Universidade do Minho
Escola de Engenharia

MESTRADO EM
ENGENHARIA INFORMÁTICA
1º ANO | 1º SEMESTRE

ARQUITETURAS DE SOFTWARE

Relatório de Projeto 1º Fase

Francisco Saraiva | PG39287

Ano Letivo 2019/2020
Braga, 30 de Novembro 2019

Conteúdos

Ilustrações

Introdução

Este documento serve de suporte e detalha a análise efetuada e os aspetos técnicos de implementação da aplicação **ESS Trading Platform**, para a componente curricular de projeto da cadeira de arquiteturas de software.

Nesta entrega da 1º fase foi proposta arranjar uma solução para o problema de oferecer uma plataforma de negociação para permitir a investidores ou traders de gerir posições no mercado financeiro por CFDs (*Contract For Differences*). Elaboração passou por uma análise e levantamento de funcionalidades para a plataforma, desenvolvimento de mockups e visualização das várias funcionalidades, identificar 3 a 4 funcionalidades principais, atributos de qualidade e condicionantes e a elaboração de diagramas para representar a estrutura da aplicação.

1. Contexto do problema

A ESS Ltd, pretende criar um produto para negociação de CFDs através de uma plataforma de negociação onde utilizadores (investidores e traders) abrem, fecham e gerem posições no mercado financeiro onde podem de vários ativos financeiros escolher para ingressar na compra e venda de CFDs, dos quais acções, *commodities* como ouro e petróleo, índices ou até moedas.

Estas plataformas geralmente possuem corretores financeiros que tratam da transação, gratuitamente ou com uma taxa aplicável a um número de posições mínimas por mês. Nesta plataforma pretende-se puder aceder a uma lista de variados ativos financeiros e dar a possibilidade ao utilizador de comprar ou vender ativos atráves de contratos de diferença, denominados de CFDs. Numa posição em que o utilizador pretende **comprar** um ativo (pelo preço base + margem de corretor) quando sobe, irá ser criado um CFD de **long** com intuito de vir a lucrar com a subida de preço do ativo, pelo inverso quando um ativo vir a descer de preço o utilizador irá querer lucrar **vendendo** o ativo com um CFD de **short**. Em CFDs o intuito de lucrar passa pela diferença do preço base em iniciou o contrato com a oscilação do preço do ativo no mercado, em que quanto maior a diferença maior o lucro. Os utilizadores podem também num CFD escolher uma quantidade para ter maior aproveitamento do investimento no ativo.

Os requisitos/funcionalidades propostas do projeto são:

- Manter valores dos ativos a ser negociados via CFDs;
- Registo de contas na plataforma com um *plafond* inicial;
- Utilizadores podem abrir CFDs sobre ativos disponíveis, de compra e venda;
- CFDs estão ligados a serem definidos com limites de ganho e de perda estipulados pelos utilizadores (*Take profit* e *Stop loss*);
- CFDs podem ser fechados a qualquer momento por acção dos utilizadores;
- Utilizadores podem monitorizar em tempo real os seus portfolios de CFDs e visualizar o valor atual dos ativos adquiridos

Com base na análise foi escolhido desenvolver este projeto numa plataforma web para dar aos utilizadores as funcionalidades pretendidas.

2. Tecnologias

Para o desenvolvimento deste projeto, foi decidido criar uma aplicação por linha de comandos em **Node.js** com o intuito de oferecer as funcionalidades de gestão de CFDs. A linguagem de programação utilizada é **Typescript**, uma linguagem open source da família do **Javascript**, que oferece um paradigma orientado a objetos, com classes e tipagem estrita nos atributos das entidades ao contrário de Javascript.

Sendo o projeto desenvolvido em Node.js, que podem ser consultados há vários pacotes necessários ao funcionamento do mesmo que foram incorporados para a solução:

- **Typescript**, para traduzir o código orientado a objetos para Javascript;
- **Boxen**, para display de caixas decorativas no terminal;
- **Clear**, para limpar o terminal;
- **Cli-table**, para auxiliar na criação de tabelas de apresentação no terminal;
- **Figlet**, para desenhar palavras no terminal;
- **Inquirer**, para apoio nas seleções e inputs do utilizador no terminal;
- **MySQL**, para apoio nas operações e comunicação com a base de dados;
- **TypeORM**, um ORM para apoio na comunicação e gestão entre a aplicação e a base de dados;
- **Stock-info**, para apoio nos pedidos à api da Yahoo Finance para obter valores para os ativos.

3. Análise

3.1. Domínio

Primeiro passo para analisar o problema é tirar dos requisitos e funcionalidades já colocadas as várias entidades que pertencem no domínio. Temos **Utilizadores**, que criam **CFDs** de **Compra** ou de **Venda** entre **Ativos Financeiros** dos quais são de vários **Tipos de Ativos**.

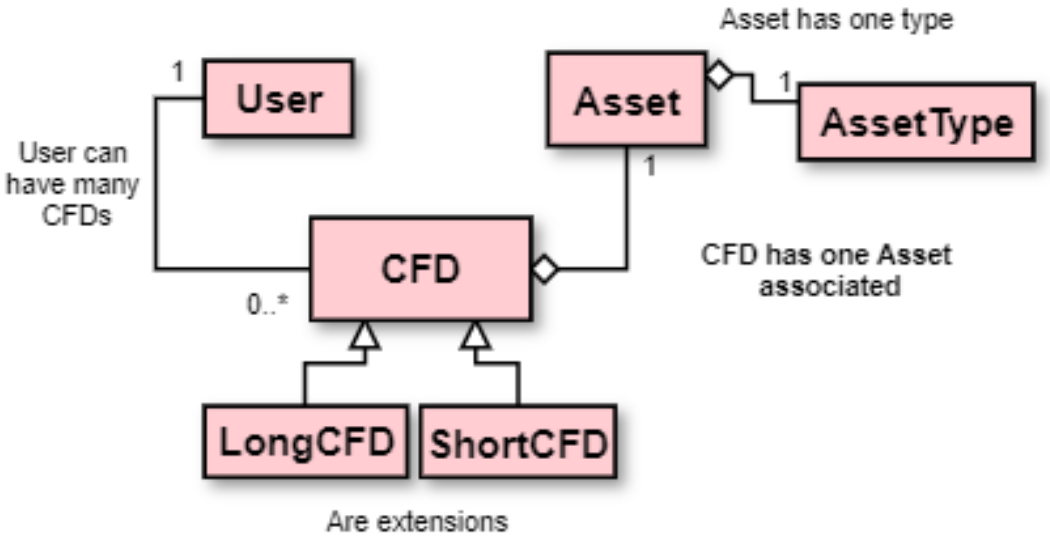

Figura 1- Diagrama de Domínio

A plataforma irá ser composta por utilizadores que interagem com os ativos criando CFDs, assim um utilizador pode criar vários contratos de diferença, que por si estão associados a um Ativo, que podem ser de vários tipos (acções, ouro, moedas…). Dentro dos CFDs existem dois tipos, o Long e o Short que são generalizações do CFD.

