



Universidade do Minho
Escola de Engenharia

MESTRADO EM
ENGENHARIA INFORMÁTICA
1º ANO | 1º SEMESTRE

ARQUITETURAS DE SOFTWARE

Relatório de Projeto 2º Fase



Francisco Saraiva | PG39287

Ano Letivo 2019/2020
Braga, 15 de Janeiro 2020

Conteúdos

Introdução	3
1. Análise	4
2. Code Smells	5
3. Refactoring	6
3.1. CFDController	6
3.2. AtivoCreator	7
3.3. AuthController	9
3.4. Favorito	10
3.5. Mercado	11
3.6. Portfolio	12
3.7. APIConexao	13
3.8. FavoritoController	15
3.9. MercadoController	16
3.10. UtilizadorController	17
3.11. Ativo	18
3.12. CFD	19
3.13. CFDDAO	20
3.14. DBConexao	21
3.15. FavoritoDAO	21
3.16. UtilizadorDAO	22
4. Métricas e análise pós refactoring	24

Ilustrações

Figura 1 - Janela principal da aplicação	4
Figura 2 - Janela de ativos	4
Figura 3 - CFDDController antes do refactor	6
Figura 4 - CFDDController depois do refactor.....	6
Figura 5 - Comentários de testes presentes.....	7
Figura 6 - Main antigo antes das mudanças ao AtivoCreator e APIConexao	7
Figura 7 - Novo Main do programa.....	7
Figura 8 - Nova classe abstrata AtivoCreator	8
Figura 9 - Exemplo do AtivoCreatorAcao	8
Figura 10 - AuthController antes do refactoring	9
Figura 11 - AuthController depois do refactoring.....	9
Figura 12 - Favorito antes do refactoring.....	10
Figura 13 - Favorito depois do refactoring	10
Figura 14 - Construtor vazio e comentários	11
Figura 15 - getAtivoPorNome antes do refactoring.....	11
Figura 16 - Mercado depois do refactoring.....	11
Figura 17 - Portfolio antes do refactoring.....	12
Figura 18 - Portfolio depois do refactoring	12
Figura 19 - APIConexao antes do refactoring.....	13
Figura 20 - APIConexao como abstrata.....	13
Figura 21 - Polimorfismo da conexão	14
Figura 22 - FavoritoControlelr antes do refactoring	15
Figura 23 - FavoritoController depois do refactoring	15
Figura 24 - MercadoController antes do refactoring	16
Figura 25 - MercadoController depois do refactoring	16
Figura 26 - UtilizadorController antes do refactoring.....	17
Figura 27 - UtilizadorController depois do refactoring	17
Figura 28 - Ativo antes do refactoring	18
Figura 29 - Ativo depois do refactoring	18
Figura 30 - Construtor novo do CFD	19
Figura 31 - CFDDAO antes do refactoring	20
Figura 32 - CFDDAO depois do refactoring	20
Figura 33 - DBConexao antes do refactoring.....	21
Figura 34 - DBConexao depois do refactoring.....	21
Figura 35 - FavoritoDAO antes do refactoring.....	21
Figura 36 - FavoritoDAO depois do refactoring.....	22
Figura 37 - UtilizadorDAO antes do refactoring	22
Figura 38 - UtilizadorDAO depois do refactoring.....	23
Figura 39 - Lines of Code do projeto original	24
Figura 40 - Lines of Code do projeto refactor	24
Figura 41 - Complexidade do projeto original.....	25
Figura 42 - Complexidade do projeto refactor	25

Introdução

Este documento serve de suporte e detalha a análise efetuada e os aspetos técnicos do refactoring de um projeto Java feito por alunos como solução da plataforma de troca de CFDs para a componente curricular de projeto da cadeira de arquiteturas de software.

Nesta entrega da 2ª fase foi proposto um projeto anónimo a cada grupo de alunos para encontrarem code smells e efetuar refactoring no código de maneira a melhorar os projetos, com uma avaliação das métricas e impacto no programa. Elaboração do trabalho passou por três fases: a de identificação e exploração do projeto e catalogamento dos code smells, de aplicações de técnicas de refactoring, melhoramento e limpeza do código, e comparação de métricas e análise do impacto do melhoramento ao programa.

