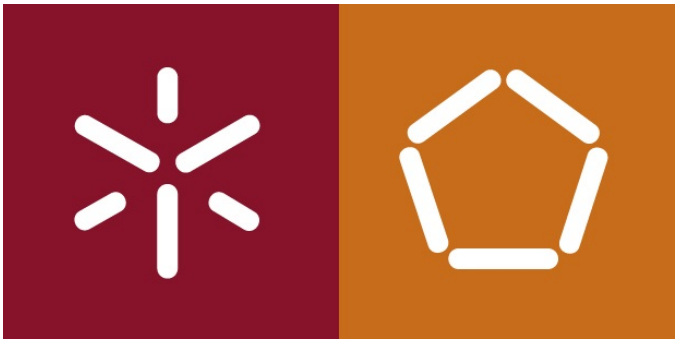
Universidade do Minho

Mestrado Integrado em Engenharia Informática -
Engenharia de Sistemas de Software



ESS Trading Platform

Relatório do Trabalho Prático 1 - Parte 2

Arquitetura de Software

Francisco Freitas

A81580

Pedro Freitas

A80975

November 30, 2019

Contents

1 Introdução

No âmbito da unidade curricular de Arquitetura de Software foi-nos proposto um trabalho que consistiu duas fases. Passada a primeira fase onde tivemos de implementar uma solução a um problema de um enunciado, sem dar muita ênfase nem ter muito cuidado com a arquitetura da solução, chegou a hora de reconstruir a nossa ideia inicial tomando muita atenção à arquitetura que desejavamos ter e os padrões de design que queríamos aplicar.

Assim durante este relatório vamos explicar o resultado final e justificar algumas opções tomadas.

2 ESS Trading Platform

Uma plataforma de negociação é uma aplicação que permite investidores e traders abrir, fechar e gerir posições no mercado financeiro, que podem envolver compra e venda de ativos financeiros, por exemplo ações, commodities (ouro, petróleo), índices ou moeda.

Nesta plataforma baseamo-nos em contratos CFD (Contract For Differences), que é estabelecida entre duas partes : um "comprador"(long) e um "vendedor"(short). Existem dois tipos destes contratos que é definido quando se estabelece o contrato. Quando o utilizador acha que o valor atual de um ativo vai subir ele estabelece um **contrato de compra**, onde o lucro desse contrato é dado por: *ValorVendaFutura - ValorCompraAtual*. Quando o utilizar acha que um valor atual de um ativo vai descer estabece um **contrato de venda**. Neste tipo de contrato o lucro que vai obter é dado por *ValorVendaAtual - ValorCompraFutura*.

Esta negociação não implica que tenhamos o produto em causa. Por exemplo, podemos negociar uma ação de uma entidade sem de facto a termos.

Assim esta plataforma terá de seguir alguns requisitos pré-estabelecidos:

- A plataforma deverá ser responsável por manter os valores dos ativos a serem negociados via CFD's
- A plataforma deverá permitir a abertura de contas a investidores com plafond inicial para investimento.
- A plataforma deverá permitir que investidores abram posições (CFDs) sobre ativos disponíveis, quer seja de compra ou de venda. Também deverá ser possível definir valores para fecho de contrato : valores de *Take profit* e de *Stop Loss*.
- A plataforma deverá permitir aos investidores monitorizar em tempo real o seu portfolio de CFDs e para cada um visualizar o valor atual do ativo adquirido.

3 Principais Funcionalidades

O objetivo da plataforma é gestão de posições (CFD-Contract For Differences),isto é,abrir, fechar e gerir posições no mercado financeiro referentes a ativos,como por exemplo ações e commodities. Para esse efeito a plataforma tem que suportar as seguintes Funcionalidades.

- Registar Utilizadores
- Comprar e vender contratos
- Monitorizar em 'tempo real' do portfólio de CFDs de cada utilizador
- Visualizar o valor atual dos ativos adquiridos pelos Utilizadores

4 Atributos de qualidade

Depois de uma análise detalhada dos requisitos gerais, chegou-se à conclusão que o sistema teria que possuir os seguintes atributos de qualidade.

4.1 Facilidade de Modificação

O sistema desenvolvido permite com alguma facilidade a modificação da interface gráfica do Utilizador.

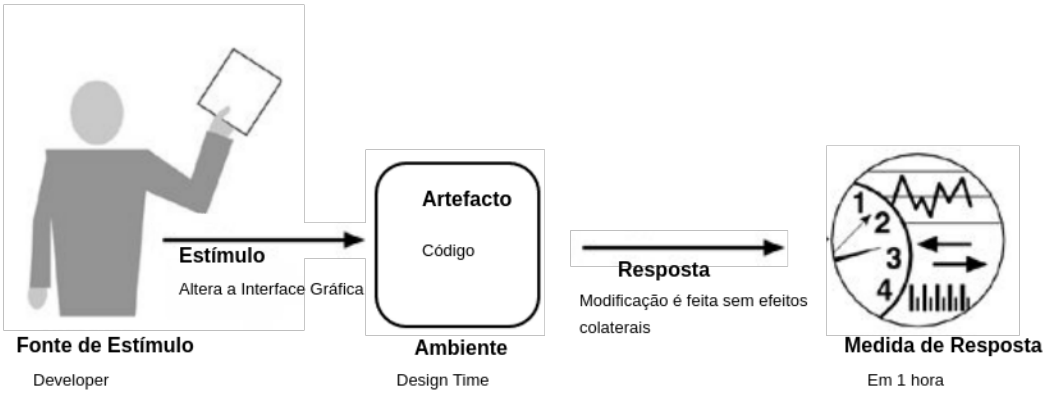


Figure 1: Cenário concreto de Facilidade de modificação

O sistema foi construído tendo em conta a escabilidade.Sendo assim, o acréscimo de novas funcionalidades não implica grande modificações no sistema.Como o sistema foi desenvolvido com o padrão arquitetural Cliente-Servidor todas as modificações feitas no Servidor não afetam o Cliente.

4.2 Facilidade de Utilização

O sistema deve ser simples de forma a que a sua utilização seja intuitiva.

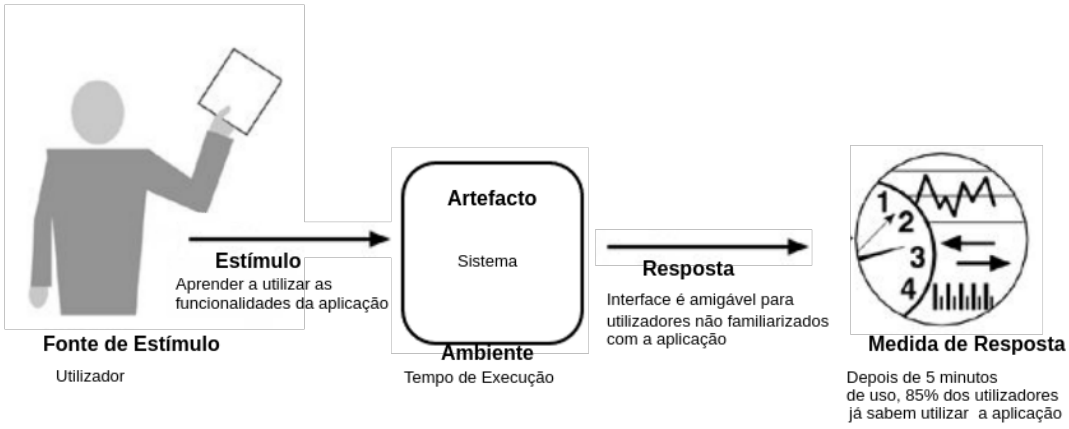


Figure 2: Cenário concreto de Utilização

A utilização do sistema deve ser fácil e direta,sendo que o único requisito para usufruir deste é possuir conhecimentos básicos de manuseamento de computores/internet.