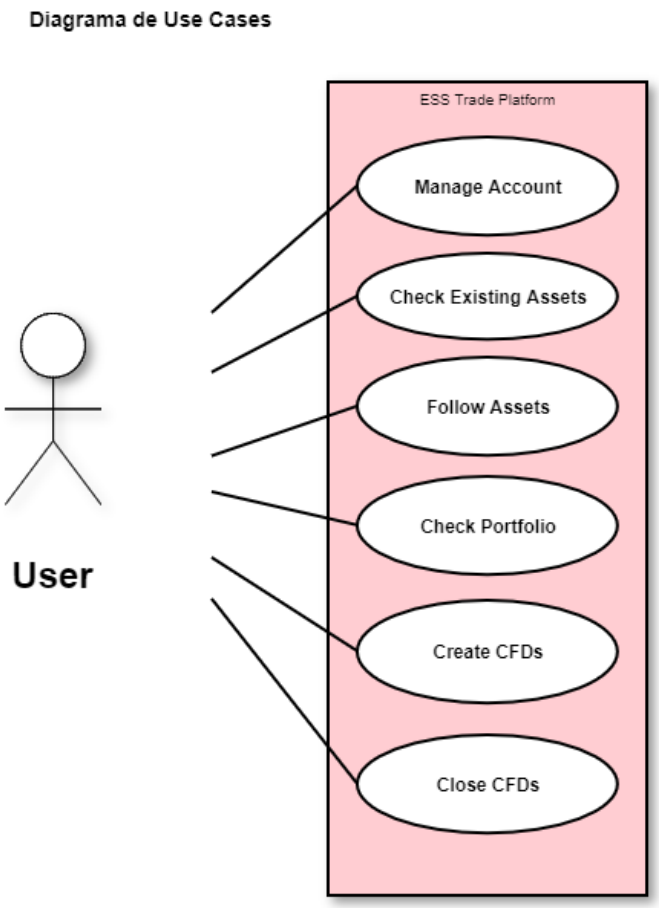

Figura 2- Diagrama de Use Cases

As funcionalidades passam por um sistema de autenticação que faz o login e o registo de novos utilizadores e sua edição, verificar os ativos existentes e as suas flutuações no mercado, seguir ativos, gerir CFDs criando e fechando para os vários ativos existentes.

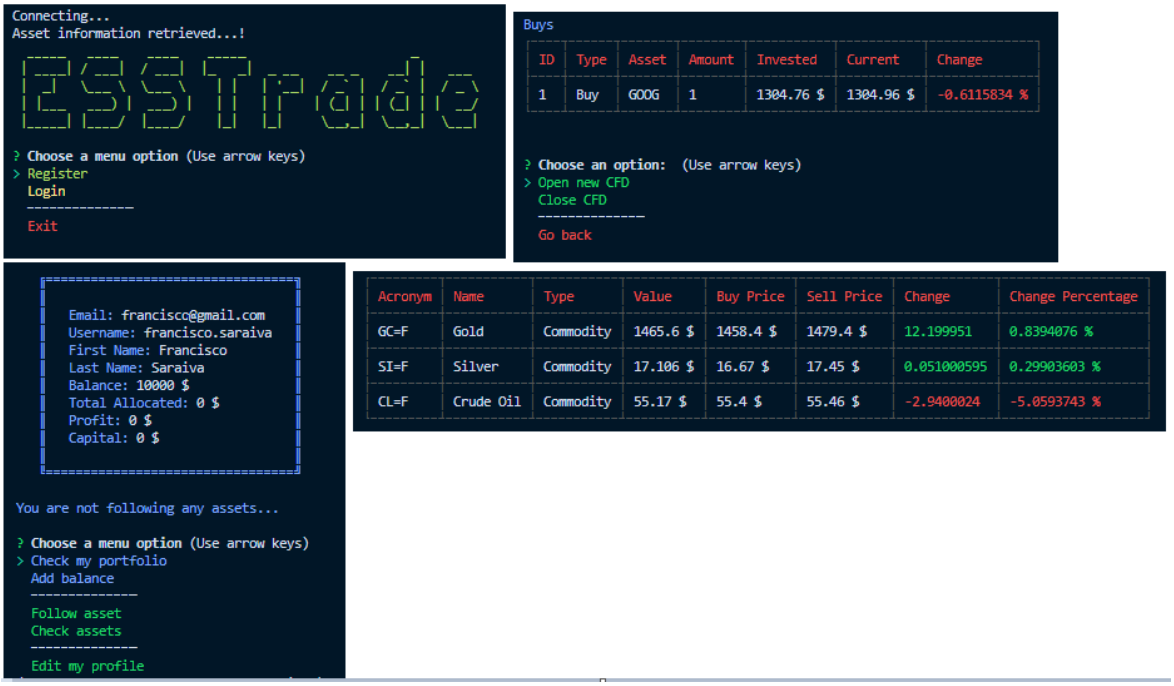

Figura 3 - Mockups da aplicação

Aspeto visual da aplicação no terminal de consola, em que os menus são navegados com as setas do computador e informação apresentada em caixas ou tabelas.

3.2. Arquitetura

3.2.1. Padrão arquitetural

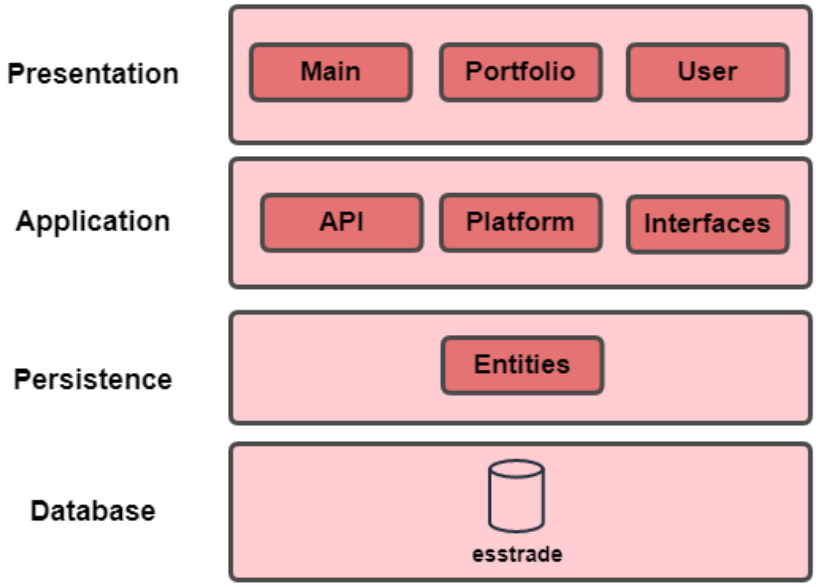

Figura 4 - Padrão Arquitetural

O padrão utilizado e o que foi implementado nesta aplicação é uma arquitetura em camadas em que cada camada interage com as debaixo tendo pouco ou nenhum conhecimento das camadas.