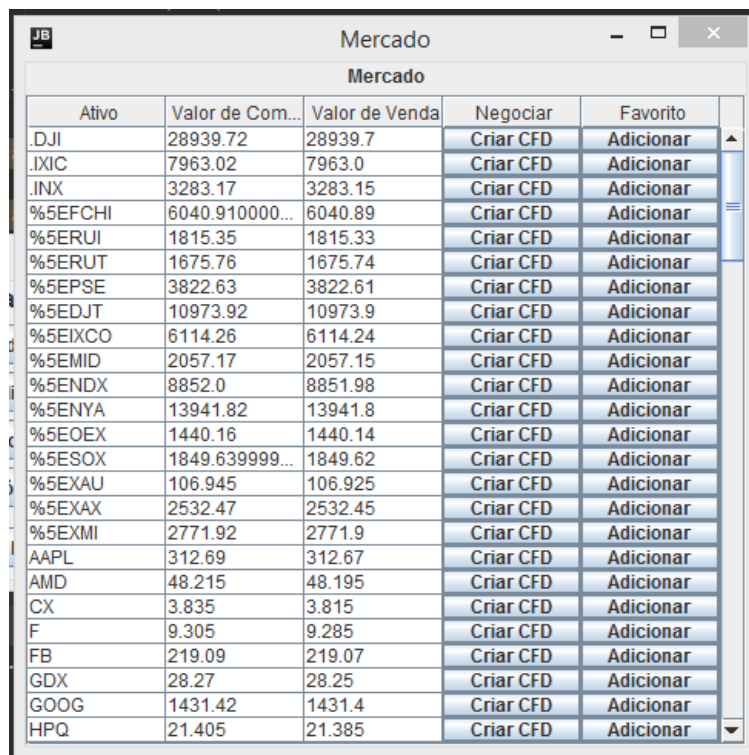
1. Análise

O projeto recebido foi um projeto Java numa arquitetura **MVC**, com camada de serviços e camada de persistência. O projeto apresenta boa estrutura e aplicação de **single responsibility principle**, abstração das classes e o seu funcionamento independente e relação entre classes.

Algumas classes apresentam métodos vazios e incompletos como por exemplo as classes responsáveis pela persistência dos dados da aplicação.



Figura 1 - Janela principal da aplicação



Ativo	Valor de Com...	Valor de Venda	Negociar	Favorito
.DJI	28939.72	28939.7	Criar CFD	Adicionar
.IXIC	7963.02	7963.0	Criar CFD	Adicionar
.INX	3283.17	3283.15	Criar CFD	Adicionar
%5EFCHI	6040.910000...	6040.89	Criar CFD	Adicionar
%5ERUI	1815.35	1815.33	Criar CFD	Adicionar
%5ERUT	1675.76	1675.74	Criar CFD	Adicionar
%5EPSE	3822.63	3822.61	Criar CFD	Adicionar
%5EDJT	10973.92	10973.9	Criar CFD	Adicionar
%5EIXCO	6114.26	6114.24	Criar CFD	Adicionar
%5EMID	2057.17	2057.15	Criar CFD	Adicionar
%5ENDX	8852.0	8851.98	Criar CFD	Adicionar
%5ENYA	13941.82	13941.8	Criar CFD	Adicionar
%5EOEX	1440.16	1440.14	Criar CFD	Adicionar
%5ESOX	1849.639999...	1849.62	Criar CFD	Adicionar
%5EXAU	106.945	106.925	Criar CFD	Adicionar
%5EXAX	2532.47	2532.45	Criar CFD	Adicionar
%5EXMI	2771.92	2771.9	Criar CFD	Adicionar
AAPL	312.69	312.67	Criar CFD	Adicionar
AMD	48.215	48.195	Criar CFD	Adicionar
CX	3.835	3.815	Criar CFD	Adicionar
F	9.305	9.285	Criar CFD	Adicionar
FB	219.09	219.07	Criar CFD	Adicionar
GDX	28.27	28.25	Criar CFD	Adicionar
GOOG	1431.42	1431.4	Criar CFD	Adicionar
HPQ	21.405	21.385	Criar CFD	Adicionar

Figura 2 - Janela de ativos

Verificando o funcionamento do programa, existem alguns detalhes e funcionamento incorreto dos requisitos à plataforma, como o registo de utilizadores não funciona, é possível adicionar favoritos sem estar autenticado, é possível criar CFDs também sem estar autenticado. A ausência de mensagens de aviso ou erros também não alertam para o que está a acontecer quando a interagir com a interface gráfica do programa.

