



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

Introdução	3
1. Contexto do problema	4
2. Tecnologias	5
3. Análise	6
3.1. Domínio	6
3.2. Arquitetura	9
3.2.1. Padrão arquitetural	9
3.2.2. Modelo relacional	10
3.2.3. Diagrama de classes	11
3.2.4. Diagramas de Sequência	12
3.3. Cenários de atributos de qualidade	14
4. Padrões de design	15
4.1. Observer	15
5.1. Facade	17
6. Padrões de design	19

Ilustrações

Figura 1- Diagrama de Domínio	6
Figura 2- Diagrama de Use Cases.....	7
Figura 3 - Mockups da aplicação.....	8
Figura 4 - Padrão Arquitetural	9
Figura 5 - Modelo Relacional	10
Figura 6 - Diagrama de Classes	11
Figura 7 - Diagrama de Sequência (Verificar Ativos).....	12
Figura 8 - Diagrama de Sequência (Criar CFD compra).....	13
Figura 9 - Cenário confiabilidade	14
Figura 10 - Cenário escalabilidade	14
Figura 11 - Esquema do observer	15
Figura 12- Asset implementa interface Subject.....	15
Figura 13 - Asset contem a propriedade dos observadores que irá notificar.	16
Figura 14- Os métodos herdados da interface Subject.....	16
Figura 15 - Método que chama a notificação	16
Figura 16- Implementação do Observer no CFD.....	16
Figura 17 - Propriedade do subject a observar no CFD	16
Figura 18 - Construtor do CFD com o registo de observador ao asset	17
Figura 19 - Método update da interface Observer	17
Figura 20 - Esquema do facade	17
Figura 21 - Facade no projeto	18
Figura 22 - Implementação do Observer no User.....	19
Figura 23 - Propriedades novas	19
Figura 24 - Tabela muitos para muitos para guardar a lista dos ativos a seguir do utilizador	19
Figura 25 - Método Update do Subject no User	19
Figura 26 - UI dos assets a seguir e a opção no menu	20

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 1ª 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 poder aceder a uma lista de variados ativos financeiros e dar a possibilidade ao utilizador de comprar ou vender ativos através 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**.

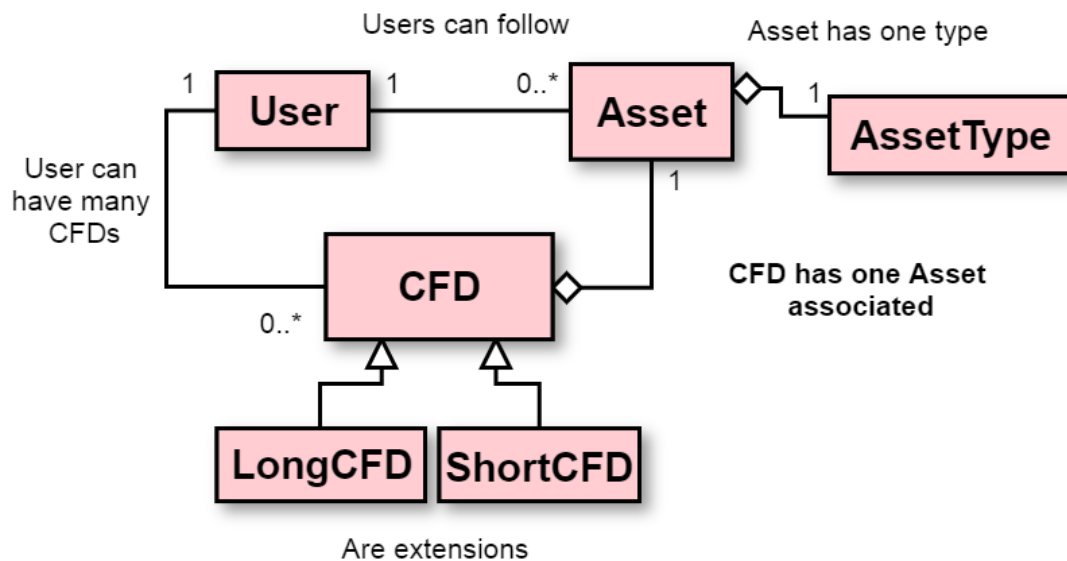
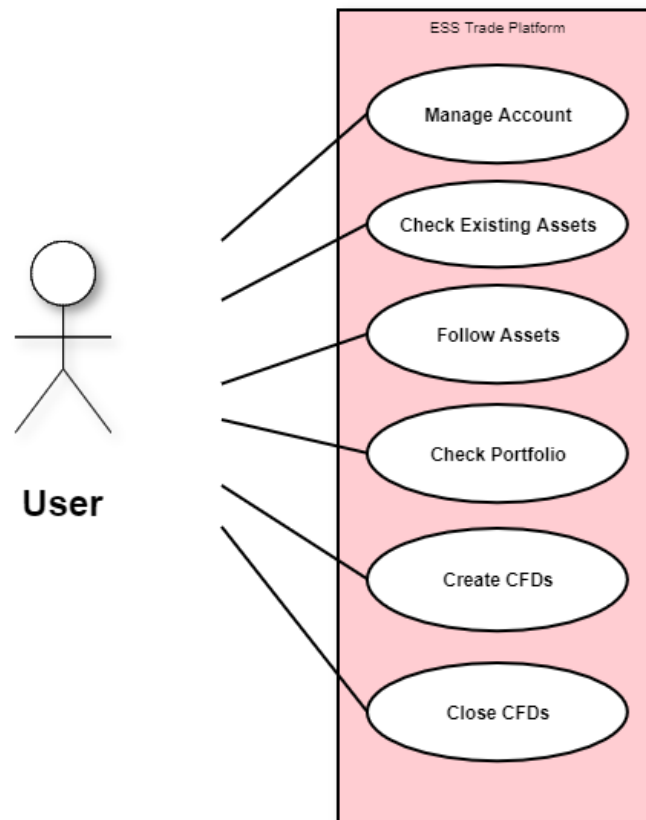