3.2.1.1. Presentation

Na primeira camada de apresentação existem os componentes responsáveis pela interação com o utilizador e apresentação do conteúdo para o terminal, em que cada componente: **main**, **portfolio** e **user**, contém um conjunto de funções auxiliares.

Main (funções para menu principal da aplicação):

- MainMenu
- Login
- Register

User (funções do menu de utilizador)

- AddBalance
- CheckAssets
- EditProfile
- FollowAsset
- GetPortfolio
- LoggedMenu

Portfolio (funções do portfolio do utilizador)

- CFDMenu
- CloseCFD
- OpenCFD

As funções de apresentação seguem um esquema em cascata de onde saltam entre si conforme as escolhas nos menus do utilizador. As funções de apresentação seguem o seguinte esquema:

FunctioName(clear, data)

Em que **FunctionName** é o nome da função do menu, **clear** um booleano que indica se é para limpar o ecrã de informação anterior e **data** a informação a passar para esse método para se puder trabalhar com a informação passada.

De entre os três conjuntos, **Main** a **data** que é passada são os **assets**(ativos) criados no início da aplicação e nos conjuntos **User** e **Portfolio** a **data** é uma instância da classe **Trader** de onde se pode aceder a todas as classes e métodos.

3.2.1.2. Application

Segunda camada de aplicação contém os componentes responsáveis pela aplicação ao problema e funcionamento do mesmo.

API é onde a classe **ApiResponse** está, em que se converte a resposta obtida da API da Yahoo para um objeto legível e interpretado para atualizar os valores dos ativos, é chamada na sincronização da API com a atualização dos ativos.

Platform é onde a classe **Trader** está, classe central ao funcionamento da aplicação, funcionada como um **Facade** para se puder aceder as todos os métodos necessários, desde o utilizador logado, os ativos existentes e os CFDs do utilizador.

Interfaces são onde as interfaces da aplicação estão guardadas, nomeadamente o Observer e Subject.

3.2.1.3. Persistence

A terceira camada de persistência é responsável pelo componente das entities, que são as classes do domínio e responsáveis pelas ações onde se efetuam persistência dos dados na base de dados, estendendo de uma classe auxiliar do pacote **TypeORM** (documentação aqui).

Com as classes/entidades/modelos do nosso problema/aplicação, o pacote providencia variadas ferramentas que ajudam na persistência das tabelas na base de dados. Estendendo as classes da aplicação à classe do pacote **BaseEntity** temos as classes como representações de modelos a criar na base de dados, através de anotações nas variáveis e como se relacionam entre as várias classes do sistema. Assim podemos criar instâncias de objetos que fácilmente devido à extensão da classe **BaseEntity** podemos gravar registo na base de dados através do método **save()**.

3.2.1.4. Database

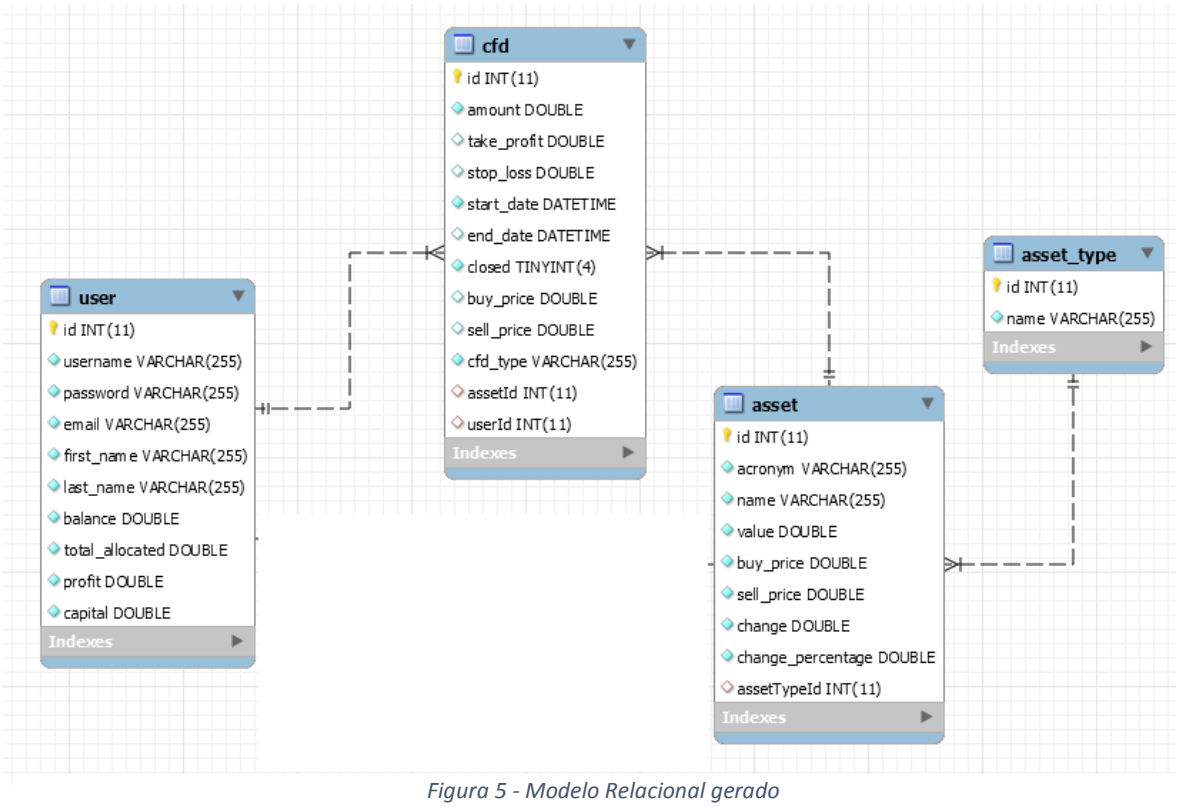

Figura 5 - Modelo Relacional gerado

A quarta camada e mais simples é a própria base de dados que é usada para guardar os dados. Com a implementação do pacote **TypeORM** e as notações nas classes é criado o seguinte modelo relacional à iniciação da aplicação e criada a base de dados isolada apenas com as especificações passadas nas classes e no ficheiro de configuração **ormconfig.json**.

Na aplicação existe a função **SeedDatabase()** que corre na inicialização da aplicação onde cria todas as instâncias dos ativos existentes e comunica com a API da Yahoo finance para obter valores reais e atualizados.

