando uma interface comum a todos os *DAOs* permite ao código da camada *Business* invocar os *DAOs* sem conhecimento de outros métodos que implementações de *DAO* possam ter.

Os *DAOs* tornam-se responsáveis por aceder aos dados, de modo que os módulos na camada *Business* não tenham essa responsabilidade, respeitando o *Single Responsibility Principle*.

3.5.8 Singleton e Database Connection Pool

De modo a distribuir conexões à base de dados, decidimos usar uma *Connection Pool* que encapsula as conexões na classe `SQLConnection` com o intuito de abstrair o máximo possível da implementação dos acessos à BD.

Conseguimos realizar isto fazendo a *Connection Pool* retornar a interface `DBConnection`, que possui apenas os métodos necessários.

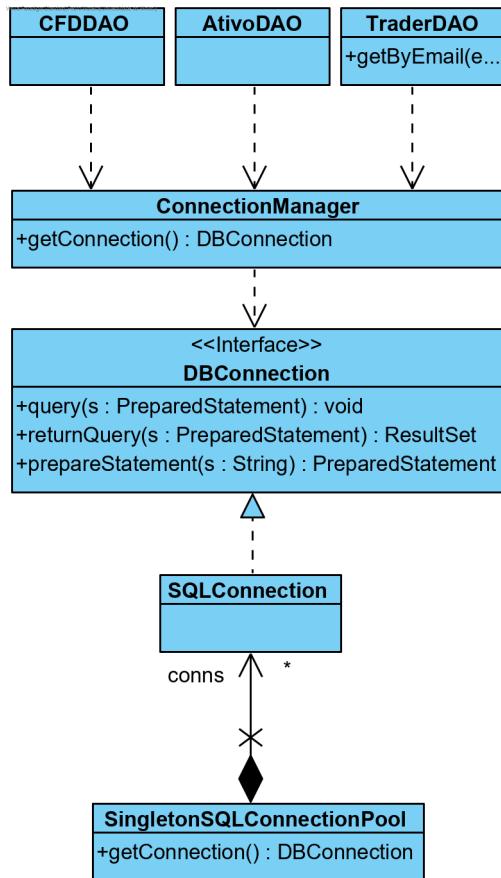


Figura 19: Singleton & Connection Pool

Assim, cada *DAO* apenas necessita de obter uma conexão através do `ConnectionManager` e está pronto para executar as *queries* que desejar.

Visto só fazer sentido haver uma *Connection Pool* para gerir as conexões, optamos por aplicar o padrão **Singleton** para garantir que só existe uma instância

desta classe.

De notar que, ao abstrair a implementação das conexões à base de dados, os objectos que a usam não têm conhecimento dos detalhes da ligação, tais como as credenciais ou o endereço, facilitando assim qualquer possível modificação do método de acesso à BD.

3.6 Modelo Lógico da Base de Dados

Para persistência de dados decidimos usar uma base de dados MySQL com recurso a 3 tabelas, **trader**, **cfд** e **ativo**.

A tabela **trader** está responsável por armazenar as informações de todos os traders registados no nosso sistema, como, por exemplo, o seu saldo.

A tabela **cfд** contém todos os *CFDs*, abertos ou fechados, criados pelo nosso sistema e associa-os ao *trader* responsável e ao ativo em questão, através de chaves estrangeiras referentes às outras 2 tabelas.

A tabela **ativo** enumera os diferentes ativos suportados pela nossa aplicação guardando o seu id – que é necessário para a obtenção dos valores de compra e venda através da API –, o seu nome e o tipo de ativo em questão.