Figura 1- Diagrama de Domínio

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. Mais tarde foi lançado um pedido para um novo requisito em que utilizadores podem seguir ativos para ter notificações sobre as suas alterações.

Diagrama de Use Cases

*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

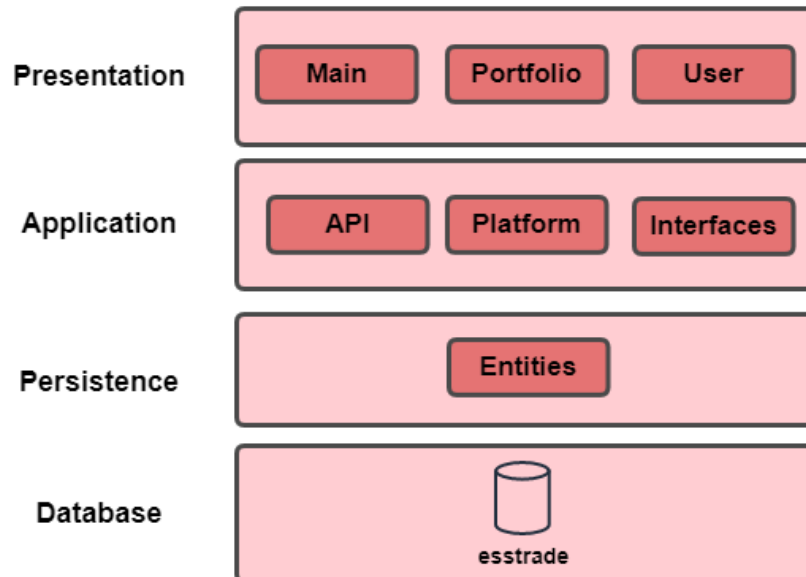


Figura 4 - Padrão Arquitetural

Provavelmente o padrão mais utilizado e o que foi implementado nesta aplicação, uma arquitetura em camadas em que cada camada interage com as de baixo tendo pouco ou nenhum conhecimento das camadas.

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 nesse aspeto.

Segunda camada de aplicação contém os componentes responsáveis pela aplicação ao problema e funcionamento do mesmo, em que api está responsável pela cobertura de adaptação com a api da Yahoo finance para processar os dados, platform a classe mãe que incorpora todos os objetos correntes da aplicação enquanto funciona, e interfaces onde estão guardadas interfaces comuns usadas para resolver e ajudar às funcionalidades do programa.

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, extendendo de uma classe auxiliar do pacote **TypeORM**. Esta camada também engloba um pouco das responsabilidades da camada de aplicação.