3.2.2. Diagrama de classes

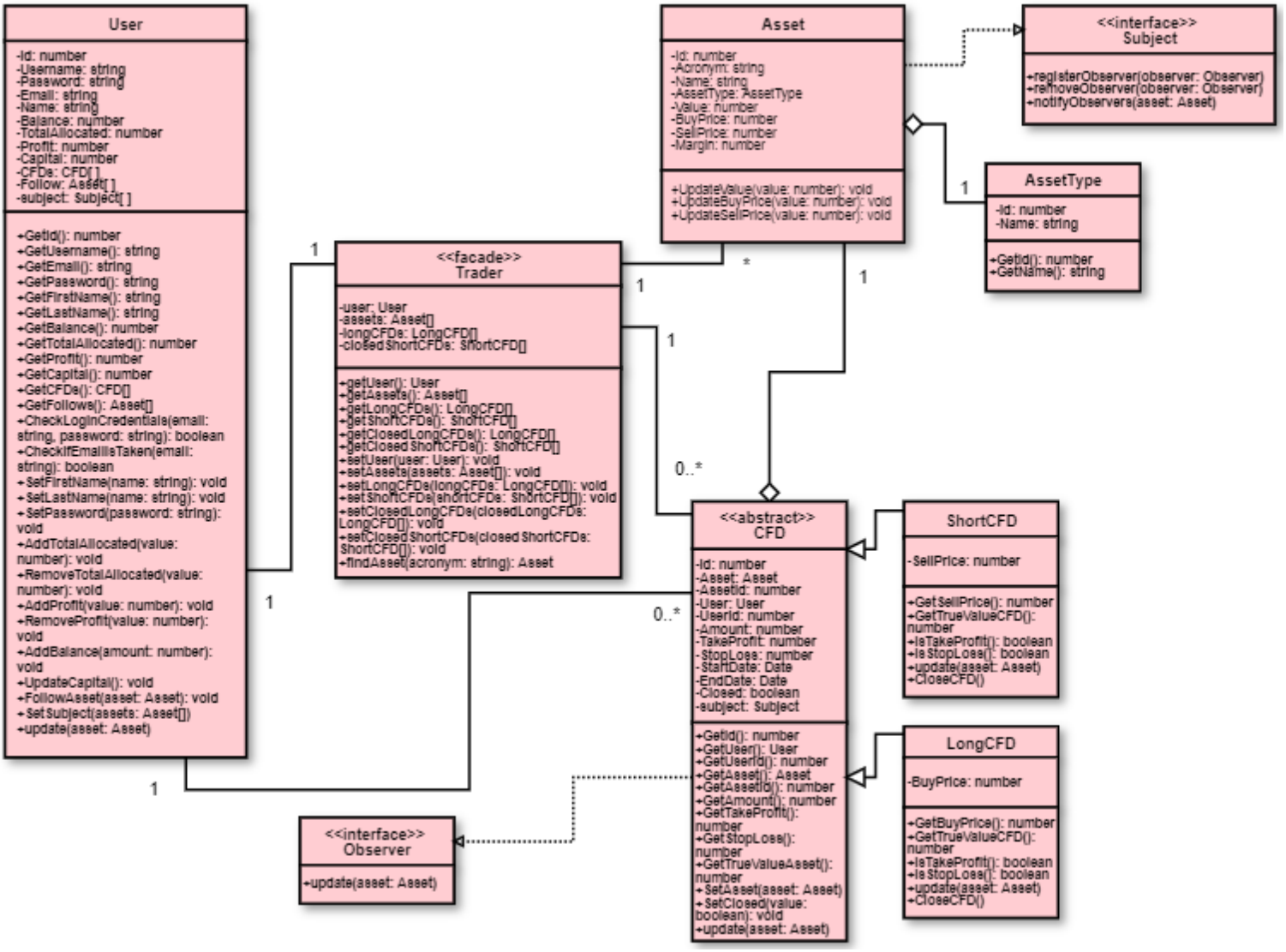

Figura 6 - Diagrama de Classes

Após análise e estrutura do modelo relacional temos adaptado o diagrama de classes da aplicação. Todas as entidades estão aqui representadas com os mesmos atributos que no modelo relacional e relações com mais alguns detalhes. Temos uma classe central da aplicação **Trader** para simular uma instância do login na aplicação com os dados do utilizador.

A classe **CFD** é uma classe abstrata que serve apenas para explicitar as suas propriedades bases às classes que estendem de si, o **ShortCFD** e **LongCFD** com cada das duas tendo as suas propriedades e métodos próprios funcionando diferentes de si.

Para apoio e como requisito do projeto temos interfaces para o padrão Observer, em que User e CFD implementam a interface Observer para observar a classe Asset, que por si, implementa a interface Subject para notificar todos os seus observadores quando existe alguma mudança.

3.2.2.1. Funcionamento

A aplicação está composta a funcionar dentro do esquema seguinte:

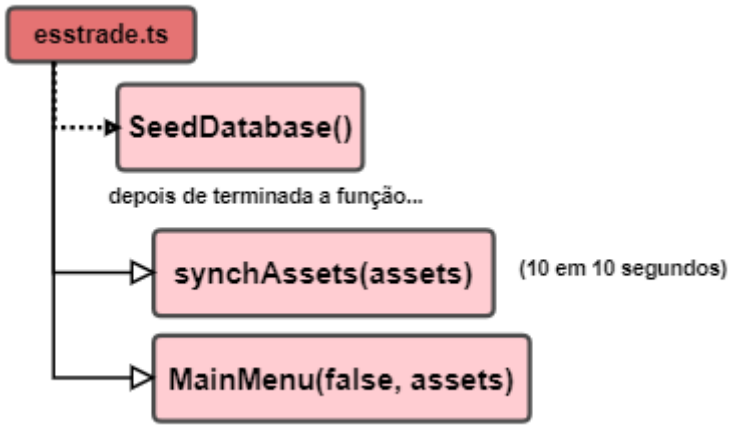

Figura 7 - Funcionamento da aplicação ao iniciar

No ficheiro **esstrade.ts** é onde a aplicação inicia, de início correndo a função **SeedDatabase()** onde popula os ativos com os valores da API e as cria na base de dados. Após a sua execução, corre duas funções, em que **synchAssets(assets)** corre de 10 em 10 segundos num processo no fundo e atualiza o valor dos ativos conforme a API do Yahoo e **MainMenu(false, assets)** que chama a função do menu principal da aplicação limpado o ecrã e enviando as instâncias dos assets criadas ao iniciar a aplicação.

3.2.2.2. Ativos

Os ativos correm e são atualizados na função **synchAssets(assets)** de 10 em 10 segundos para obter valores reais da API da Yahoo, atualizando assim os valores e obter valores reais. Corre no ínicio da aplicação por detrás do funcionamento da aplicação e interação com os menus, assim implementando com o padrão Observer, podemos enviar notificações de quando os ativos mudam de valor.

3.2.2.3. CFDs

Os CFDs dos utilizadores são fechados de duas maneiras: manualmente, por input dos utilizadores no menu de portfolio ou automaticamente à atualização do ativo a que o CFD pertence via padrão Observer, quando estes têm limites estipulados de **StopLoss** ou **TakeProfit**, sendo a margem atingida do CFD, fecha e notifica o utilizador de que foi fechado automaticamente no terminal.