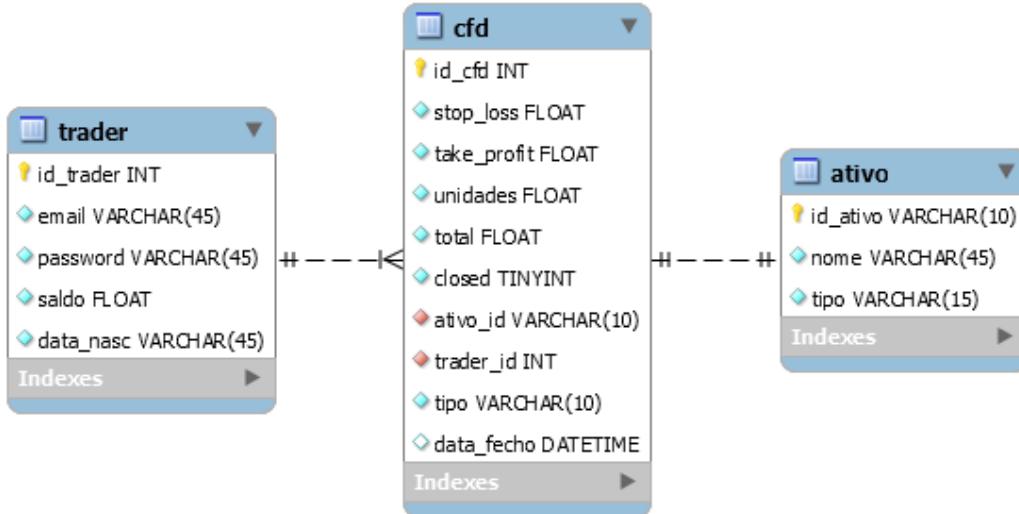


Figura 20: Modelo lógico da base de dados

4 Implementação do novo requisito

Este novo requisito consistia na implementação de uma *watchlist* de ativos, ou seja, ter ativos como espécie de favoritos, para que fosse possível receber notificações quando a variação do seu preço fosse relativamente alta.

Para que isto fosse possível, era preciso estar “de olho” nos preços dos ativos. Deste modo, e como já tinha sido implementado para outra funcionalidade do sistema, o padrão **Observer** é o ideal para implementar nesta situação.

Conseguimos aproveitar o padrão já implementado mudando apenas o argumento do método `notifyObservers` que, em vez de ser o id do ativo, passou a ser o ativo em si. Porém, esta alteração não foi uma necessidade, mas sim um melhoramento, relativamente à versão anterior, pois facilitou o acesso aos preços necessários para notificar ou não os respetivos *observers*.

Como para este padrão é necessário um observador, criámos uma nova classe, `NotificationManager`, que implementa a interface `Observer`. Na figura seguinte podemos verificar o resultado dessa alteração.

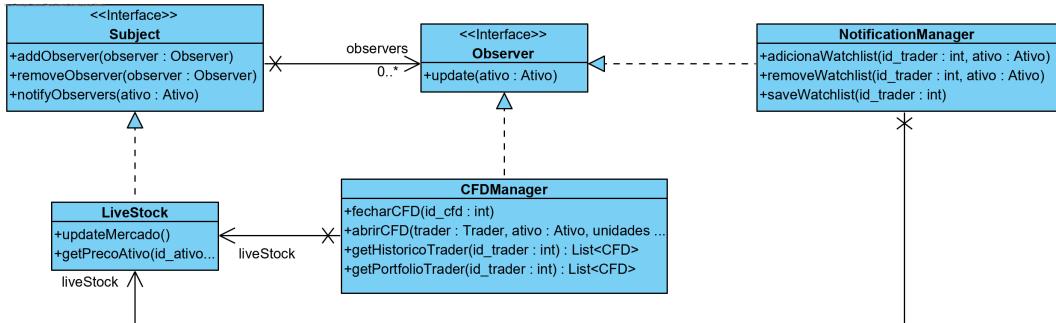


Figura 21: Observer

Esta nova classe conta com uma lista dos ativos que o utilizador que se encontra logado no momento tem na sua *watchlist*. Sempre que o utilizador realiza o *logoff*, esta lista é reiniciada.