A quarta camada e mais simples é a própria base de dados que é usada para guardar os dados.

3.2.2. Modelo relacional

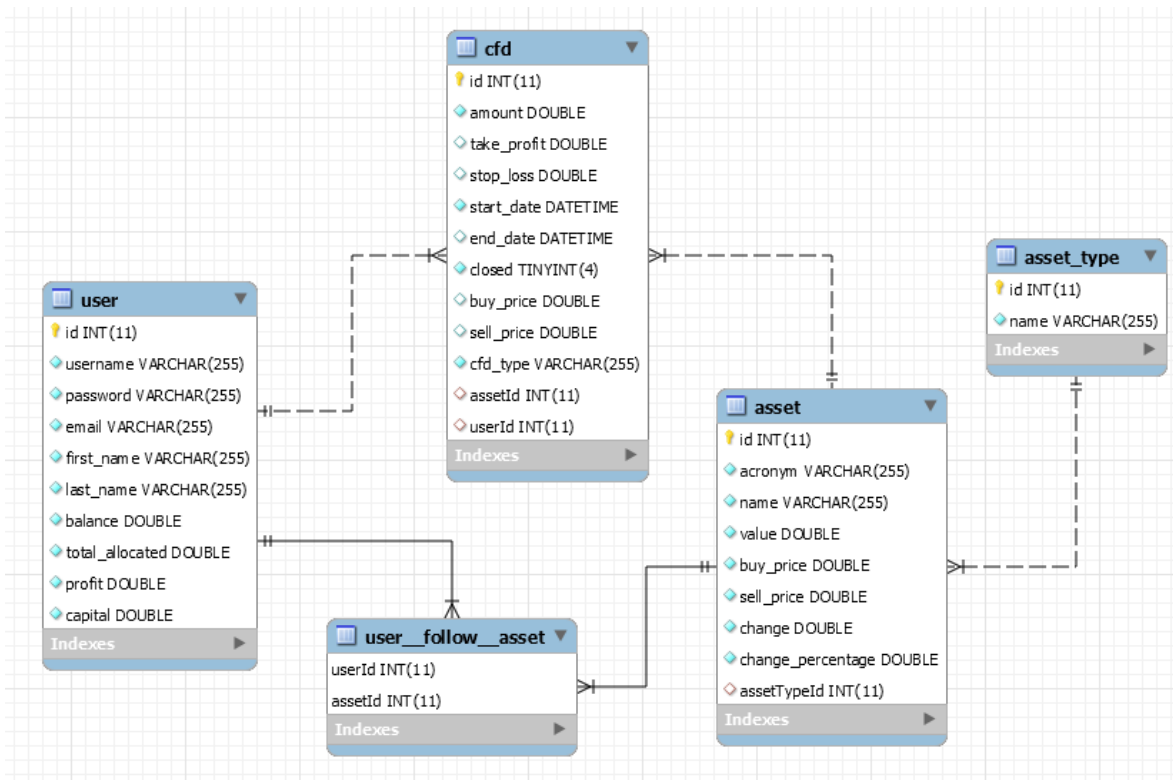


Figura 5 - Modelo Relacional

Aplicando da análise do domínio do problema chegamos ao seguinte modelo relacional usado para persistir os dados da aplicação. Com os utilizadores **user** que terá um **username**, **password**, **email**, **first_name** e **last_name** opcionais para definir depois, **balance** que é o dinheiro em conta na aplicação, **total_allocated** que é a quantia total investida em CFDs, **profit** em que é apresentado o investimento total no momento com os CFDs ativos e por fim **capital** em que apresenta o resultado provisório atual do utilizador.

Os ativos estão representados em **asset**, composto por **acronym**, o seu identificador na bolsa, **name** para identificador qual o nome, **value** que apresenta o valor atual do ativo, **buy_price** e **sell_price** são o valor de compra e venda do ativo respetivamente, **change** e **change_percentage** são a mudança variável do ativo em número e percentagem e por fim **assetTypeId** que guarda a referência a que tipo de ativo é. **Asset_type** é a tabela representativa do tipo de ativo que é, podendo ser stock, commodity ou moeda.

No meio temos **user_follow_asset** a tabela usada para guardar as listas de seguimento dos ativos dos utilizadores e ativos.