4.3 Segurança

O sistema deve proporcionar segurança dos dados pessoais dos utilzadores.

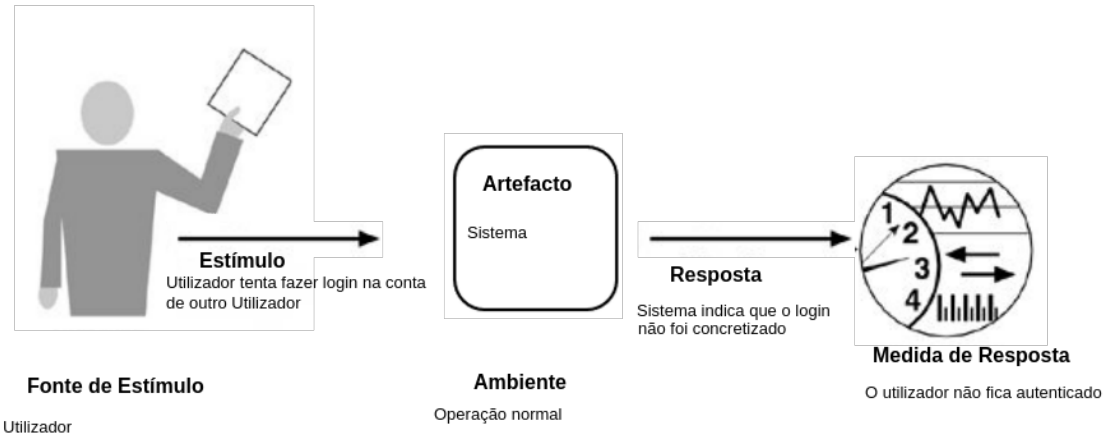


Figure 3: Cenário concreto de Segurança

O sistema, através da autenticação, protege os dados pessoais.Deste modo só um utilizador autenticado é capaz de usufruir das funcionalidades do sistema. Para além disso este é auditável, isto é , regista a actividades a um nível de detalhe suficiente para o reconstruir.No caso do sistema são guardadas todas as transações entre cliente e Servidor para, em caso de falha do sistema, essas transações poderem ser recuperadas.

5 Condicionantes

Condicionante	Descrição
Implementação em java	A implementação da plataforma será feita em java. Posto isto, existem restrições provenientes da línguagem.
Execução por terminal ou IDE	A plataforma não apresenta interface gráfica para o utilizador. Por este motivo será executada na linha de comandos do terminal ou num IDE que suporte Java
Imparcialidade	A plataforma terá que ser suportada nos diferentes sistemas operativos

6 Estratégia de Solução

Depois de uma análise detalhada do problema e da construção do modelo de domínio verificou-se que as entidades principais da ESS Ltd são o Utilizador, Contrato, Ativo e o Registo.

O Ativo contém as informações relevantes sobre as ações e *commodities* a serem negociadas que vão sendo atualizadas,em tempo real, com os dados recolhidos da API. Sempre que um utilizador comprar ou vender contratos vai-se ao Ativo buscar os valores de compra e venda deste e associa-se um destes ao contrato. Caso o utilizador queira consultar um valor passado do Ativo terá que aceder aos seus registos ou contratos referentes a esse ativo.

O Contrato é criado pela necessidade de guardar informações acerca das compras/vendas efetuadas sobre ativos por um utilizador. Por isso guarda consigo o identificador do utilizador e do ativo, bem como a quantidade de ativo e o valor pelo o qual este foi adquirido. Para conseguir distinguir contratos de venda e contratos de compra foi adicionado um booleano. Para que o utilizador consiga distinguir no seu portfólio contratos fechados e contratos abertos é seguida a mesma metodologia acrescentando um booleano (*true* se fechado, *false* se aberto). Nesta primeira versão quando um contrato é criado, o utilizador tem que definir sempre o Stop Loss e o Take Profit.

O Registo foi criado com intuito do utilizador poder ver os seus contratos fechados e ter uma noção da variação do valor de ativos já adquiridos. Para esse efeito o registo tem um identificador, o identificador do utilizador e o identificador do contrato.Para além disso contém o lucro/perda realizado com o fecho do contrato.

Depois de uma análise detalhada dos requisitos da aplicação, chegou-se á conclusão que é necessário ter uma conexão servidor-cliente. O cliente manda mensagens ao servidor (pedidos). Estas mensagens são interpretadas pelo servidor enviando também uma mensagem como resposta para o cliente. Desta forma é possível o serviço pode ser utilizado por vários clientes em simultâneo.

Para manter os valores dos ativos sempre atualizados foi criada uma classe que através de Threads vai fazendo alterações na base de dados no preço de compra e venda do ativo.

6.1 Arquitetura

O foco principal de todo este projeto será a organização e modelação da arquitetura do nosso sistema. Tendo em conta que os objetivos funcionais são permitir que um utilizador consiga investir em ativos de imediato concluímos que a arquitetura indicada deveria ser uma arquitetura **three-tiered client-server**.

Esta arquitetura caracteriza-se por permitir todas as funcionalidades de uma arquitetura de cliente-servidor "normal" onde o cliente faz um pedido a um servidor e este responde ao cliente, e também dividir todo o sistema em três camadas principais. As camadas presentes nete tipo de arquitetura são a camada do utilizador, a camada de middle-tier e por fim a camada de backend.

A camada do utilizador caracteriza-se pela interface do utilizador. Esta interface permite ao utilizador interagir com o sistema e fazer-lhe pedidos. O sistema tratará de apenas devolver a resposta ao pedido que o utilizador realizou.

A camada de middle-tier é a camada de aplicação do sistema e é a responsável por receber os pedidos do utilizador e enviar de volta as respostas. Ele processa os pedidos que chegam, calcula os dados que vai precisar da camada de dados e pede esses dados. Recebendo esses dados ele calcula o que tem de responder e como e edevolve a resposta ao utilizador que fez o pedido.

Por último a camada de backend é a camada de dados. Esta camada recebe pedidos de dados vindas da camada de aplicação e devolve os dados que lhe foram pedidos.

Através desta relação entre as três camadas é possível ao utilizador usufruir de todas as funcionalidades do sistema.

6.1.1 No nosso sistema

No nosso sistema cada camada corresponde a um package presente no projeto.

O package **Cliente** corresponde à camada do utilizador. Tal como foi referido anteriormente esta camada é responsável pela interface e responsável por permitir ao cliente fazer pedidos ao sistema. Neste package temos a classe que representa o menú (parte "gráfica"), a classe que interpreta os inputs do utilizador para interagir com a interface, e duas classes que correpondem à Thread de escrita para o sistema, e à Thread de leitura de respostas vindas do sistema.

O package **Servidor** corresponde à camada de aplicação do sistema. Este package contém todas as classes que nos permite ler, interpretar e responder aos pedidos do utilizador, assim como interagir com a camada de dados. Para tal temos uma classe para cada entidade relevante ao sistema (ativo,contrato,utilizador,registo), com o objetivo de a representar em objetos. Temos também uma classe que lê e envia respostas ao utilizador (classe *Comunicacao*). Esta classe *Comunicacao* recebe os pedidos do utilizador, e envia a mensagem à classe *ThreadCliente* que interpreta e processa o pedido. Este processamento é suportado pela classe "principal" *ESS_ltd*. Esta tem acesso à camada de dados e quando é invocado um método implementado aqui, ele responde ao pedido depois de aceder aos dados guardados e que podem ser apresentados ao utilizador. Além disto temos as classes que implementam os padrões de design (iremos falar mais à frente).