3.2.3. Diagramas de Sequência

Para exemplos de diagramas de sequência da aplicação temos para duas funcionalidades:

- Ver Ativos
- Criar CFD de compra

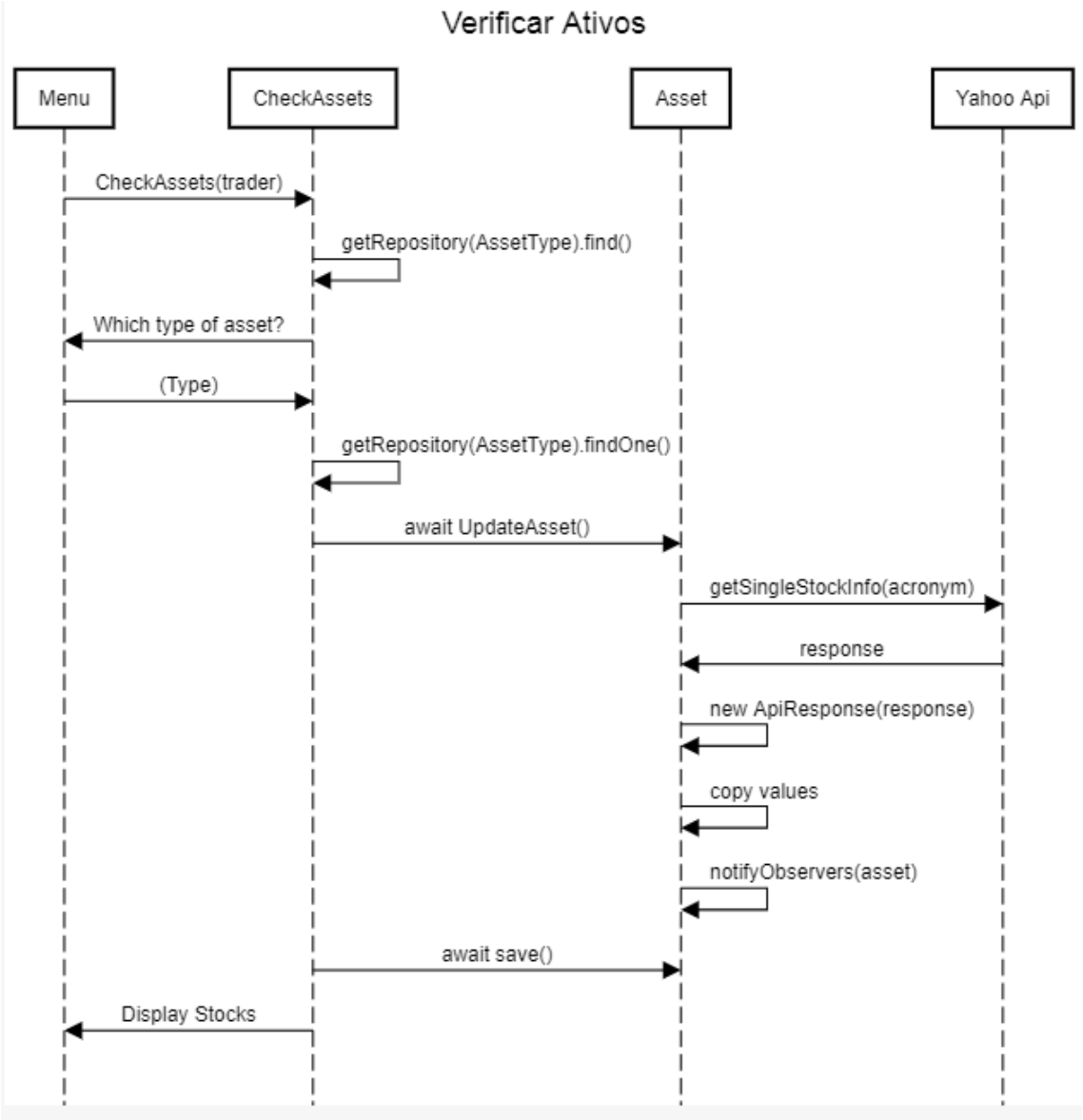

Figura 8 - Diagrama de Sequência (Verificar Ativos)

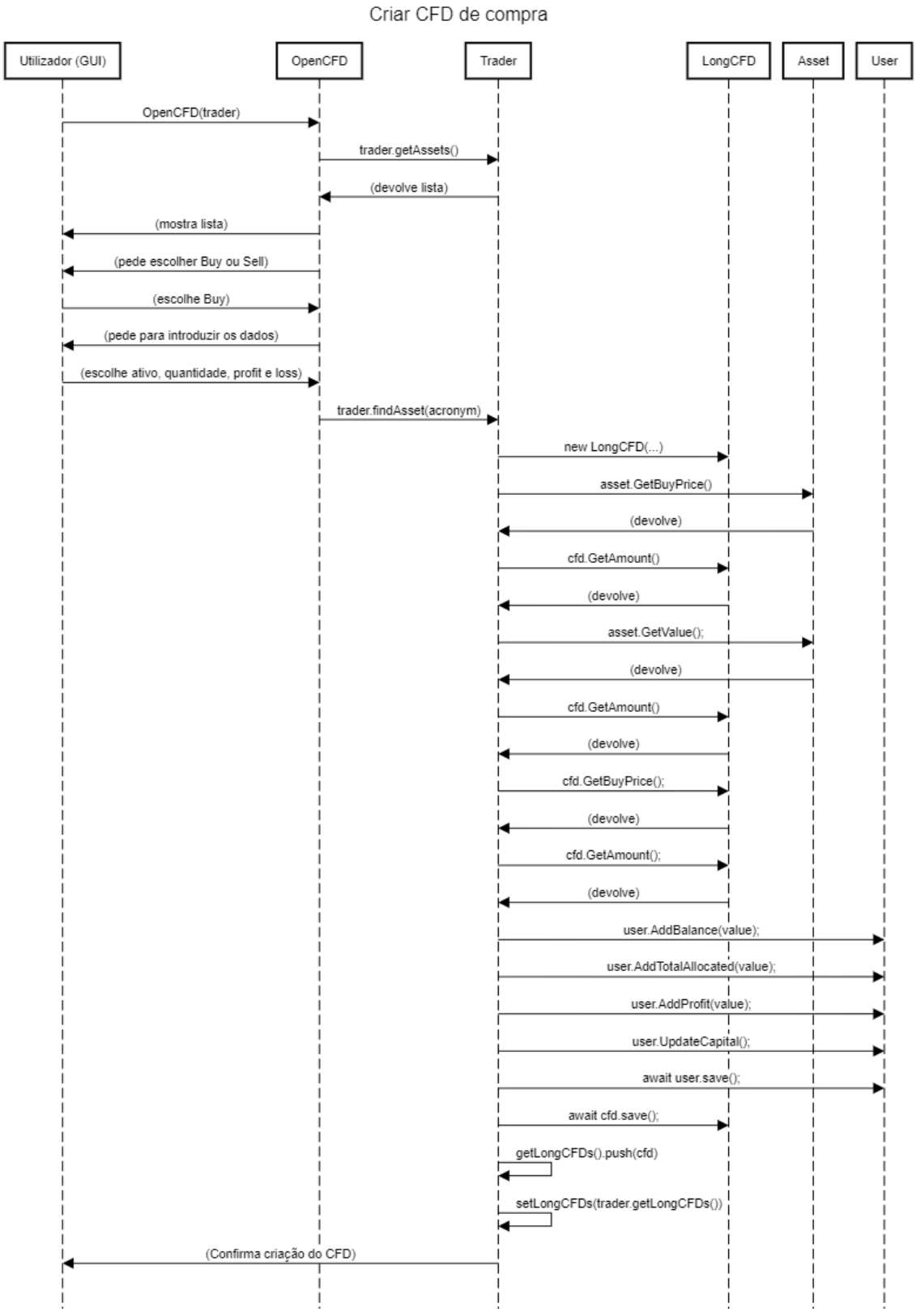

Figura 9 - Diagrama de Sequência (Criar CFD compra)