Por fim temos **cfd** que é a representação do contrato entre o utilizador e ativo tendo **amount** para a quantia comprada do ativo, **take_profit** e **stop_loss** são os limites impostos para o cfd fechar automaticamente, **start_date** e **end_date** são as datas de início e fim do cfd, **buy_price** para o valor do ativo no ato da compra de CFDs, **sell_price** para o valor do ativo no ato da venda de CFD, **cfd_type** usado para diferenciar entre um LongCFD e ShortCFD, **assetId** para guardar o ativo referente ao contrato e **userId** para a referência ao utilizador que iniciou o contrato.

3.2.3. Diagrama de classes

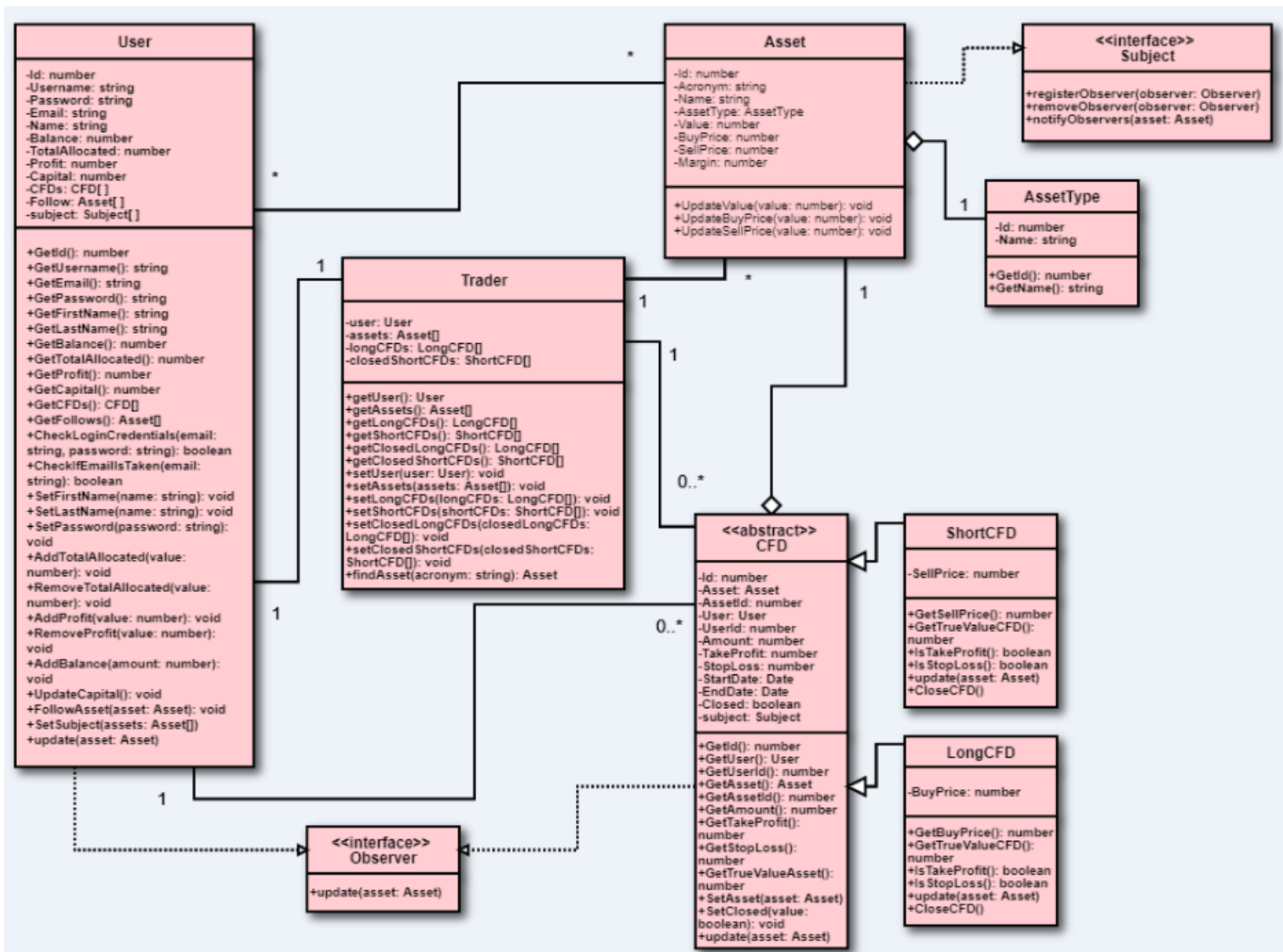


Figura 6 - Diagrama de Classes

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.