Para que fosse possível ser a UI a classe responsável pela visualização da notificação, foi necessário incluir na nova classe uma referência para o Facade. Foi também necessário tornar o Facade bidirecional com a UI, ou seja, incluir uma referência da UI no Facade. Deste modo, conseguimos que as tarefas permanecessem devidamente separadas pelos respetivos módulos, apesar de aumentar o acoplamento entre as classes.

Na classe UI foram feitas alterações para que fosse possível adicionar e remover ativos à *watchlist*, assim como na base de dados para que fosse possível guardar a *watchlist* de cada utilizador. Esta última alteração pode ser vista na figura seguinte.

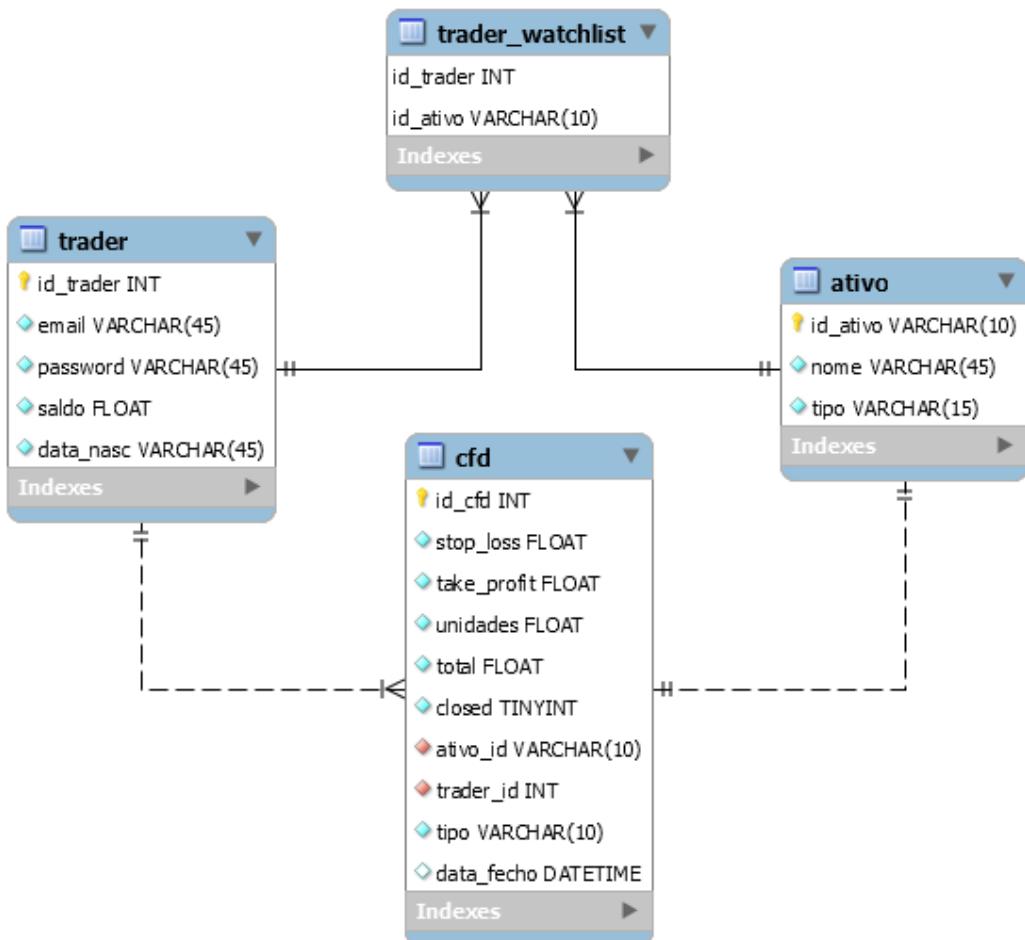


Figura 22: Novo modelo lógico da base de dados

Para concluir este capítulo, segue-se o Diagrama de Classes final do projeto.