3.3. Cenários de atributos de qualidade

Foram especificados 3 cenários de atributos de qualidade para a aplicação para assegurar o funcionamento dos requisitos ou o produto em geral.

Dois tipos escolhidos foram:

- Confiabilidade
- Escalabilidade

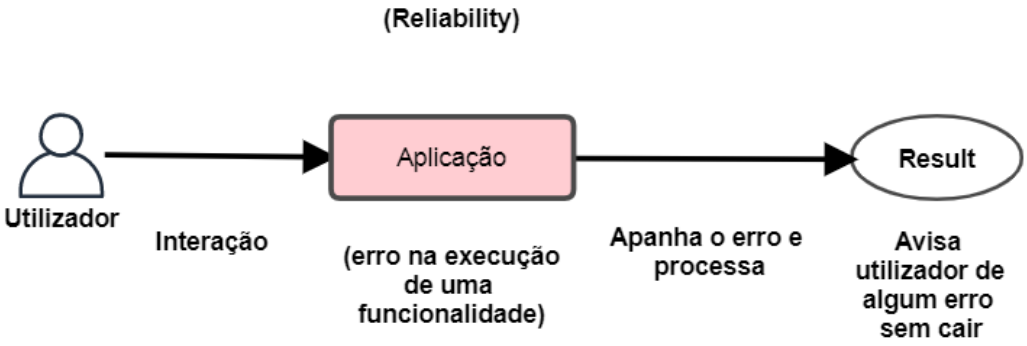

Figura 10 - Cenário confiabilidade

Neste cenário temos um caso em que o utilizador interage com a aplicação em que muitas operações ocorrem por detrás, propensas a erros, a aplicação está segura para validações dos inputs e operações das funções da aplicação sem deixando a aplicação deixar de funcionar, mostrando algum erro ou tratando com validações.

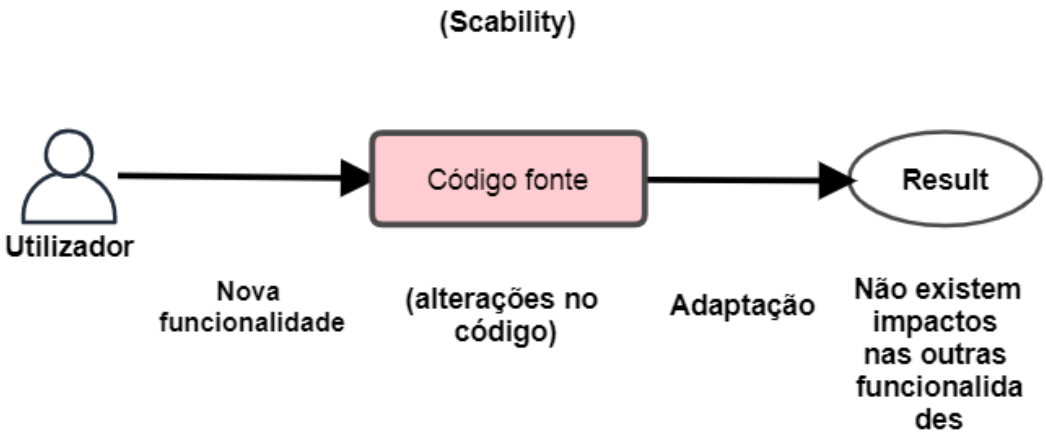

Figura 11 - Cenário escalabilidade

Neste cenário temos um posto em prática durante o projeto em que é pedido uma nova funcionalidade à aplicação e é necessárias alterações no código. Idealmente não queremos que se altere o que existe, apenas adicionar as propriedades ao código para que a funcionalidade seja implementada e sem impactos no resto do programa.

4. Padrões de design

Para a conceção da aplicação foram utilizados dois padrões de desenho:

- Observer
- Facade

4.1. Observer

O Observer é um padrão de design utilizado para notificar várias classes que **observam** uma outra por mudanças e assim responder às suas alterações no momento em que notifica os **subjects**.

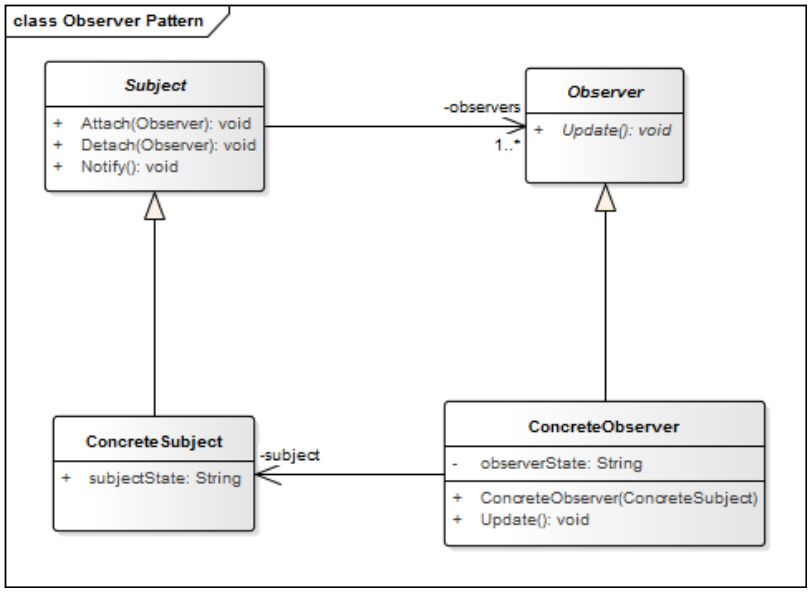

Figura 12 - Esquema do observer

A implementação do observer no projeto está presente entre o Asset e o CFD, em que o CFD observa o Asset por alterações às suas propriedades e notifica os CFDs para também atualizarem os seus.

```
import { Observer } from './observer';
import { Asset } from '../entities/asset';

export interface Subject {
    registerObserver(observer: Observer);
    removeObserver(observer: Observer);
    notifyObservers(asset: Asset);
}
```

```
import { Asset } from '../entities/asset';

export interface Observer {
    update(asset: Asset);
}
```