3.2.4. Diagramas de Sequência

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

- Ver Ativos
- Criar CFD de compra

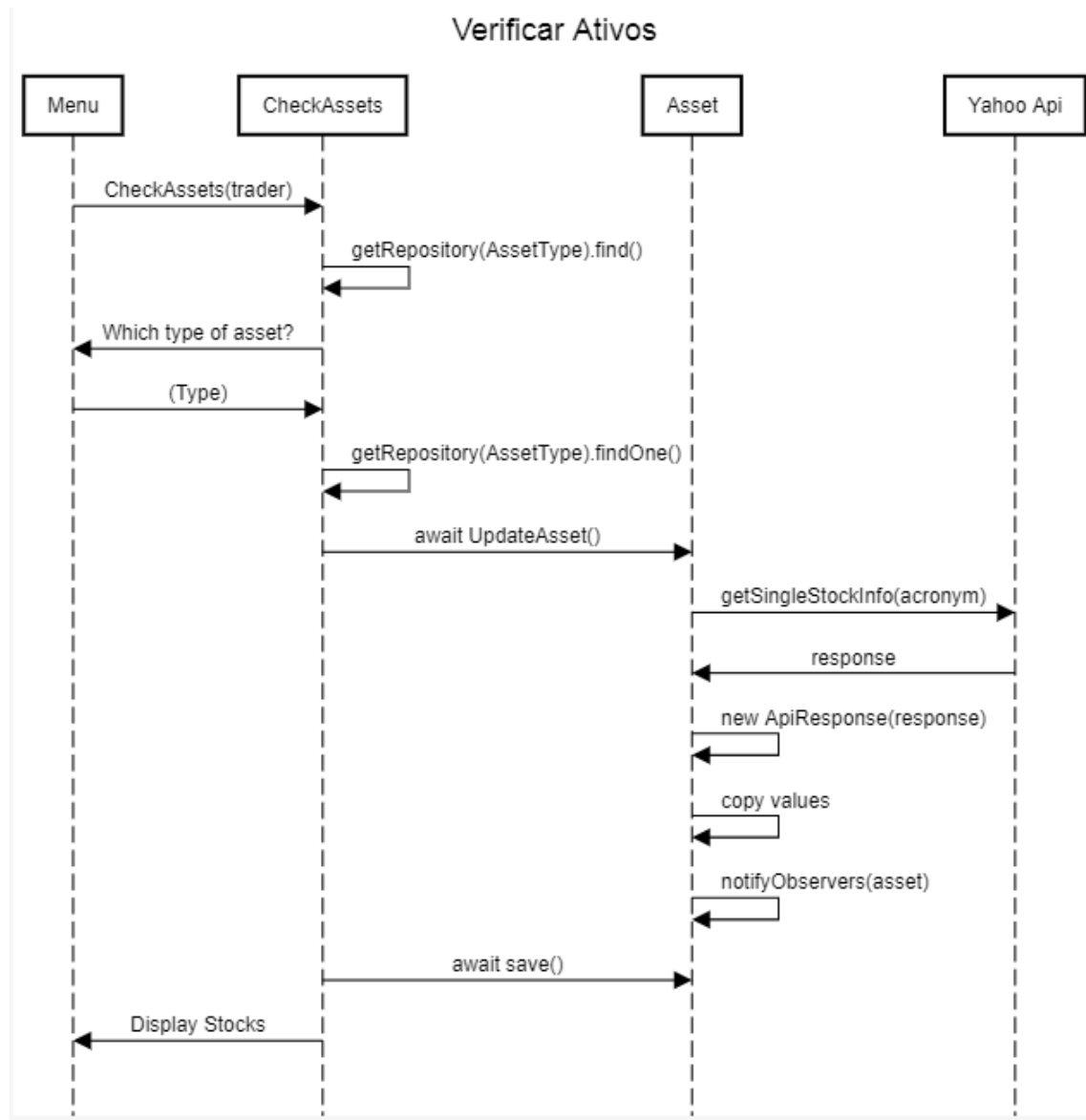


Figura 7 - Diagrama de Sequência (Verificar Ativos)