Online Trading Platform

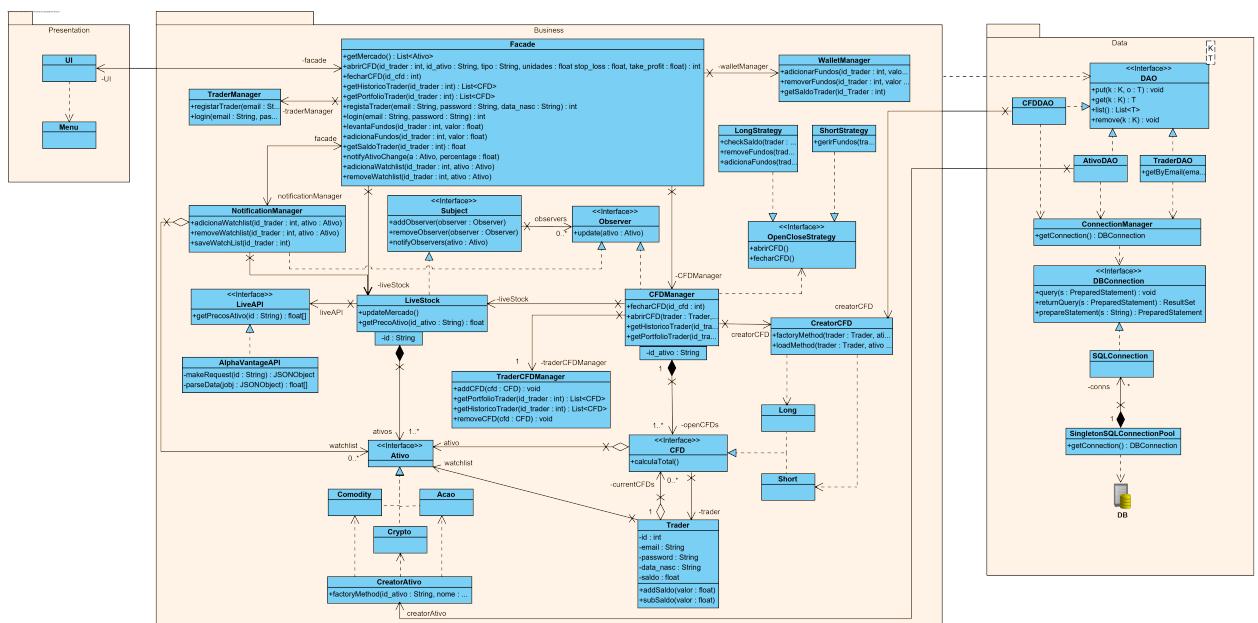


Figura 23: Diagrama de Classes final

5 Conclusão

Com este trabalho foi possível ao grupo aplicar os conhecimentos adquiridos ao longo da unidade curricular, nomeadamente *Design Decisions*, *Design Patterns* e *Architectural Patterns*, assim como outras boas práticas no contexto de POO.

Comparando com a 1^a Fase deste projeto, podemos verificar que a arquitetura da aplicação sofreu uma mudança drástica, transformando-se duma superclasse para uma implementação com várias classes, cada uma com a sua função bem definida, interfaces e *design patterns*.

A nova arquitetura também permite que sejam facilmente adicionadas ou removidas algumas peças do sistema devido à programação orientada à interface que o grupo levou a cabo. Um exemplo disto foi a incorporação, como um pequeno extra, da classe *Crypto*, uma implementação da interface *Ativo*, que representa as criptomoedas na bolsa.

Apesar do número de classes ser um valor considerável, devendo-se ao facto do grupo ter tentado ao máximo aplicar os *Design Patterns* **apropriados**, o resultado final ainda é, na opinião do grupo, facilmente comprehensível por outros. Este facto leva a que alterações no código possam ser feitas facilmente por developers que não participaram no desenvolvimento da arquitetura.