Por último, o package **DAOS** corresponde à camada de dados. Esta tem uma classe para cada tipo de objeto que queremos guardar na nossa base de dados (no nosso caso é uma base de dados *SQLite*). Cada classe tem implementado os métodos que fazem querys à base de dados e retorna os resultados obtidos. Para poder fazer querys foi necessário a criação de outra classe que trata da conexão com a base de dados em si.

6.2 Design Patterns

No que diz respeito à estratégia de solução, temos de pensar também em **design patterns**. Estes padrões ajudam a solucionar um problema comum, sendo mais fácil a sua resolução, através de uma "linguagem" comum entre Designers de Software.

Assim tivemos de estruturar o nosso pensamento e objetivos do nosso problema, de forma a encontrarmos situações e requisitos onde os padrões fossem justificáveis e benéficos. Neste capítulo vamos falar sobre os padrões escolhidos e como eles se aplicam na nossa solução.

6.2.1 Observer

O padrão **Observer** é um padrão usado quando existe uma dependência de um para muitos entre classes. Com isto sempre que um objeto atualiza o seu estado, todo os objetos dependentes dele são automaticamente avisados e atualizados.

Como funciona?

Para este padrão são necessárias pelo menos duas classes e duas interfaces, sendo que cada classe implementará uma interface. Estass interfaces são denominadas *Subject* e *Observer*.

A interface *Subject* será implementada pela classe que representa o objeto cujo estado vai ser alterado. Já a interface *Observer* vai ser implementada por uma classe que depende (ou usa) o objeto que implementa a outra interface.

Com isso, sempre que o objeto altera o seu estado, a sua interface irá invocar um método da interface do *Observer*. Através disso a(as) classe(classes) que a implementam vão atualizar o seu estado para os novos valores.

Na nossa solução:

Quanto à nossa solução, achamos por bem usar o padrão **Observer** na relação entre *Contratos* e *Ativos*, pois existe uma relação de um para muitos visto que um ativo estará envolvido em mais que um contrato.

A classe *Ativo* será responsável por implementar a interface *Subject*, enquanto que a interface *Observer* será implementada pela classe *Contrato*. Assim sempre que o valor de um ativo é atualizado, a interface do *Subject* chama a interface do *Observer* atualizando assim o valor do ativo nos contratos que contêm esse mesmo ativo.

Em termos práticos, sempre que um utilizador cria um contrato esse mesmo contrato toma o papel de *Observer*. Assim o ativo estará sempre a atualizar o seu próprio estado e consequentemente atualiza o valor nos contratos que o "observam". Quando o valor dos ativos estiver fora do intervalo entre o *stopless* e o *take profit*, definido na criação do contrato, este irá ser fechado automaticamente.

6.2.2 Memento

O padrão **Memento** é um padrão usado para restaurar estados. Este padrão permite, tal como o nome sugere, reverter o estado de um objeto para um momento anterior.

Como funciona?

Para garantirmos as potencialidades deste padrão vamos precisar de três classes. Uma classe é responsável por "ser" o estado atual. Um objeto desta classe tem como principais funções poder atualizar o seu estado ou devolver o seu estado. Esta classe representa o *memento* em si. Outra classe é a classe responsável por gerir todos os estados de um programa. Esta classe consegue guardar estados em objetos *Memento*. Por último temos a classe que armazena os estados numa lista de *Mementos* tendo por isso a possibilidade de restaurar estados anteriores.

Na nosssa solução

Numa análise ao problema que nos foi apresentado surgiu-nos a possibilidade de o servidor poder *crashar* num momento critico para algum utilizador. Assim optamos por usar este padrão para poder permitir que o utilizador efetue a operação que estava a realizar quando ocorreu a falha do servidor.

Com isto foi preciso criar uma nova classe que represente o estado atual do sistema para o cliente. Esta classe *Estado* guarda o pedido(uma string) feito pelo utilizador ao servidor, guarda um booleano

que nos diz se o pedido já foi ou não respondido, o identificador do pedido assim como o identificador do utilizador que fez esse pedido.

Quanto ao padrão propriamente dito irá ser preciso também outras duas novas classes. Uma delas representa o pedido em si e a outra representa o *Memento* própriamente dito. A classe *Pedido* é formada por um estado (objeto da classe *Estado*) e consegue guardar o seu estado atual em forma de *Memento* assim como atualizar o seu próprio estado quando recebe um *Memento*. A classe *Memento* representa o Memento em si, e também é composto por um Estado.

A classe responsável por armazenar todos os estados vai ser a classe do *Utilizador*. Esta classe além da lista de Mementos, tem consigo o Pedido mais recente dele e a posição do último pedido respondido.

6.2.3 DAO

O padrão **DAO** (*Data Acess Object*) é um padrão estrutural que nos permite isolar a camada de negócio(ou de aplicação) da camada de persistência, usando uma API abstrata. Esta API tem como principal funcionalidade esconder da apliação toda a complexidade envolvida nas operações de CRUD. Isto permite às duas camadas evoluirem sem uma saber o estado da outra.

Como funciona?

De forma a podermos separar as camadas anteriormente referidas vamos precisar de uma classe cujo objeto representa algo que queremos guardar na camada de persistência (também conhecida como *classe de domínio*). Esta classe não implementa nenhum comportamento, apenas é usada para manter a informação de um objeto coerente. Depois vamos precisar de uma classe onde teremos a interface da nossa API, ou seja, as que contém todas as operações que podemos fazer (desde adicionar objetos, remover, ir buscar ou atualizar). Assim também será necessário uma classe que implementa esta interface estando aqui implementadas todas as operações descritas na interface. Após isso qualquer classe poderá usufruir desta informação instânciando um objeto da classe que implementa a interface.

Na nossa solução

No que toca ao nosso sistema decidimos que iríamos usar DAO's para separar a camada de negócio da camada de persistência que se caracteriza por ser a ponte entre o próprio sistema e a base de dados relacional *SQLite*.

Depois de uma análise concluímos que as informações que queremos guardadas agrupam-se em classes como *Ativos*, *Contratos*, *Pedidos*, *Registos*, *Utilizadore*s, *Seguidores* e *Notificações*, havendo assim uma tabela para cada um destes. Em consequência disso cada uma destas classes vai ter uma outra classe que implementará a interface do respetivo **DAO**.

Na nossa implementação acabamos por ignorar a classe de interface do DAO visto que optamos por usar a interface de um *Map<Integer, Objeto>* onde na nossa camada de implementação da mesma fazemos um @Override de cada método que vamos precisar. Esta decisão veio da necessidade de procurarmos objetos pelo seu id (que será uma key de um map), sendo o nosso map composto por *<idObjeto,Objeto>*.

Na classe principal instanciamos um objeto de cada classe de imlementação de DAO, pois esta é a responsável por gerir todo o sistema. Com isto esta pode responder a qualquer pedido de um utilizador e sempre que precisar de alguma informação que não está guardado em memória, esta pedirá à classe de *ObjetoDAO* que esta comunicará com a base de dados e devolverá o pretendido.

6.2.4 Thread Pool

Como funciona

Uma threadPool mantém um conjuntos de threads à espera de tarefas para serem executadas com o supervisionamento do programa.Esta circunstância aumenta a perfomance e evita a latência da frequente criação e destruição de threads para realização de tarefas.Promove a reutilização de threads.

6.2.5 Na nossa solução

Para evitar o overhead de memória através da criação de um thread por cada conexão de um cliente ao Servidor foi implementado este padrão. Assim é possível ter um número indeterminado de clientes conetados ao servidor sem que este fique comprometido.Com este padão também tambem evitámos que uma thread fique bloqueada à espera de informação proveninente do cliente.Deste modo a comunicação entre cliente e Servidor é assincrona, o que se revela uma vantagem visto que as threads presentes no Servidor deixam de ficar à espera de informação. As classes AsynchronousServerSocketChannel, AsynchronousSocketChannel, AsynchronousChannelGroup e AsynchronousServerSocketChannel presentes no pacote java NIO contribuiram para a implementação do padrão.