Foram criadas as duas interfaces necessárias à implementação do padrão. Subject é uma interface que é implementada nas classes que serão observadas, e a interface Observer é as que são implementadas nas classes que observam.

```
export class Asset extends BaseEntity implements Subject {
```

Figura 13- Asset implementa interface Subject

```
private observers: Observer[] = [];
```

Figura 14 - Asset contem a propriedade dos observadores que irá notificar.

```
registerObserver(observer: Observer) {
    this.observers.push(observer);
}
removeObserver(observer: Observer) {
    let index = this.observers.indexOf(observer);
    this.observers.splice(index, 1);
}
notifyObservers(asset: Asset) {
    for (let observer of this.observers) {
        observer.update(asset);
    }
}
```

Figura 15- Os métodos herdados da interface Subject

```
public async UpdateAsset() {
    var asset_copy = new Asset(this.Acronym, this.Name, this

    var response = await si.getSingleStockInfo(this.GetAcrony
    var apiResponse = new ApiResponse(response);
    this.Value = apiResponse.Price;
    this.BuyPrice = apiResponse.Buy;
    this.SellPrice = apiResponse.Sell;
    this.Change = apiResponse.Change;
    this.ChangePercentage = apiResponse.ChangePercentage;

    this.notifyObservers(asset_copy);
}
```

Figura 16 - Método que chama a notificação

O método **UpdateAsset()** é o responsável por atualizar os valores do ativo, em que no fim de comunicar com a API da Yahoo e fazer as suas alterações, manda uma cópia de si aos observadores para serem notificados das mudanças e adaptarem.

```
export abstract class CFD extends BaseEntity implements Observer {
```

Figura 17- Implementação do Observer no CFD

```
private subject: Subject;
```

Figura 18 - Propriedade do subject a observar no CFD

```
constructor(Asset: Asset, User: User, Amount: num
    super();
    this.Asset = Asset;
    this.User = User;
    this.Amount = Amount;
    this.TakeProfit = TakeProfit;
    this.StopLoss = StopLoss;
    this.StartDate = StartDate;
    this.EndDate = EndDate;
    this.Closed = Closed;

    this.subject = Asset;
    if (Asset != undefined)
        Asset.registerObserver(this);
}
```

Figura 19 - Construtor do CFD com o registo de observador ao asset

```
public update(asset: Asset) {
    this.Asset = asset;
}
```

Figura 20 - Método update da interface Observer

Chamando do Asset o método **notifyObservers()** o CFD consegue assim receber o Asset atualizado e atualizar o seu com qualquer alteração e estar atualizado com o valor atual.

4.2. Facade

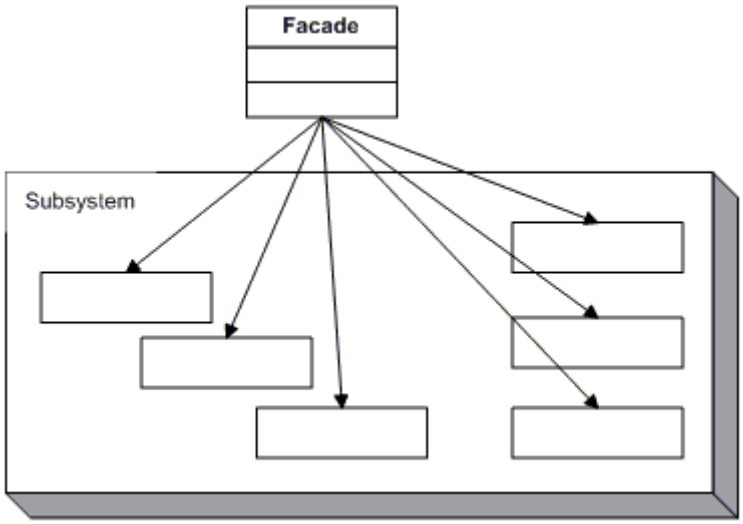

Figura 21 - Esquema do facade

O Facade é um esquema em que se corre variados métodos de uma ou mais classes, a partir de uma só.

Facade pode ser verificado no projeto na classe **Trader** onde todos as entidades da sessão do utilizador e da aplicação estão criadas e instanciadas para se puderem aceder através de uma instância do Trader, é a classe cetral da aplicação por onde tudo se controla.

```
export class Trader {

    private user: User;

    private assets: Asset[];

    private longCFDs: LongCFD[];
    private shortCFDs: ShortCFD[];

    private closedLongCFDs: LongCFD[];
    private closedShortCFDs: ShortCFD[];

    constructor(user: User, assets: Asset[], longCF
        //user logged in
        this.user = user;
        //assets running in the background
        this.assets = assets;
        //open cfds
        this.longCFDs = longCFDs;
        this.shortCFDs = shortCFDs;
        //closed cfds
        this.closedLongCFDs = closedLongCFDs;
        this.closedShortCFDs = closedShortCFDs;
    }
```

Figura 22- Facade implementado na classe Trader

5. Novo requisito (Seguir Ativo)

Poucos dias antes da entrega do projeto foi pedido um novo requisito ao projeto. Seria necessário implementar uma funcionalidade que permitisse ao utilizador seguir determinados ativos que escolha e de ser notificado de quando sofrem alguma alteração.