2. Code Smells

Após uma análise ao código e às várias classes do projeto foram encontradas as seguintes categorias de code smells nas seguintes classes:

- **Bloaters**
 - AtivoCreator
- **OO Abusers**
 - AuthController
 - CFDController
 - Favorito
 - Mercado
 - Portfolio
 - APIConexao
- **Dispensable**
 - CFDController
 - FavoritoController
 - MercadoController
 - UtilizadorController
 - Ativo
 - CFD
 - Favorito
 - Mercado
 - CFDDAO
 - DBConexao
 - FavoritoDAO
 - UtilizadorDAO
 - APIConexao
 - AtivoCreator
- **Couplers**
 - FavoritoController
 - Ativo

Verifica-se em maior número neste projeto smells dispensáveis como comentários não necessários, código que não é usado, imports que nunca são chamados e funções que não oferecem nenhuma funcionalidade ou estão vazias.

3. Refactoring

Dentro das classes identificadas e code smells nelas contidas, as seguintes mudanças foram efetuadas nas seguintes classes do projeto:

3.1. CFDController

Esta classe continha comentários desnecessários e uma atribuição nova de um objeto instanciando um objeto vazio e chamando os vários construtores para passar os diversos valores ao contrário de utilizar o construtor da classe CFD. Muitas variáveis tinham nomes que não declaravam o que representavam ou eram ambíguas.

Como técnicas foram aplicadas limpeza de código, atribuição de valores de métodos a variáveis para melhor leitura e compreensão e simplificação da construção de um CFD.

```
CFD cfd = new CFD(); //@SMELL ooabuser

cfd.setUtilizador(u);
cfd.setAtivo(a);
cfd.setStopLoss(sl);
cfd.setTakeProfit(tp);
cfd.setCompra(isCompra);
cfd.setValorAbertura(a.getValorVenda());
cfd.setQuantidade(unidades);

u.getPortfolio().getCfds().add(cfd);

//cfdDAO.save(cfd) //@SMELL dispensable

return cfd;

public void calcularSLTP(CFD cfd){
    //TODO - FAZER ISTO //@SMELL dispensable
    throw new UnsupportedOperationException();
}
```

Figura 3 - CFDController antes do refactor

```
public CFD getCFD(Utilizador utilizador, String nome){
    return utilizador.getPortfolio().getCFD(nome);
}

public CFD createCFD(Utilizador utilizador, Ativo ativo, double unidades, double stopLoss, double takeProfit, boolean isCompra) {

    double valorAbertura = ativo.getValorVenda();
    CFD cfd = new CFD(ativo, utilizador, unidades, stopLoss, takeProfit, isCompra, valorAbertura);
    utilizador.getPortfolio().getCfds().add(cfd);

    return cfd;
}

public void calcularSLTP(CFD cfd){
    throw new UnsupportedOperationException();
}
```

Figura 4 - CFDController depois do refactor

3.2. AtivoCreator

Responsável pela criação dos ativos e criação de ficheiros JSON esta classe contém bloaters, dispensables e muito código repetido para a mesma função que é ler e criar os vários ficheiros json. Muito comentários de testes presentes foram removidos, mas a maior mudança era a de limpeza do código repetido e repartição dos métodos.

```
while(itr2.hasNext()){
    JSONObject jsonObject = (JSONObject) itr2.next();
    String nome = (String) jsonObject.get("ticker");
    double valorVenda = (Double) jsonObject.get("price");
    double valorCompra = valorVenda + 0.02;
    //System.out.println("Nome: " + nome + " Preço de compra: " + valorCompra + "€ Preço de venda: " + valorVenda + "€"); //@SMELL dispensable
    Moeda moeda = new Moeda(nome, valorCompra, valorVenda);
    Moedas.add(moeda);
}
```

Figura 5 - Comentários de testes presentes

Antes do refactoring a esta classe foi feito o refactoring à classe **APIConexao** onde foi aplicada a técnica de mudança de condição conforme cada conexão feita à API em troca de polimorfismo (Exemplo mais abaixo no capítulo da **APIConexao**).

```
public class Main {

    public static void main(String[] args) {

        AtivoCreator creator = new AtivoCreator();
        Mercado m = creator.createAtivos();
        MercadoController mc = new MercadoController(m);

        Utilizador u1 = new Utilizador();
        UtilizadorController uc = new UtilizadorController(u1);

        new MainUI(uc, mc).initComponents();

    }
}
```

Figura 6 - Main antigo antes das mudanças ao AtivoCreator e APIConexao

```
public class Main {

    public static void main(String[] args) {

        List<Moeda> moedas;
        List<Indice> indices;
        List<Acao> acoes;

        AtivoCreatorMoeda creatorMoeda = new AtivoCreatorMoeda();
        moedas = creatorMoeda.createAtivos();

        AtivoCreatorIndice creatorIndice = new AtivoCreatorIndice();
        indices = creatorIndice.createAtivos();

        AtivoCreatorAcao creatorAcao = new AtivoCreatorAcao();
        acoes = creatorAcao.createAtivos();

        Mercado mercado = new Mercado(indices, acoes, moedas);
        MercadoController mc = new MercadoController(mercado);

        Utilizador utilizador = new Utilizador();
        UtilizadorController uc = new UtilizadorController(utilizador);

        new MainUI(uc, mc).initComponents();

    }
}
```