Contudo, o desenvolvimento do projeto, mais precisamente da arquitetura resultado, foi de elevada dificuldade, pois o grupo não tinha qualquer experiência no âmbito do desafio, levando a um constante pensamento “Não estaremos a complicar demasiado isto?”.

Posto tudo isto e para terminar, o grupo faz uma auto-avaliação positiva da produto final.

Appendices

Apêndice com as imagens relativas à primeira fase do projeto.

A Atributos de Qualidade

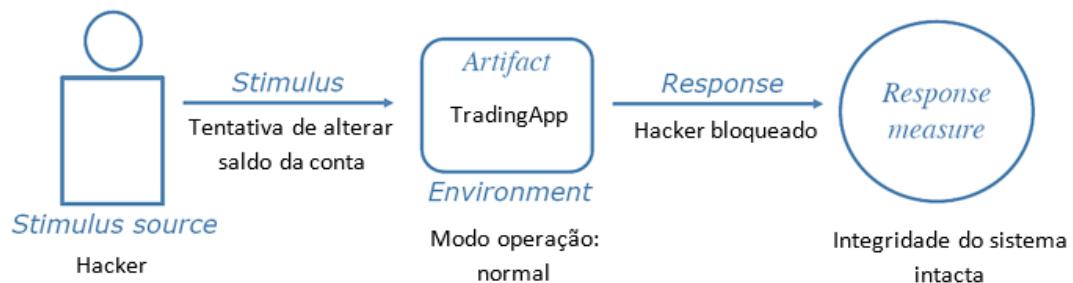


Figura 24: QA: Security

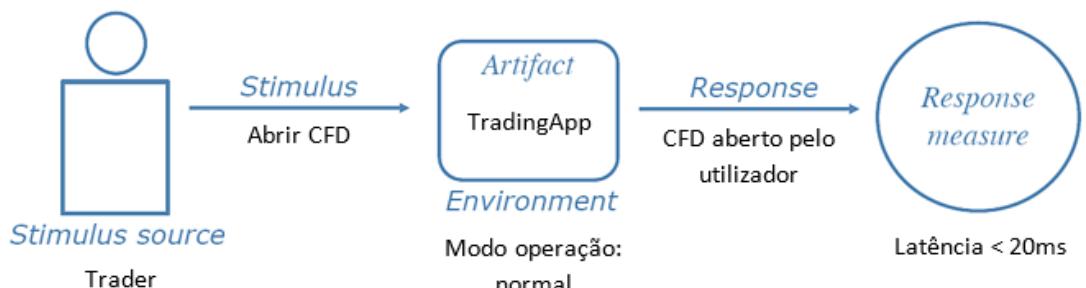


Figura 25: QA: Performance (1)

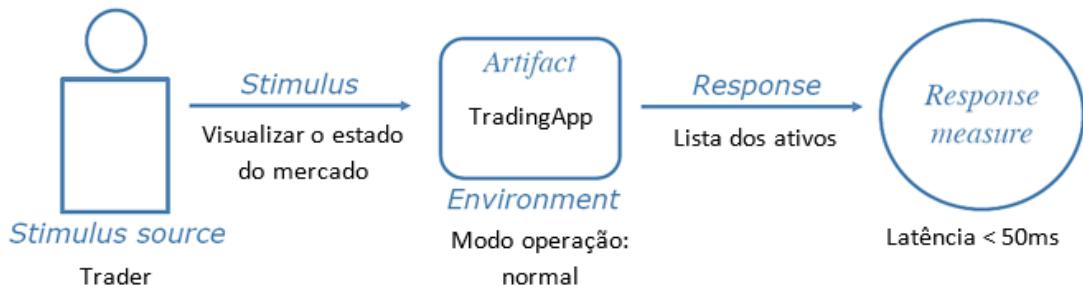


Figura 26: QA: Performance (2)