7 Modelação

Nesta secção vamos falar sobre a modelação e opções tomadas ao longo da realização do projeto.

7.1 Modelo de Domínio

O Modelo de Domínio captura as entidades relevantes para o sistema e a relação entre eles.

7.1.1 Diagrama de Modelo de Domínio

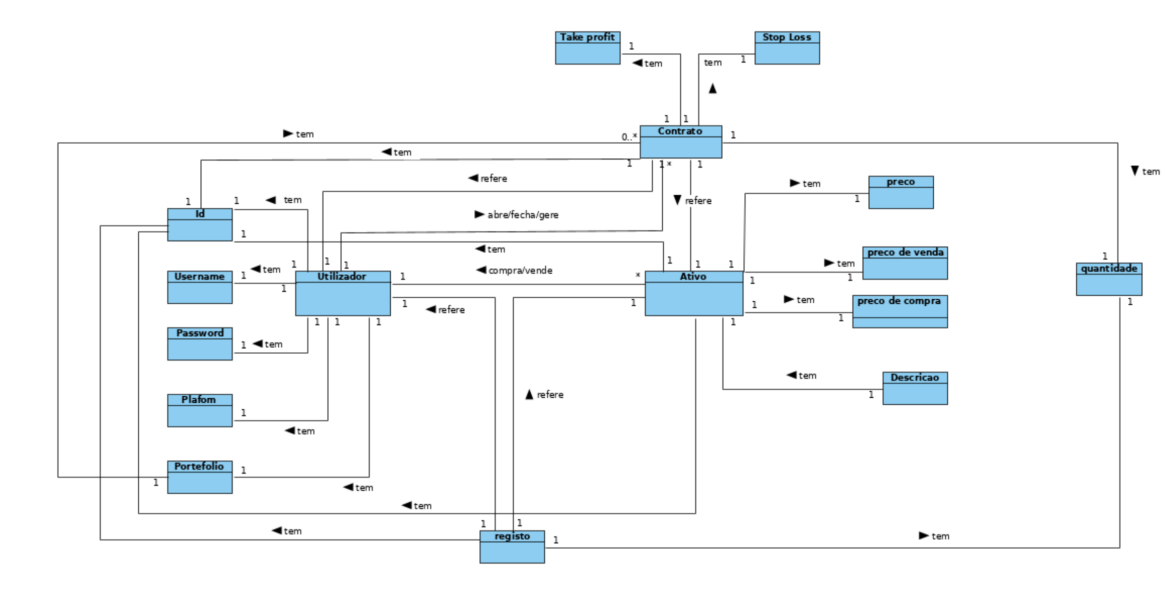


Figure 4: Modelo de Domínio

7.1.2 Entidades Relevantes

Ativo

Esta entidade representa um **ativo** em si. Sendo uma plataforma de Tradding, esta entidade vai ter um id(único) que diz respeito a um ativo, assim como preço de venda e de compra e ainda a descrição.

Utilizador

Esta entidade representa um utilizador (**trader**) da plataforma. Esta entidade também terá associado um id(único), um username e uma password, um planfom inicial e todo o portefólio de contratos efetuados.

Registo

Esta entidade representa o histórico de contratos fechados. Esta entidade refere-se às entidades envolvidas num contrato (um **utilizador** e um **ativo**). Além dos id's destas duas entidades, teremos também um id próprio, a quantidade e o lucro obtido.

Contrato

Por último temos a entidade **Contrato** que é o foco de toda a plataforma. Cada contrato refere-se à posição tomada de um **Utilizador** em relação a um **Ativo**. Esta posição é caracterizada pelo tipo de

contrato tomado (compra ou venda), assim como preco, lucro máximo e perda máxima, a quantidade e o estado da mesma.

7.2 Use Cases

Os Use Cases da nossa plataforma é uma forma de representarmos as principais funcionalidades que queremos implementar no sistema. Assim foi necessária uma análise ao enunciado do problema e depois de alguma pesquisa sobre o funcionamento geral de uma plataforma de Tradding foi possível construir este diagrama.

7.2.1 Diagrama de Modelo de Use Cases

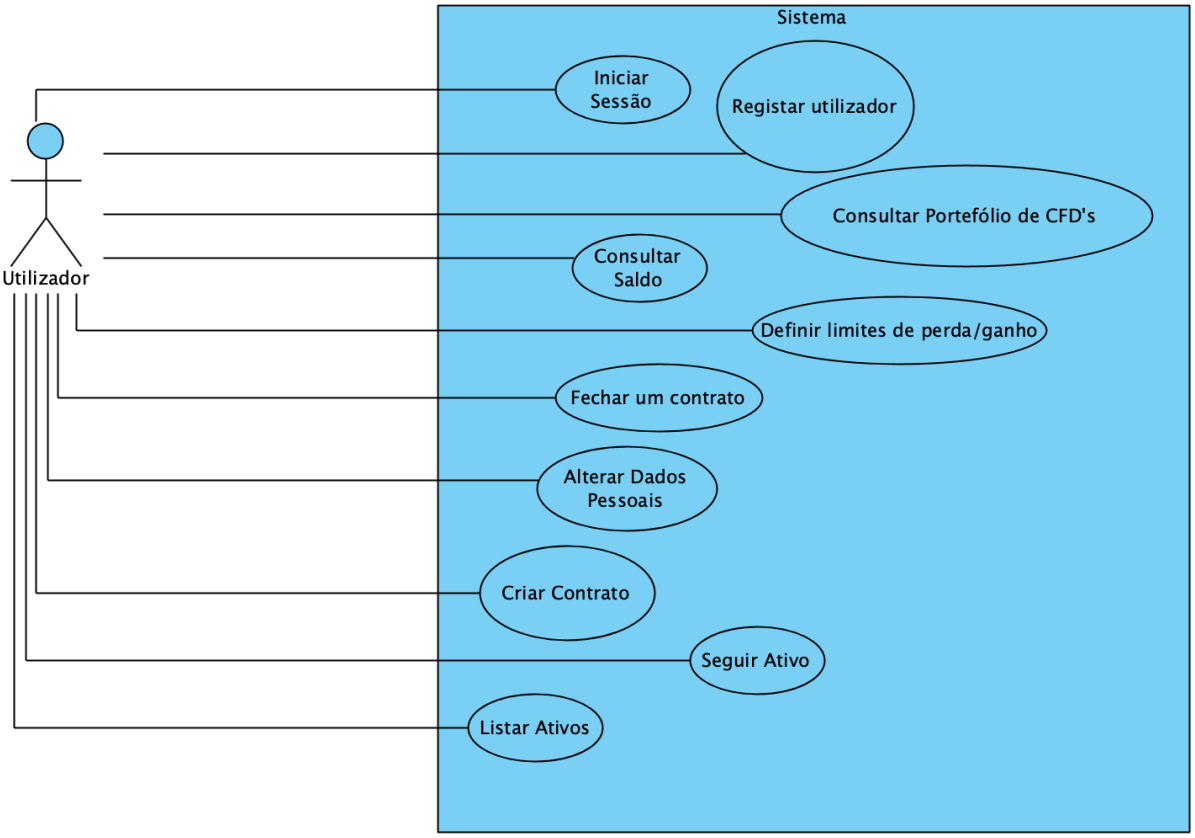


Figure 5: Diagrama de Modelo de Use Cases

7.3 Diagramas de Comportamento

Para representar o comportamento de algumas funcionalidades decidimos usar os diagramas de sequência mais detalhadamente(sub-sistemas e implementação).