Figura 7 – Novo Main do programa

Alterando a responsabilidade do mercado criar e chamar os ativos, foi mudado a que cada classe que estende da classe abstrata **AtivoCreator**, cria a sua própria lista de ativos correspondente com as suas próprias definições e propriedades, **AtivoCreatorMoeda** cria as moedas, **AtivoCreatorIndice** cria os índices e **AtivoCreatorAcao** cria as ações.

```
public abstract class AtivoCreator {  
    private final double COTACAO = 0.02;  
  
    public double getCOTACAO(){  
        return this.COTACAO;  
    }  
}
```

Figura 8 - Nova classe abstrata AtivoCreator

```
public class AtivoCreatorAcao extends AtivoCreator {  
    private final String NAMEJSONARRAY = "companiesPriceList";  
  
    public AtivoCreatorAcao() { super(); }  
  
    public List<Acao> createAtivos() {  
        String File = "";  
        APIConexaoAcao apiAcao = new APIConexaoAcao();  
        File = apiAcao.getDados();  
  
        try {  
            List<Acao> Acoes = new ArrayList<>();  
            Object obj = new JSONParser().parse(new FileReader(File));  
            JSONObject json = (JSONObject) obj;  
            JSONArray jsonArray = (JSONArray) json.get(NAMEJSONARRAY);  
            Iterator itr2 = jsonArray.iterator();  
  
            while (itr2.hasNext()) {  
                JSONObject jsonObject = (JSONObject) itr2.next();  
                String nome = (String) jsonObject.get("ticker");  
                double valorVenda = (Double) jsonObject.get("price");  
                double valorCompra = valorVenda + this.getCOTACAO();  
                Acao acao = new Acao(nome, valorCompra, valorVenda);  
                Acoes.add(acao);  
            }  
        }  
    }  
}
```

Figura 9 - Exemplo do AtivoCreatorAcao

Em cada serviço de criação dos ativos chamam as conexões às APIs correspondentes chamando também os serviços apropriados ao ativo que pertencem. Foi assim aplicada a troca das condições ao tipo de ativos a criar em troca de polimorfismo criando vários serviços para cada lista de ativo do mercado, retirando a responsabilidade do serviço também criar um mercado, aplicando a **single responsibility principle**.

3.3. AuthController

Controlador responsável pela interação da autenticação da aplicação, contém Object Orientation Abusers como as lógicas de cálculo de login de admin e a criação do objeto Utilizador. Para refactoring foi limpo o código, variáveis renomeadas, simplificada as expressões de lógica, colocados alguns métodos inline e extraído métodos novos para melhor leitura.

A classe **Utilizador** também levou um construtor novo para esta solução.

```
public Boolean login(String email, String pass) {
    System.out.println("Recebi os dados: ");
    System.out.println("User: " + email);
    System.out.println("Pass: " + pass);

    if(email.equals("admin") && pass.equals("admin")){ //@SMELL ooabuser
        return true;
    }

    return false;
}

public void registar(String nome, String email, String password, double fundos)

    Utilizador u1 = new Utilizador(); //@SMELL ooabuser
    u1.setNome(nome);
    u1.setEmail(email);
    u1.setPassword(password);
    u1.setFundos(fundos);
}
```

Figura 10 - AuthController antes do refactoring

```
private Conta conta;

public Boolean login(String email, String password) {
    mensagemLogin(email, password);
    return email.equals("admin") && password.equals("admin");
}

public void registar(String nome, String email, String password, double fundos) {
    Utilizador u1 = new Utilizador(nome, email, password, fundos);
}

public void mensagemLogin(String email, String password){
    System.out.println("Recebi os dados: ");
    System.out.println("User: " + email);
    System.out.println("Pass: " + password);
}

public void logout() {
    System.out.println("Logout feito com sucesso");
    System.out.println("A sair .... ");
    System.exit(status: 0);
}
```

Figura 11 - AuthController depois do refactoring

3.4. Favorito

Classe representativa dos ativos favoritos no projeto. Contém um Objecto Orientation abuser e dispensables. Foi efetuada limpeza de código e simplificada a expressão lógica do **getAtivo**.

```
public Ativo getAtivo(String nome){
    for(int i = 0; i <= favoritos.size(); i++){
        if(favoritos.get(i).getNome().compareTo(nome) == 0){ //@SMELL ooabuser
            return favoritos.get(i);
        }
    }
    return null;
}

@Override
public void update() {
    //@SMELL dispensable
}
```