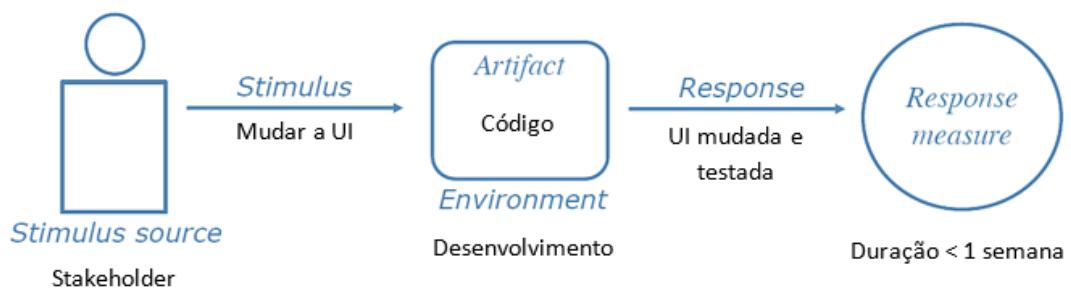


Figura 27: QA: Modifiability

B Diagramas de Sequência

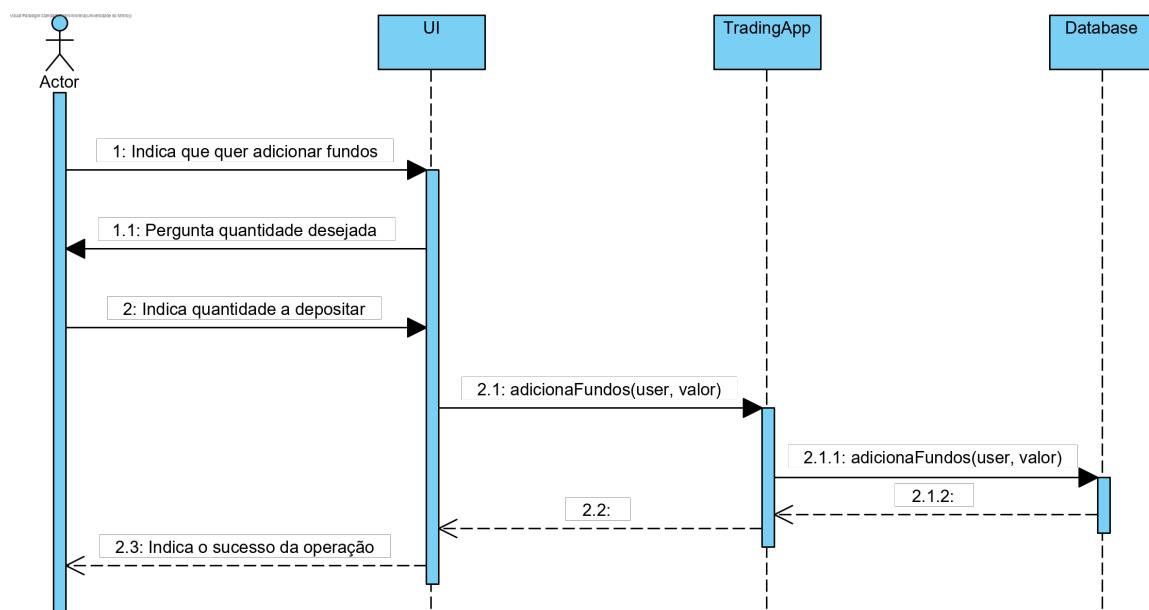


Figura 28: DS: Adicionar fundos