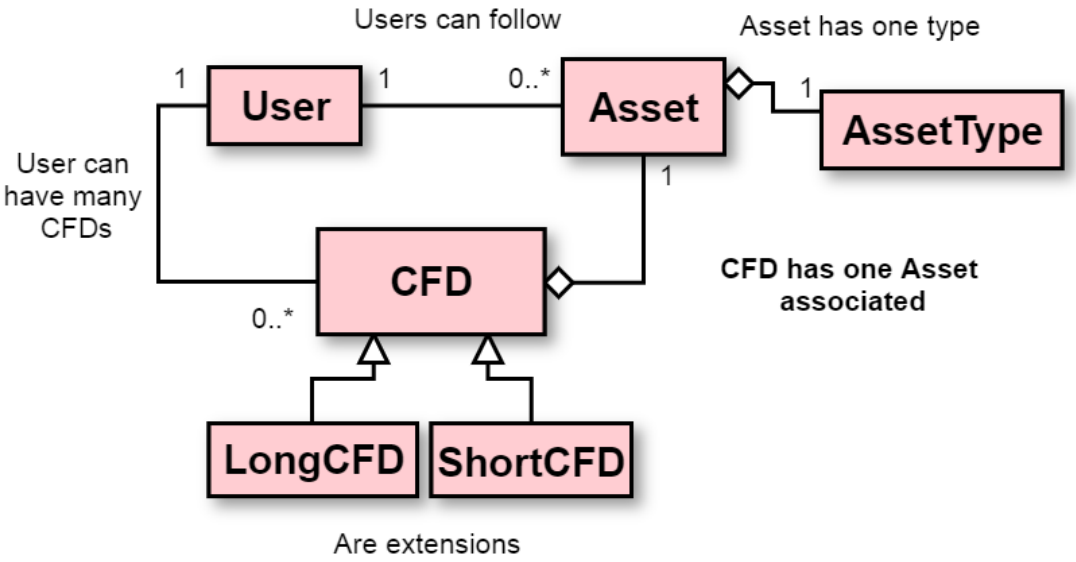

Figura 23 - Modelo de domínio novo

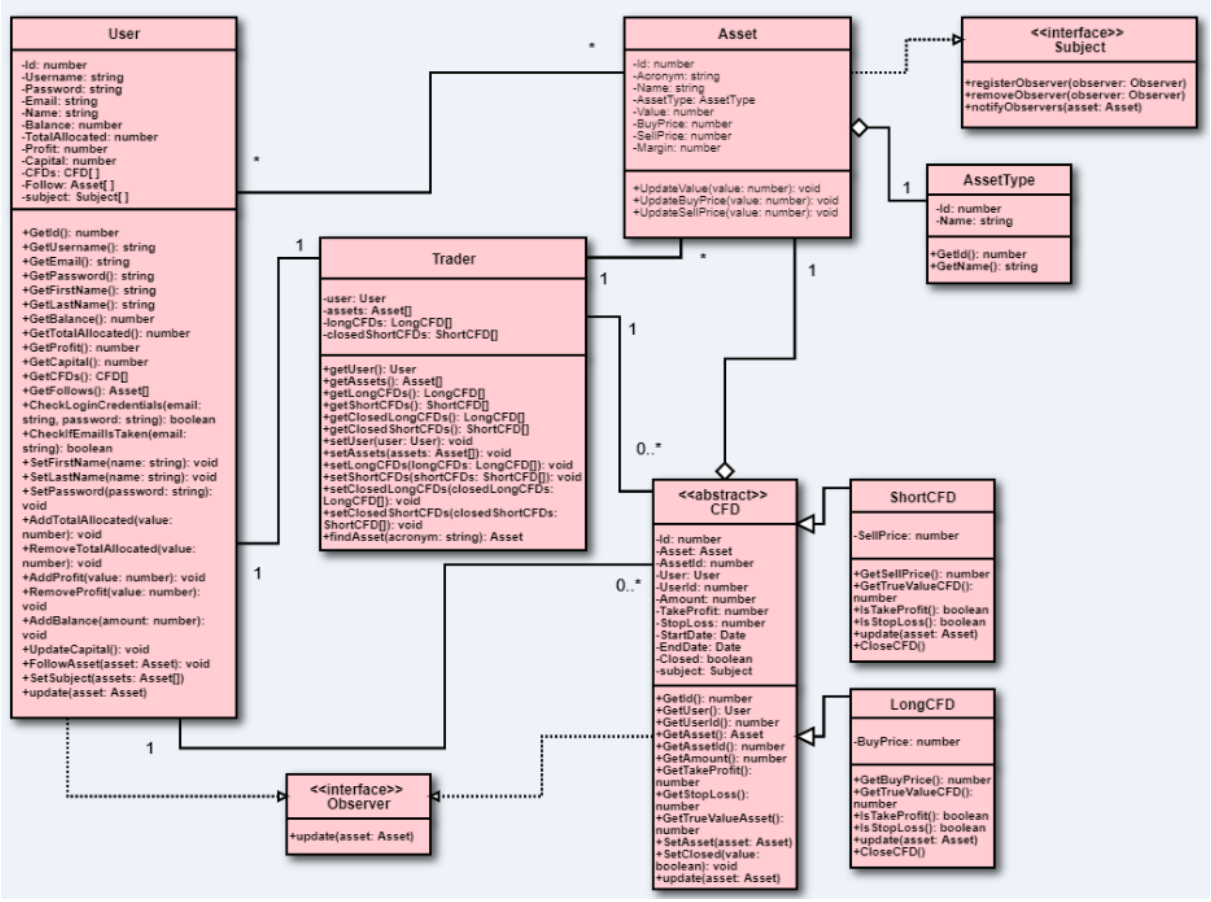

Figura 24 - Diagrama de Classes após novo requisito

Com este requisito pôde-se verificar se a aplicação estava bem definida e estruturada para não ser necessária alterações ou muito poucas. O requisito preenche os critérios para o padrão Observer, então foi implementada no **User** a interface **Observer** para também vigiar os ativos que deseja.

```
export class User extends BaseEntity implements Observer {
```

Figura 25 - Implementação do Observer no User

```
@ManyToMany(type => Asset, { eager: true })
@JoinTable()
private Follow: Asset[];

private subject: Subject[];
```

Figura 26 - Propriedades novas

Foram adicionadas propriedades novas, uma para conter a lista dos ativos que o utilizador deseja seguir, e um array de subjects que são os Asset. Isto para persistência reflete-se criando uma tabela de muitos para muitos através do ORM da aplicação.

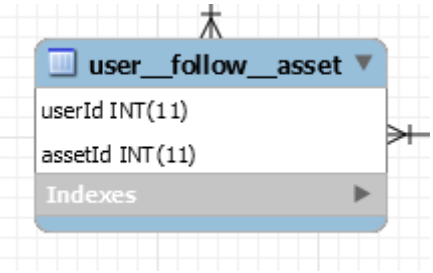

Figura 27 - Tabela muitos para muitos para guardar a lista dos ativos a seguir do utilizador

```
update(asset: Asset) {
    for (let index = 0; index < this.Follow.length; index++) {
        var followed = this.Follow[index];
        if (followed.GetAcronym() == asset.GetAcronym() && followed.GetValue() != asset.GetValue()) {
            console.log(chalk.blue(`\nThe asset ${asset.GetAcronym()} has changed value to ${asset.GetValue()} $ from ${followed.GetValue()} $`));
            return;
        }
    }
}
```

Figura 28 - Método Update do Subject no User

Por fim implementado o **update()** da interface adaptamos ao problema que queremos, recebendo notificação do Asset, recebemos uma cópia e notificamos ao utilizador da alteração do ativo que está a seguir.

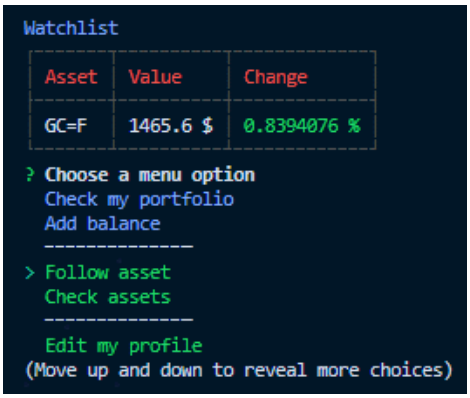

Figura 29 - UI dos assets a seguir e a opção no menu

Por fim levou à criação de mais uma função de apresentação **follow_asset.ts** para a interação e input do utilizador para adicionar ativos à lista.