Figura 12 - Favorito antes do refactoring

```
public Ativo getAtivo(String nome){
    for(Ativo ativo: favoritos){
        if(ativo.getNome().equals(nome))
            return ativo;
    }
    return null;
}

@Override
public void update() {
    throw new UnsupportedOperationException();
}
```

Figura 13 - Favorito depois do refactoring

3.5. Mercado

Classe responsável pelo mercado no projeto, contém dispensables, object orientation abusers e código repetido. Foi feita limpeza de código, simplificação dos métodos e lógicas e removido o construtor vazio que não era usado mais comentários de teste.

Técnica utilizada para resolver o **getAtivoPorNome** foi de substituir as condições em cláusulas de proteção.

```
//private List<Commodity> commodities; //@SMELL dispensable

public Mercado(){
}
```

Figura 14 - Construtor vazio e comentários

```
//@SMELL dispensable
for(int i = 0; i < indices.size(); i++){
    if(nomeAtivo.compareTo(indices.get(i).getNome()) == 0){ //@SMELL ooabuser
        return indices.get(i);
    }
}

for(int i = 0; i < acoes.size(); i++){
    if(nomeAtivo.compareTo(acoes.get(i).getNome()) == 0){ //@SMELL ooabuser
        return acoes.get(i);
    }
}

for(int i = 0; i < moedas.size(); i++){
    if(nomeAtivo.compareTo(moedas.get(i).getNome()) == 0){ //@SMELL ooabuser
        return moedas.get(i);
    }
}

return null;
```

Figura 15 - getAtivoPorNome antes do refactoring

```
public Ativo getAtivoPorNome(String nomeAtivo) {
    Ativo indice = getIndicePorNome(nomeAtivo);
    if(indice!=null)
        return indice;
    Ativo acao = getAcaoPorNome(nomeAtivo);
    if(acao!=null)
        return acao;
    Ativo moeda = getMoedaPorNome(nomeAtivo);
    if(moeda!=null)
        return moeda;
    return null;
}

public Ativo getIndicePorNome(String nome){
    for(Indice indice: indices){
        if(indice.getNome().equals(nome))
            return indice;
    }
    return null;
}

public Ativo getAcaoPorNome(String nome){...}

public Ativo getMoedaPorNome(String nome){...}
```

Figura 16 - Mercado depois do refactoring

3.6. Portfolio

Classe que representa o portfolio do utilizador com os seus contratos de diferenças. Contém um object orientation abuser à lógica de procura do nome de um ativo no método **getCFD**. Foi limpo o código e simplificada a expressão.

```
public CFD getCFD(String nome){
    for(int i = 0; i <= cfd.size(); i++){
        if(cfd.get(i).getAtivo().getNome().compareTo(nome) == 0){ //@SMELL ooabuser
            return cfd.get(i);
        }
    }
    return null;
}
```

Figura 17 - Portfolio antes do refactoring

```
public CFD getCFD(String nome){
    for(CFD cfd: cfd) {
        return (cfd.getAtivo().getNome().equals(nome) ? cfd : null);
    }
    return null;
}
```

Figura 18 - Portfolio depois do refactoring

3.7. APIConexao

Classe responsável pela conexão à API de onde o projeto vai buscar os valores dos ativos. Contém dispensables, object orientation abusers e código repetido. Foi efetuada limpeza no código, removido código desnecessário. Os endereços foram colocados em constantes.

Foi necessária mudar o condicionalismo da leitura da API para usar polimorfismo tornado a classe uma classe abstrata, com as classes individuais, **APIConexaoMoeda**, **APIConexaoAcao**, **APIConexaoIndex**.

```
public APIConexao(){
}

public String getDados(String tipo) { //@SMELL ooabuser
    switch(tipo) {
        case "Moedas":
            return moedas();
        case "Ações":
            return acoes();
        case "Index":
            return index();
        default:
            return "";
    }
}

private String moedas() { //@SMELL dispensable

    try {
        String CryptoJson = "";
        URL url = new URL( spec: "https://financialmodelingprep.com/api/v3/cryptocurrencies");
        BufferedReader reader = new BufferedReader(new InputStreamReader(url.openStream(), charsetName: "UTF-8"));
        for (String line; (line = reader.readLine()) != null; ) {
            CryptoJson = CryptoJson.concat(line + "\n");
        }

        FileWriter fw = new FileWriter(new File( pathname: "crypto.json"));
        fw.write(CryptoJson);
        fw.close();
    }
}
```