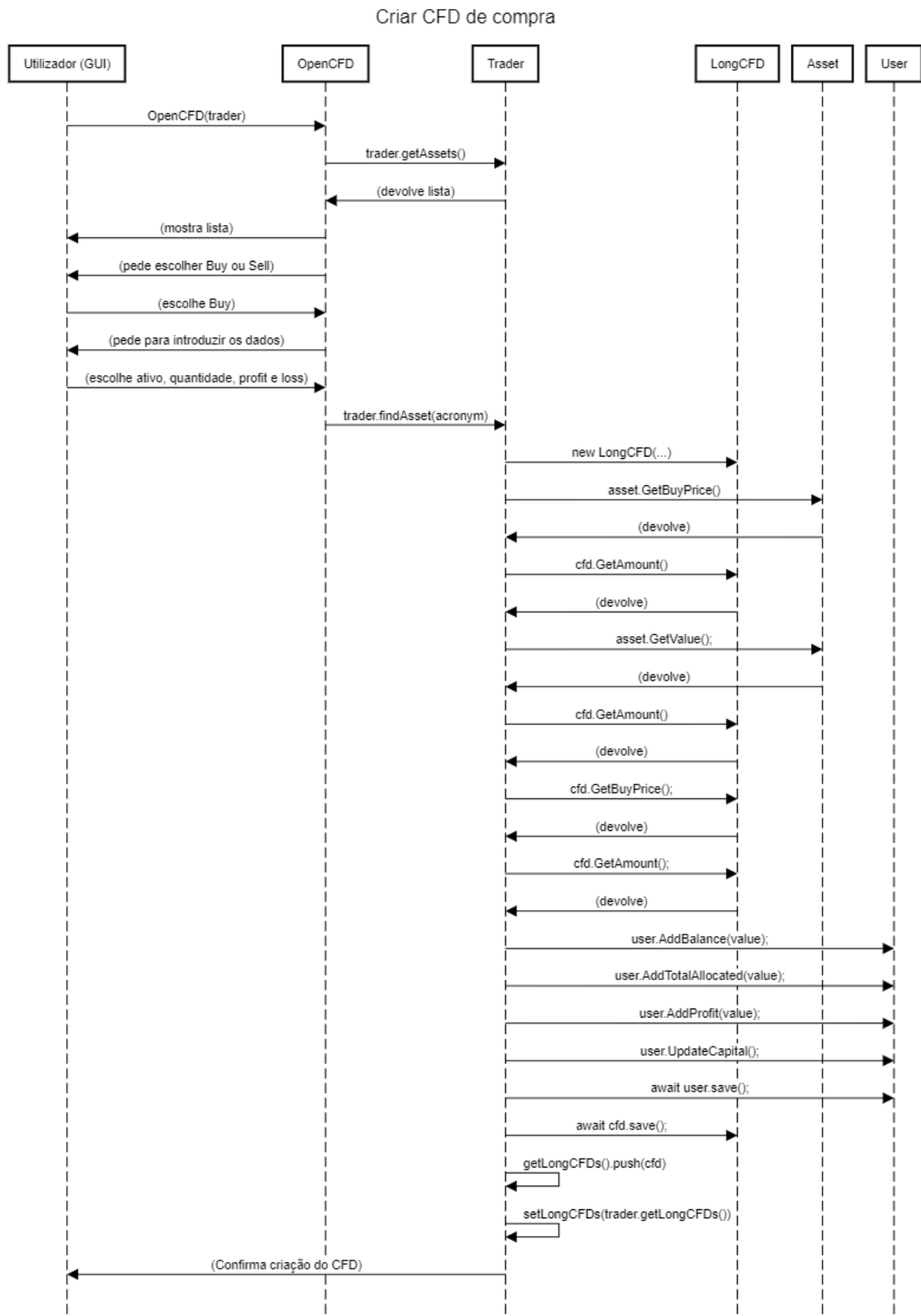


Figura 8 - Diagrama de Sequência (Criar CFD compra)