7.3.1

7.3.2 Fechar Contrato

Esta funcionalidade representa quando o utilizador quer fechar um contrato, quer seja de venda ou de compra.

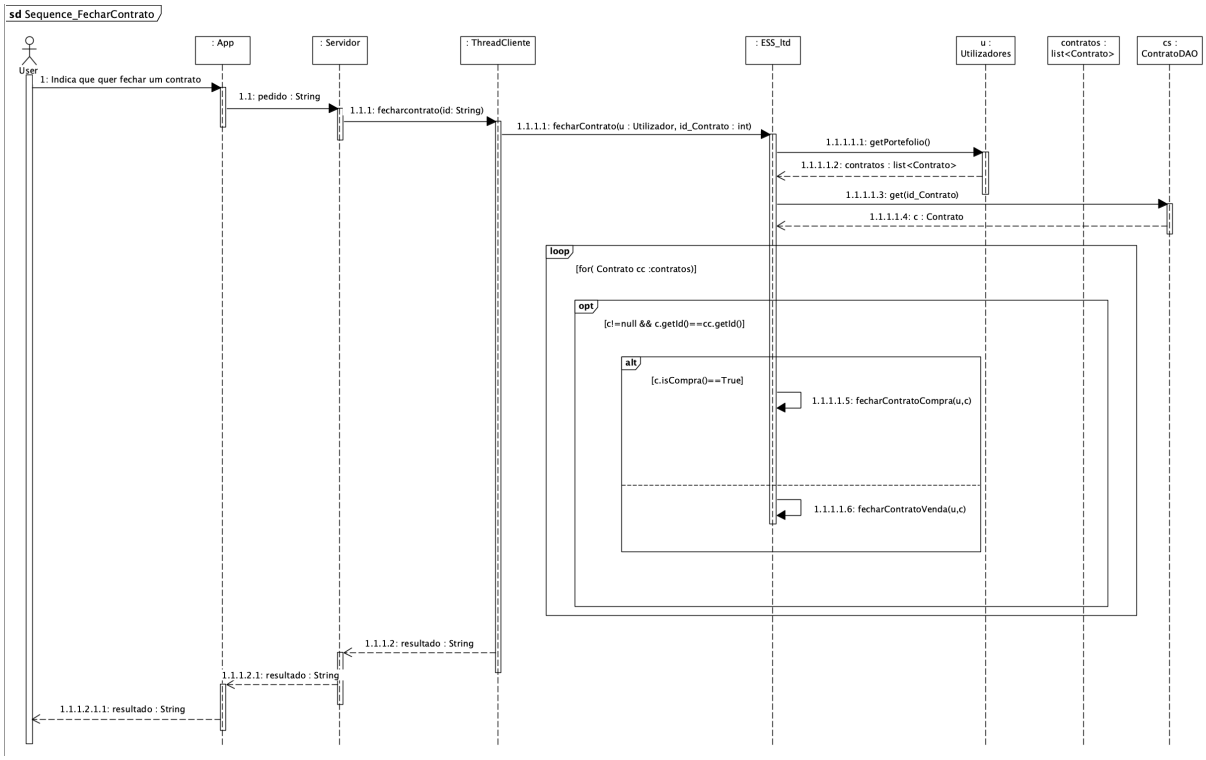


Figure 6: Diagrama de Sub-sistemas de Fechar Contrato

7.3.3 Criar Ativo

Embora esta funcionalidade seja executada em back-end sem que o utilizador tenha controlo sobre ela, achamos por bem explicar e mostrar como essa funcionalidade funciona. Assim representamo-la da seguinte maneira:

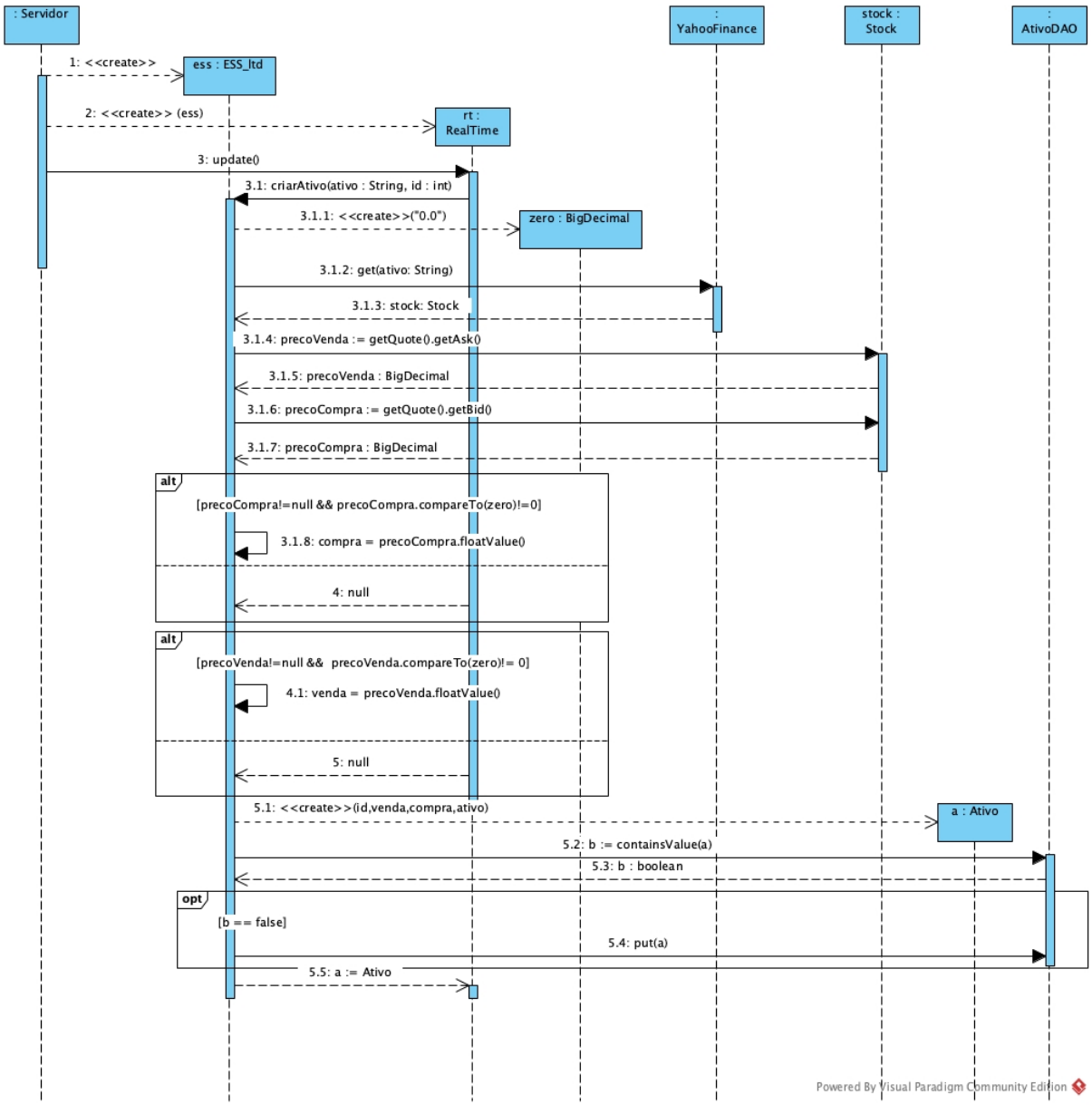


Figure 7: Diagrama de Implementação Criar Ativo

7.3.4 Update dos Valores do Ativo

Este diagrama representa a operação de um objeto ativo atualizar os seus valores (de compra e de venda):