Figura 19 - APIConexao antes do refactoring

```
public abstract class APIConexao {
    public APIConexao(){
    }
    abstract String getDados();
}
```

Figura 20 - APIConexao como abstrata

```
public class APIConexaoMoeda extends APIConexao {

    private String URL = "https://financialmodelingprep.com/api/v3/cryptocurrencies";

    public APIConexaoMoeda() { super(); }

    public String getDados(){
        try {
            String CryptoJson = "";
            URL url = new URL(URL);
            BufferedReader reader = new BufferedReader(new InputStreamReader(url.openStream(), charsetName: "UTF-8"));
            for (String line; (line = reader.readLine()) != null; ) {
                CryptoJson = CryptoJson.concat(line + "\n");
            }

            FileWriter fw = new FileWriter(new File( pathname: "crypto.json"));
            fw.write(CryptoJson);
            fw.close();

            return "crypto.json";
        } catch (IOException e) {
            e.printStackTrace();
        }
        return "crypto.json";
    }
}
```

Figura 21 - Polimorfismo da conexão

3.8. FavoritoController

Classe responsável pelo controlo do modelo favorito, contém um dispensable em comentário e um coupler no encadeamento de chamadas de métodos nos objetos bem como variáveis que precisam de nomes mais descritivos.

Foi feita limpeza de código e dispersão dos métodos em variáveis para melhor leitura.

```
public void calcularVariacao(Ativo a){
    //TODO - CALCULAR A VARIACÃO //@SMELL dispensable
    throw new UnsupportedOperationException();
}

public Favorito getFavoritos(Utilizador u) { return u.getFavorito(); }

public void apagarFavorito(Utilizador u, Ativo a){
    u.getFavorito().apagaFavorito(a); //@SMELL coupler
}
```

Figura 22 - FavoritoControlelr antes do refactoring

```
public class FavoritoController {

    public void calcularVariacao(Ativo a){
        throw new UnsupportedOperationException();
    }

    public Favorito getFavoritos(Utilizador utilizador){
        return utilizador.getFavorito();
    }

    public void apagarFavorito(Utilizador utilizador, Ativo ativo){
        Favorito favorito = utilizador.getFavorito();
        favorito.apagaFavorito(ativo);
    }

}
```

Figura 23 - FavoritoController depois do refactoring

3.9. MercadoController

Classe responsável pelo controlo do modelo Mercado, contém dispensables como imports não usados, código repetido e variáveis com nomes não descritivos.

Foi feita limpeza de código, organização dos dados, imports e variáveis e separação dos métodos.

```
public class MercadoController {  
  
    private Mercado m;  
  
    public MercadoController(Mercado m) { this.m = m; }  
  
    public Mercado getM() { return m; }  
    public Ativo getAtivo(String nome) { return m.getAtivoPorNome(nome); }  
  
    public List<Acao> listarAcoes() { return m.getAcoes(); }  
  
    public List<Moeda> listarMoedas() { return m.getMoedas(); }  
  
    public List<Indice> listarIndices() { return m.getIndices(); }  
  
    /**  
     * método bootstrap para popular a plataforma  
     */  
    public List<Ativo> getMercadoCompleto() {  
  
        List<Ativo> mercado = new ArrayList<>();  
  
        //@SMELL dispensable  
        mercado.addAll(m.getIndices());  
        mercado.addAll(m.getAcoes());  
        mercado.addAll(m.getMoedas());  
  
        return mercado;  
    }  
}
```

Figura 24 - MercadoController antes do refactoring

```
public class MercadoController {  
  
    private Mercado mercado;  
  
    public MercadoController(Mercado mercado) { this.mercado = mercado; }  
  
    public Mercado getMercado() { return mercado; }  
  
    public Ativo getAtivo(String nome) { return mercado.getAtivoPorNome(nome); }  
  
    public List<Acao> listarAcoes() { return mercado.getAcoes(); }  
  
    public List<Moeda> listarMoedas() { return mercado.getMoedas(); }  
  
    public List<Indice> listarIndices() { return mercado.getIndices(); }  
  
    /**  
     * método bootstrap para popular a plataforma  
     */  
    public List<Ativo> getMercadoCompleto() {  
        List<Ativo> mercadoNovo = new ArrayList<>();  
        adicionarMercados(mercadoNovo);  
        return mercadoNovo;  
    }  
  
    private void adicionarMercados(List<Ativo> mercadoNovo){  
        mercadoNovo.addAll(mercado.getIndices());  
        mercadoNovo.addAll(mercado.getAcoes());  
        mercadoNovo.addAll(mercado.getMoedas());  
    }  
}
```

Figura 25 - MercadoController depois do refactoring

3.10. UtilizadorController

Classe responsável pelo controlo do modelo Utilizador, contém dispensables como comentários, variáveis com nomes não descritivos e couplers com encadeamento de chamadas a métodos de objetos.