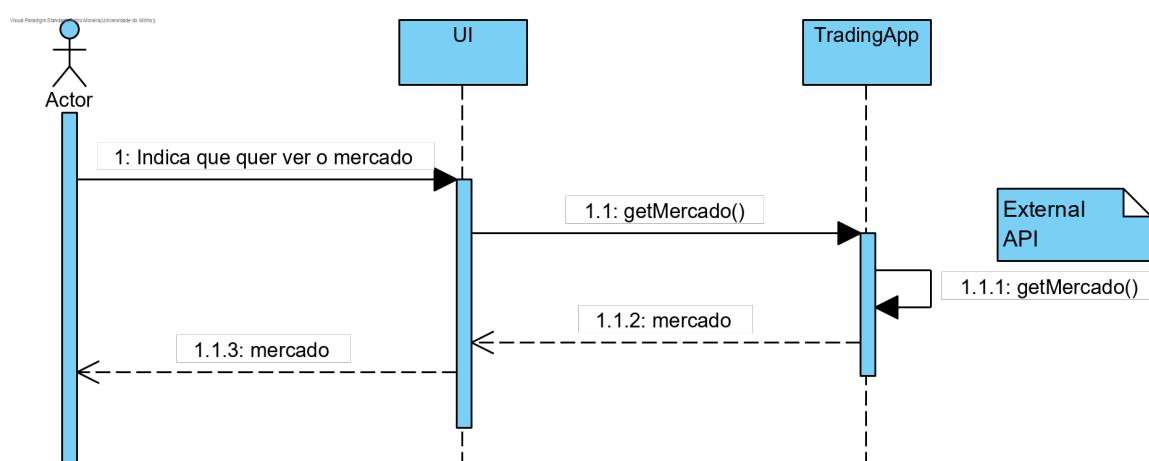


Figura 29: DS: Ver estado do mercado

Online Trading Platform

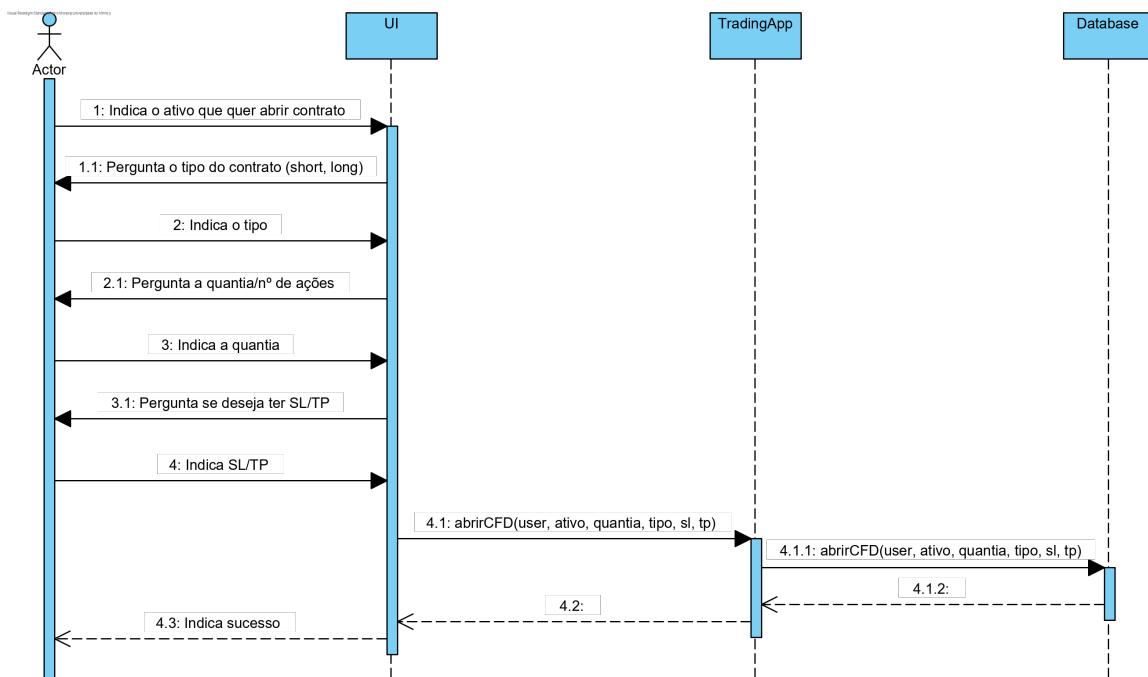


Figura 30: DS: Abrir CFD