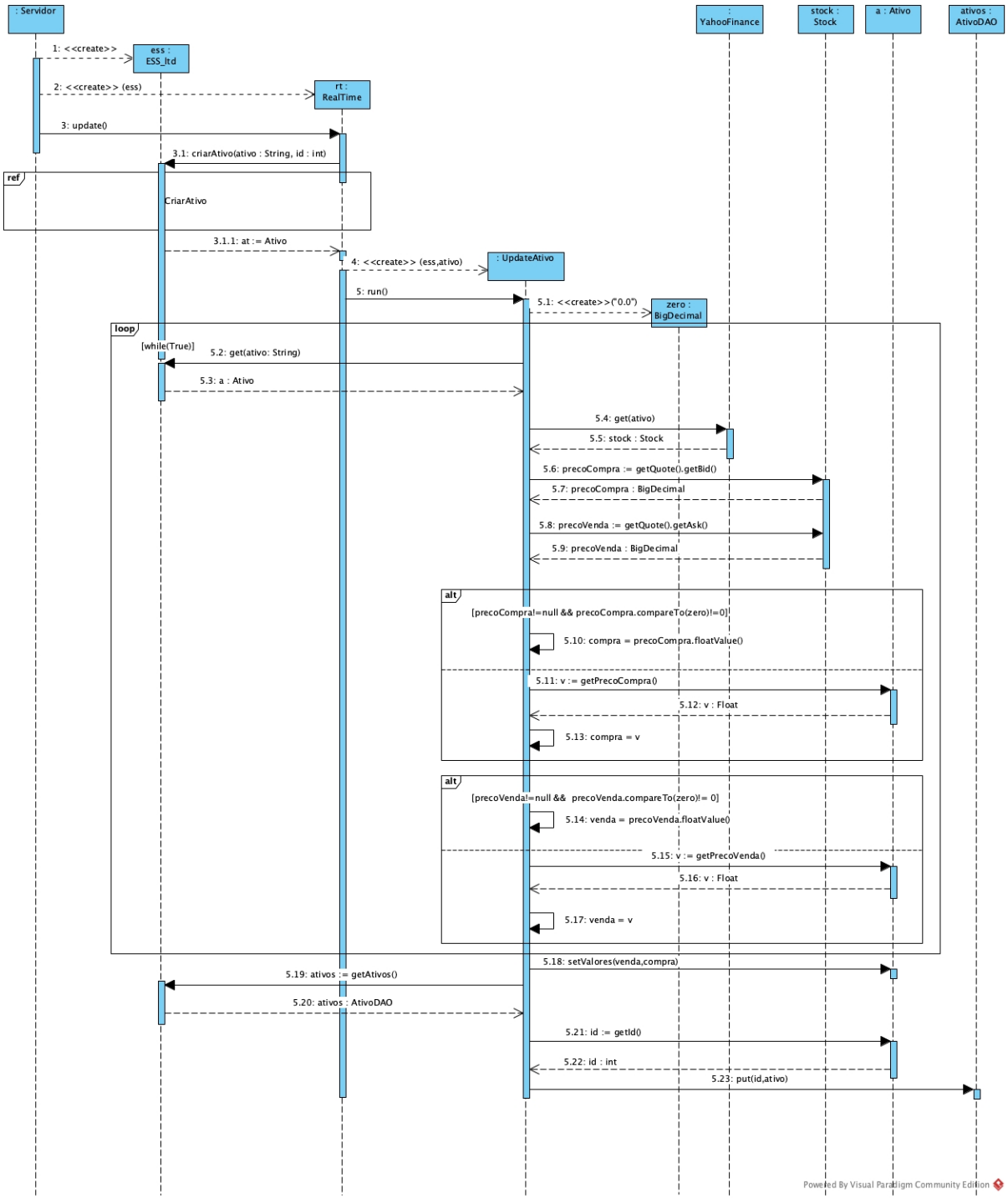


Figure 8: Diagrama de Implementação de Update de um valor de um ativo

7.3.5 Set Valores de um ativo

Este diagrama representa a operação de um objeto ativo fazer set dos seus novos valores(de compra e de venda). Este diagrama completa o anterior e achamos que devíamos representa-lo para mostrar como funciona o mecanismo do padrão Observer.