Foi limpo o código, mudado o nome às variáveis e separado o encadeamento dos métodos em variáveis auxiliares para melhor leitura.

```
public class UtilizadorController {  
  
    private Utilizador u;  
  
    public UtilizadorController(Utilizador u) { this.u = u; }  
  
    public Utilizador getU() { return u; }  
  
    public void adicionarFundos(double quantia) { u.setFundos(u.getFundos() + quantia); } |  
  
    public void atualizarFundos() {  
        // TODO - implement UtilizadorController.atualizarFundos //@SMELL dispensable  
        throw new UnsupportedOperationException();  
    }  
  
    public double getFundos() { return u.getFundos(); }  
}
```

Figura 26 - UtilizadorController antes do refactoring

```
public class UtilizadorController {  
  
    private Utilizador utilizador;  
  
    public UtilizadorController(Utilizador utilizador) { this.utilizador = utilizador; }  
  
    public Utilizador getUtilizador() { return utilizador; }  
  
    public void adicionarFundos(double quantia) {  
        double novosFundos = calcularNovosFundos(quantia);  
        utilizador.setFundos(novosFundos);  
    }  
  
    private double calcularNovosFundos(double quantia){  
        return utilizador.getFundos() + quantia;  
    }  
  
    public void atualizarFundos() {  
        throw new UnsupportedOperationException();  
    }  
  
    public double getFundos() { return utilizador.getFundos(); }  
}
```

Figura 27 - UtilizadorController depois do refactoring

3.11. Ativo

Classe abstrata modelo do Ativo em que se criam os contratos de diferença no projeto. Esta classe contém dispensables em comentários desnecessários e um coupler no override do método **ToString** em que a concatenação da string está complexa e difícil de ler.

Foi limpo o código, removido os comentários desnecessários e arranjado o método **ToString** do Ativo.

```
public void getState() {
    // TODO - implement Ativo.getState //@SMELL dispensable
    throw new UnsupportedOperationException();
}

public void setState() {
    // TODO - implement Ativo.setState //@SMELL dispensable
    throw new UnsupportedOperationException();
}

@Override
public String toString() {
    //@SMELL coupler
    return "Ativo{" +
        "nome='" + nome + '\'' +
        ", valorCompra=" + valorCompra +
        ", valorVenda=" + valorVenda +
        '}';
}
```

Figura 28 - Ativo antes do refactoring

```
public void getState() { throw new UnsupportedOperationException(); }

public void setState() { throw new UnsupportedOperationException(); }

@Override
public String toString() {
    String head = ">> Ativo <<\n";
    String nome = "Nome: "+getNome()+"\n";
    String valorCompra = "Valor Compra: "+getValorCompra()+"\n";
    String valorVenda = "Valor Venda: "+getValorVenda()+"\n";
    String ativo = head+nome+valorCompra+valorVenda;
    return ativo;
}
```

Figura 29 - Ativo depois do refactoring

3.12. CFD

Classe modelo CFD para representação do contrato de diferenças só apresenta um comentário por apagar. No refactoring da classe **CFDController** já foi criado um novo construtor que era necessário à melhor estruturação e criação do objeto no controlador.

Foi limpo o código nesta classe e limpo comentários.

```
public CFD(Ativo ativo, Utilizador utilizador, double quantidade, double stopLoss, double takeProfit, boolean isCompra, double valorAbertura) {  
    this.ativo = ativo;  
    this.utilizador = utilizador;  
    this.quantidade = quantidade;  
    this.stopLoss = stopLoss;  
    this.takeProfit = takeProfit;  
    this.isCompra = isCompra;  
    this.valorAbertura = valorAbertura;  
}
```

Figura 30 - Construtor novo do CFD

3.13. CFDDAO

A classe CFDDAO responsável pela persistência e DAO do CFD não está completa nem funcional, o que tem dispensables como comentários e métodos vazios sem código dentro.