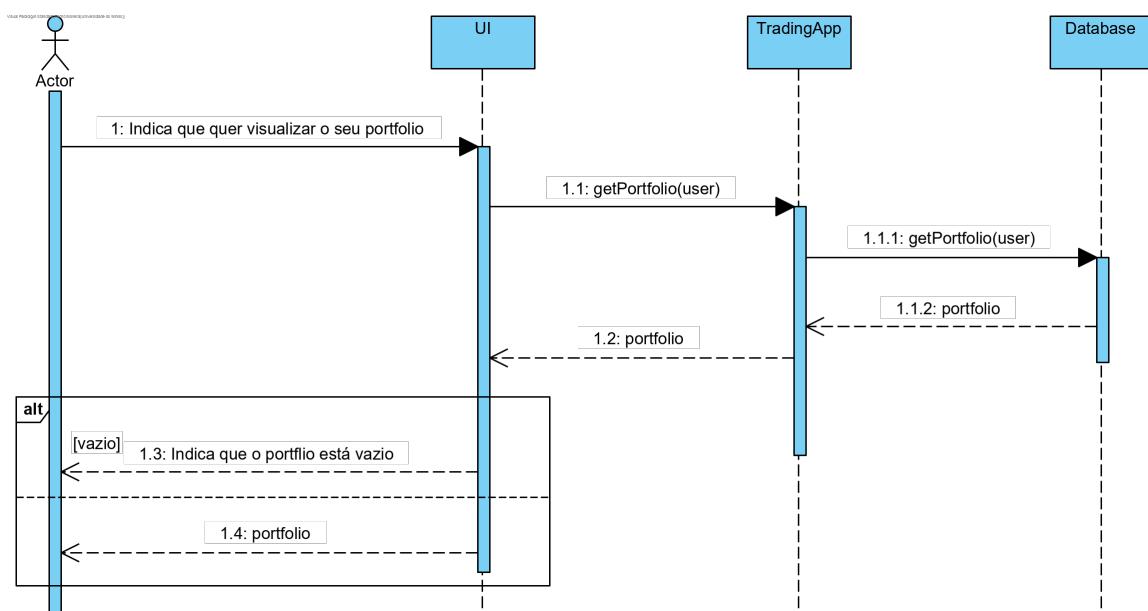


Figura 31: DS: Ver Portfolio

C Diagrama de Classes

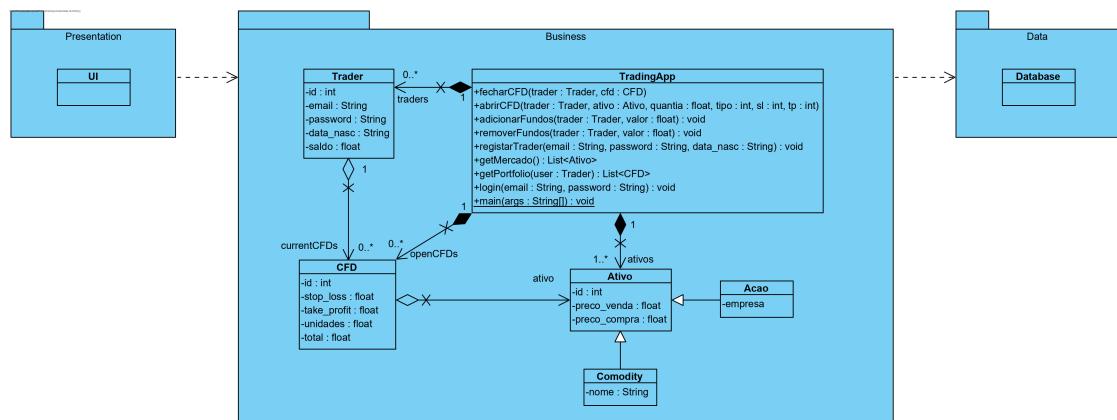


Figura 32: Diagrama de Classes