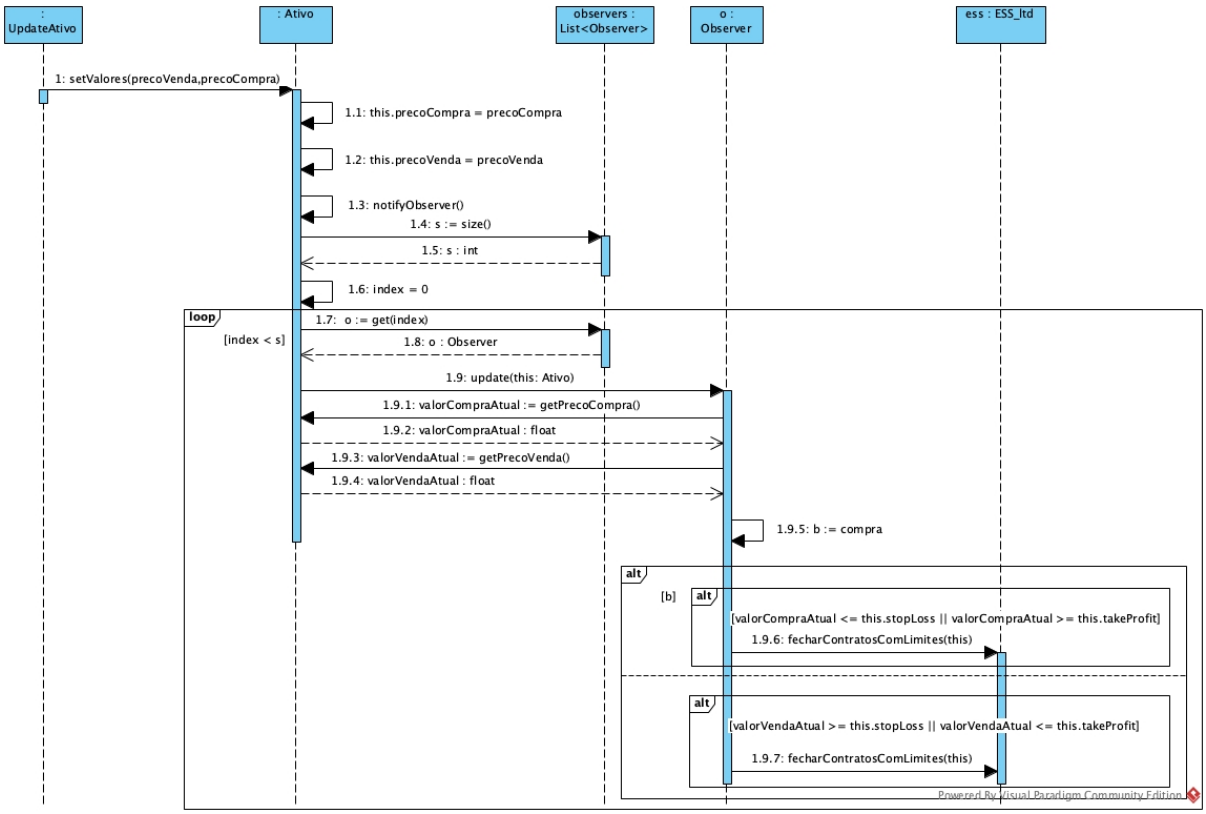


Figure 9: Diagrama de Implementação de Set de um valor de um ativo

7.4 Diagrama de Classes

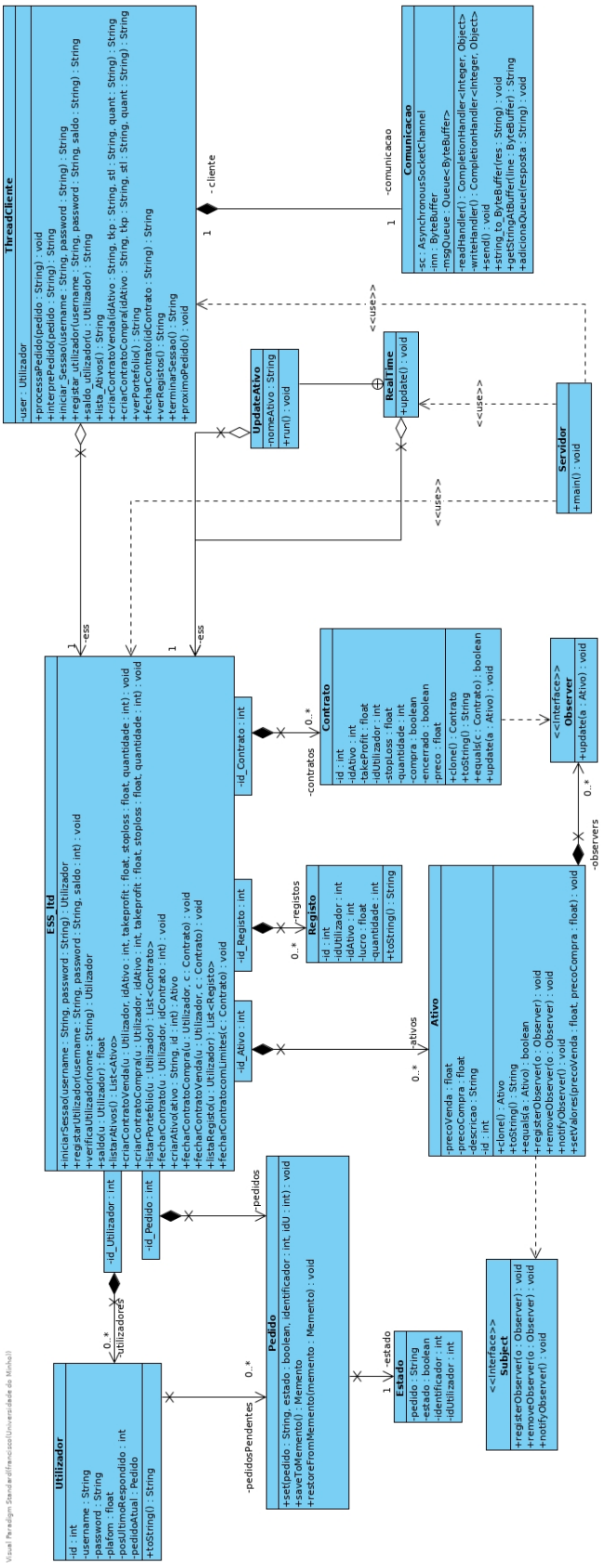


Figure 10: Diagrama de Classes

7.5 Diagrama de ORM

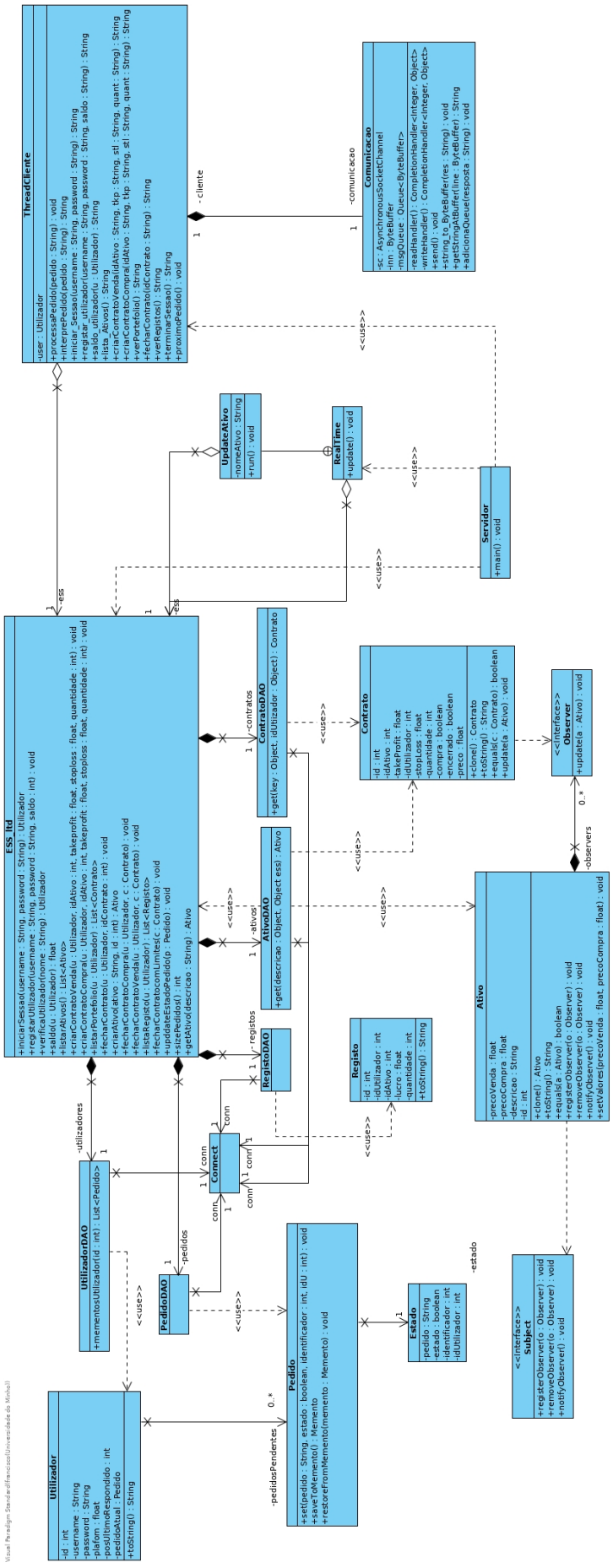


Figure 11: Diagrama de ORM

7.6 Diagrama de Packages

O diagrama de packages representa todos os pacotes (packages) necessários para o sistema e como interagem.