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

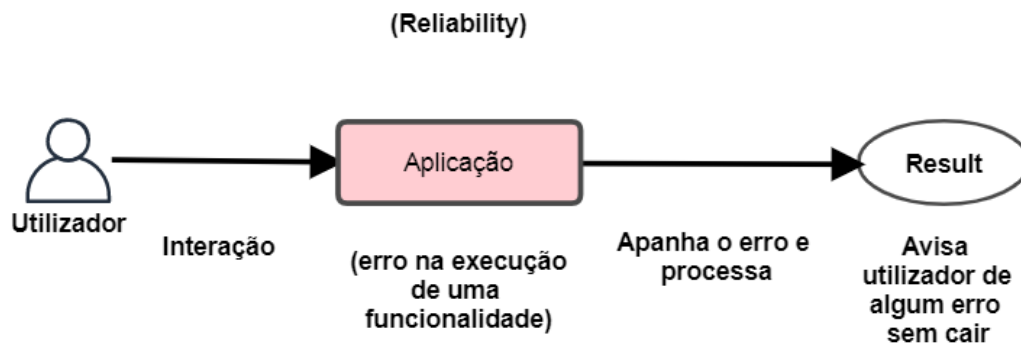


Figura 9 - Cenário confiabilidade

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.

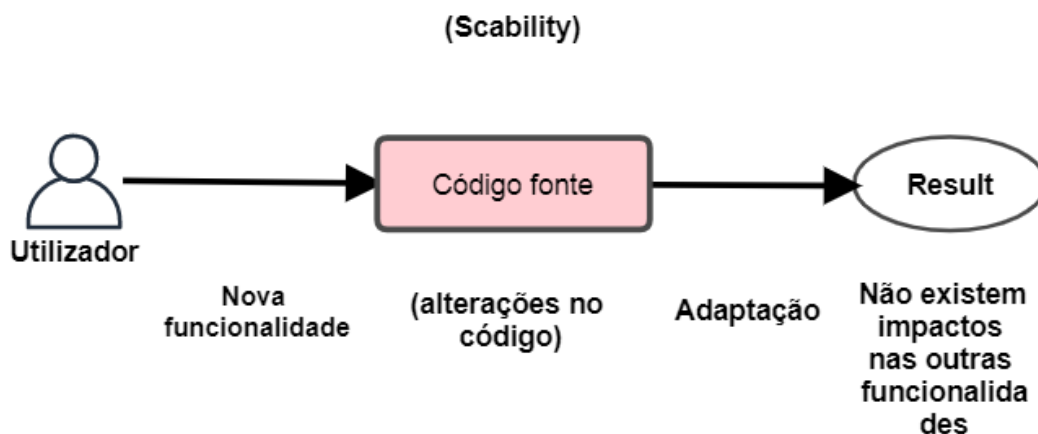


Figura 10 - Cenário escalabilidade

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 concepção 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**.

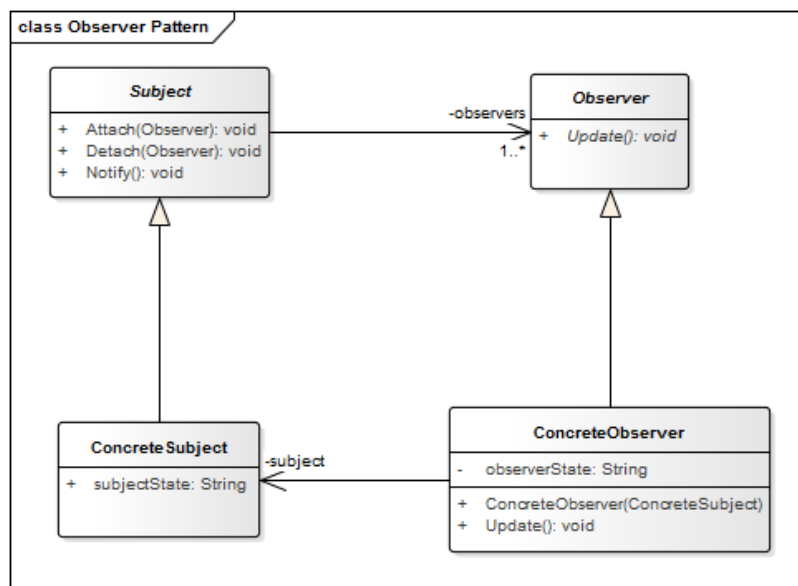


Figura 11 - Esquema do observer

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 12- Asset implementa interface Subject


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

Figura 13 - Asset contém 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 14- 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.GetAcronym);  
    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 15 - 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 16- Implementação do Observer no CFD

```
private subject: Subject;
```

Figura 17 - 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 18 - Construtor do CFD com o registo de observador ao asset

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

Figura 19 - 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

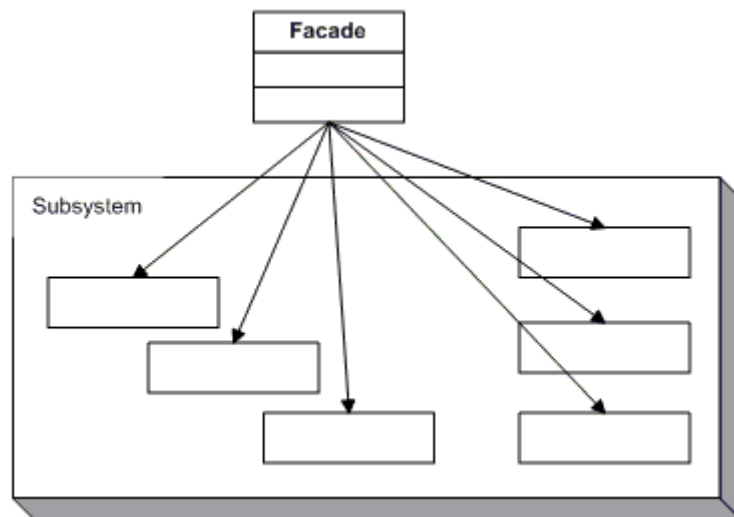


Figura 20 - Esquema do facade

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

```
public async CloseCFD() {  
    var profit = this.GetTrueValueCFD() - this.GetTrueValueAsset();  
    var newBalance = profit + this.GetTrueValueAsset();  
  
    this.GetUser().AddBalance(newBalance);  
    this.GetUser().RemoveTotalAllocated(this.GetTrueValueCFD())  
    this.GetUser().RemoveProfit(profit);  
    this.GetUser().UpdateCapital();  
    await this.GetUser().save();  
    this.SetClosed(true);  
    console.log(`\nThe Buy CFD for ${this.GetAsset().GetAcronym()} has been closed.`);  
}
```

Figura 21 - Facade no projeto

Facade pode ser verificado no projeto nas extensões à class abstrata CFD, o LongCFD e ShortCFD.

5. Padrões de design

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.

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 22 - Implementação do Observer no User

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

private subject: Subject[];
```

Figura 23 - 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.

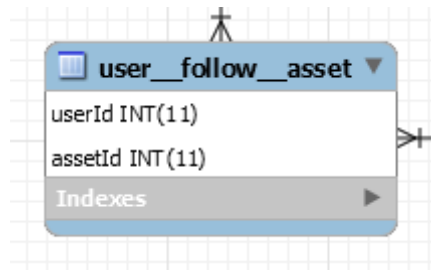
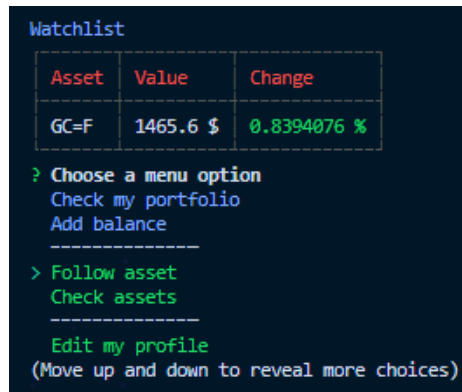


Figura 24 - Tabela muitos para muitos para guardar a lista dos ativos a seguir do utilizador

```
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 25 - 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.



The image shows a terminal window with a dark background. At the top, the word 'Watchlist' is displayed in blue. Below it is a table with three columns: 'Asset', 'Value', and 'Change'. The first row of data shows 'GC=F' as the asset, '1465.6 \$' as the value, and '0.8394076 %' as the change. Below the table, there is a menu titled 'Choose a menu option' with three items: 'Check my portfolio', 'Add balance', and 'Follow asset'. The 'Follow asset' option is highlighted with a green prompt character '>'. Below the menu, there is a section titled 'Edit my profile' with a note '(Move up and down to reveal more choices)'.

Asset	Value	Change
GC=F	1465.6 \$	0.8394076 %

> Choose a menu option
Check my portfolio
Add balance

> Follow asset
Check assets

Edit my profile
(Move up and down to reveal more choices)

Figura 26 - 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.