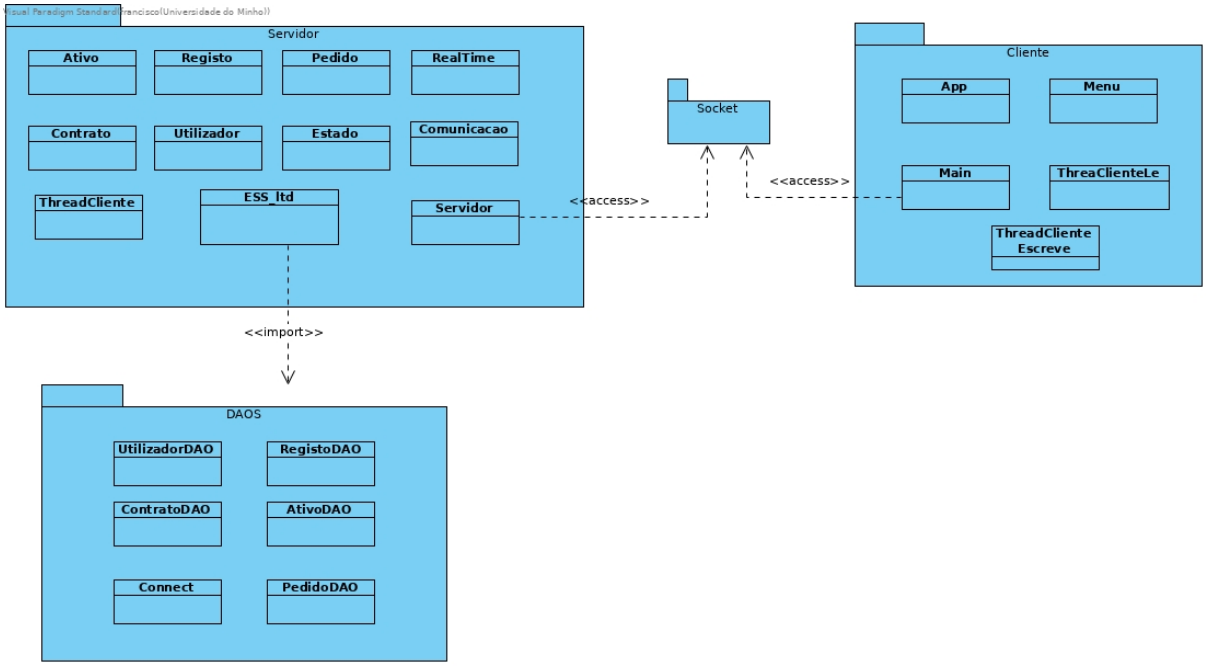


Figure 12: Diagrama de Packages

7.7 Diagrama de Instalação

Neste diagrama vemos o que é necessário para a aplicação correr corretamente na nossa máquina.

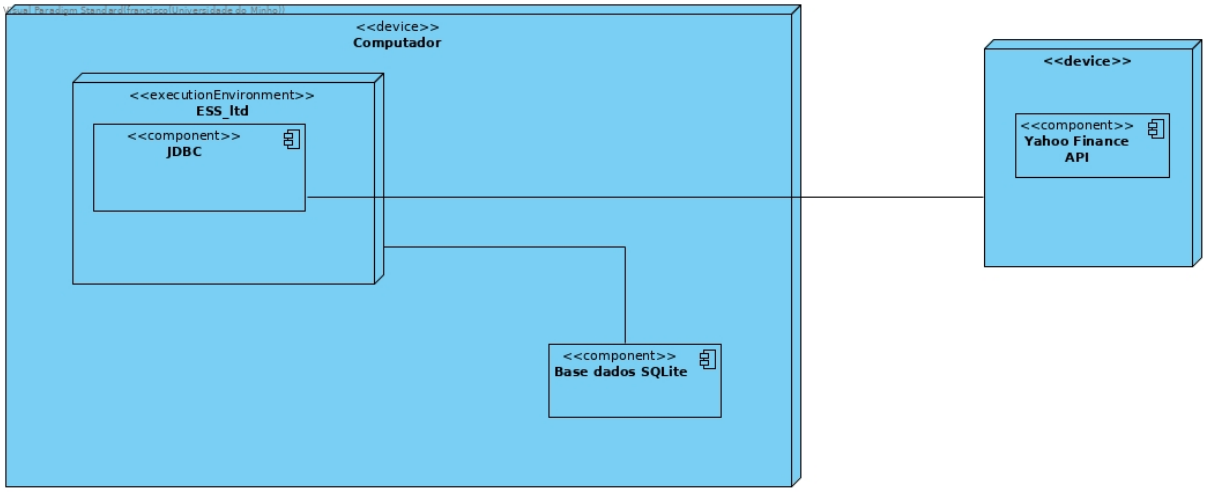


Figure 13: Diagrama de Instalação

8 Novo Requisito

- Permitir aos "traders" seguir a cotação de um conjunto ativos, sendo notificados quando um desses ativos sofre uma variação significativa no seu valor.

8.1 Alterações

Servidor

ESSltd

Foram acrescentados três métodos, o método *seguirAtivo* que associa um utilzador a um ativo, o método *novaNotificação* que tem como função criar uma notificação para um determinado utilizador referente a um ativo, quando este sofre uma variação significativa no seu valor, e por último, o método *veNotificacoes* que permite ler as notificações de um determinado utilizador.

ThreadCliente

No método *intrepretaPedido* que é responsável por interpretar e redirecionar o pedido para o método certo, foi adicionada a opção de poder ser recebida a mensagem com o pedido de seguir um ativo.

Além disso foi criado o método *seguirAtivo*, que recebe um pedido "seguir ativo" e invoca o método *seguirAtivo* implementao na classe **ESSltd**.

Utilizador

O utilizador, tal como o contrato, passou a implementar a interface *Observer* adotando assim um papel de observador. Este implementa o método de *update* que, quando chamado, verifica se houve uma alteração significativa no valor do ativo. Se sim é criada uma nova notificação. Também foi adicionada uma variável de instância, Map<Integer,Ativo>, para guardar a informação referente aos ativos que o utilizador está a seguir.

Cliente

Menu

Foi necessário alterar a interface apresentada ao utilizador, acrescentando o botão para a nova funcionalidade.

App

Neste classe foi adicionado o método *seguirAtivo*, que cria o pedido da nova funcionalidade e envia-o para o Servidor.

DAOS

Foi necessário alterar os DAOs, nomedamente o AtivoDAO e o UtilzadorDAO, onde foram adicionados novos métodos e outros sofreram pequenas alterações.

9 Interface

Nesta fase final optamos por manter a interface da fase anterior, que é bastante simples e baseada no terminal. Para tal é necessário correrms o servidor num processo de terminal e o(os) clientes(clientes) noutro(s).

Quando corremos a aplicação deparamo-nos com este menu:

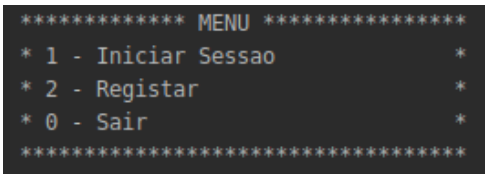


Figure 14: Menu inicial

Depois do login ser efetuado temos um menu principal que é onde podemos usufruir de todas as funcionalidades implementadas:

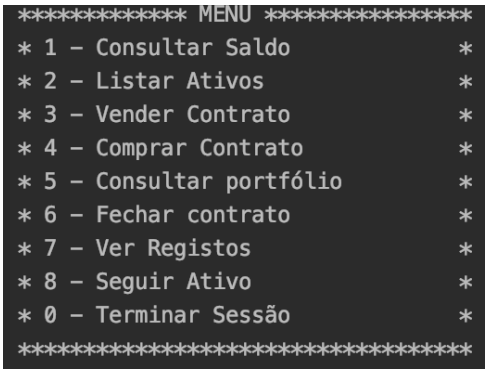


Figure 15: Menu principal

10 Conclusão

Nesta fase final do projeto melhoramos a nossa arquitetura e aplicamos padrões de design. Após concluída a primeira fase, ao longo do novo conhecimento adquirido nas aulas fomos alterando a nossa solução, de forma a aplicar padrões que se encaixavam no nosso sistema e que nos ajudavam a garantir os atributos de qualidade que definimos.

Temos noção que poderíamos fazer ainda mais algumas melhorias no que toca às funcionalidades. Um exemplo disso era relativa à nova funcionalidade, onde poderíamos permitir ao utilizador deixar de seguir um ativo também. Poderíamos também enviar uma notificação ao utilizador sempre que um contrato com limites definidos fechasse devido ao novo valor do ativo em questão.

Assim concluímos que este trabalho prático foi concluído com sucesso e que estamos conscientes do trabalho que fizemos e os conhecimentos que consolidamos. Realçamos também que agora temos mais consciência da importância de ter uma boa arquitetura e um bom uso de padrões de design, quer para simplificar a nossa solução quer para facilitar futuras alterações ao sistema. De facto notamos uma diferença bastante considerável a implementações anteriores, onde a nossa preocupação era apenas ter tudo funcional, ignorando toda a estruturação devida. Com isto saímos daqui muito satisfeitos com o nosso processo de evolução.