Foi limpo o código e adicionadas exceções no chamamento dos métodos.

```
public class CFDDAO implements DAO {  
  
    public CFD get(int id) {  
        // TODO - implement CFDDAO.get //@SMELL dispensable  
        throw new UnsupportedOperationException();  
    }  
  
    public List<CFD> getAll() {  
        // TODO - implement CFDDAO.getAll //@SMELL dispensable  
        throw new UnsupportedOperationException();  
    }  
  
    @Override  
    public void save(Object o) {  
        //@SMELL dispensable  
    }  
  
    @Override  
    public void update(Object o) {  
        //@SMELL dispensable  
    }  
  
    @Override  
    public void delete(Object o) { System.out.println("cfd deleted"); }
```

Figura 31 - CFDDAO antes do refactoring

```
public class CFDDAO implements DAO {  
  
    public CFD get(int id) {  
        throw new UnsupportedOperationException();  
    }  
  
    public List<CFD> getAll() {  
        throw new UnsupportedOperationException();  
    }  
  
    @Override  
    public void save(Object o) {  
        throw new UnsupportedOperationException();  
    }  
  
    @Override  
    public void update(Object o) {  
        throw new UnsupportedOperationException();  
    }  
  
    @Override  
    public void delete(Object o) { System.out.println("cfd deleted"); }
```

Figura 32 - CFDDAO depois do refactoring

3.14. DBConexao

Classe responsável pela conexão a uma base de dados para persistência não está completa, pelo que apresenta um dispensable em forma de comentário.

```
public class DBConexao {  
  
    private DBConexao conexao;  
  
    public DBConexao getInstance() {  
        // TODO - implement DBConexao.getInstance //@SMELL dispensable  
        throw new UnsupportedOperationException();  
    }  
}
```

Figura 33 - DBConexao antes do refactoring

```
public class DBConexao {  
  
    private DBConexao conexao;  
  
    public DBConexao getInstance() {  
        throw new UnsupportedOperationException();  
    }  
}
```

Figura 34 - DBConexao depois do refactoring

3.15. FavoritoDAO

Classe responsável pelo DAO do Favorito não se encontra totalmente implementado pelo que só contém dispensables como comentários e métodos vazios.

```
public class FavoritoDAO implements DAO {  
    @Override  
    public Object get(int id) { return null; }  
  
    @Override  
    public List getAll() { return null; }  
  
    @Override  
    public void save(Object o) {  
        //@SMELL dispensable  
    }  
  
    @Override  
    public void update(Object o) {  
        //@SMELL dispensable  
    }  
  
    @Override  
    public void delete(Object o) { System.out.println("fav deleted"); }  
}
```

Figura 35 - FavoritoDAO antes do refactoring

```
public class FavoritoDAO implements DAO {  
    @Override  
    public Object get(int id) { return null; }  
  
    @Override  
    public List getAll() { return null; }  
  
    @Override  
    public void save(Object o) {  
        throw new UnsupportedOperationException();  
    }  
  
    @Override  
    public void update(Object o) {  
        throw new UnsupportedOperationException();  
    }  
  
    @Override  
    public void delete(Object o) { System.out.println("fav deleted"); }
```

Figura 36 - FavoritoDAO depois do refactoring

3.16. UtilizadorDAO

Classe responsável pelo DAO do Utilizador não se encontra totalmente implementado pelo que só contém dispensables como comentários e métodos vazios.

```
public class UtilizadorDAO implements DAO {  
  
    public Utilizador get(int id) {  
        // TODO - implement UtilizadorDAO.get //@SMELL dispensable  
        throw new UnsupportedOperationException();  
    }  
  
    public List<Utilizador> getAll() {  
        // TODO - implement UtilizadorDAO.getAll //@SMELL dispensable  
        throw new UnsupportedOperationException();  
    }  
  
    @Override  
    public void save(Object o) {  
        //@SMELL dispensable  
    }  
  
    @Override  
    public void update(Object o) {  
        //@SMELL dispensable  
    }  
  
    @Override  
    public void delete(Object o) {  
        //@SMELL dispensable  
    }  
}
```

Figura 37 - UtilizadorDAO antes do refactoring

```
public class UtilizadorDAO implements DAO {  
  
    public Utilizador get(int id) {  
        throw new UnsupportedOperationException();  
    }  
  
    public List<Utilizador> getAll() {  
        throw new UnsupportedOperationException();  
    }  
  
    @Override  
    public void save(Object o) {  
        throw new UnsupportedOperationException();  
    }  
  
    @Override  
    public void update(Object o) {  
        throw new UnsupportedOperationException();  
    }  
  
    @Override  
    public void delete(Object o) {  
        throw new UnsupportedOperationException();  
    }  
}
```

Figura 38 - UtilizadorDAO depois do refactoring

4. Métricas e análise pós refactoring

O projeto recebido para a segunda fase da componente prática apresentava uma boa estrutura na organização e arquitetura do projeto, mas com muito código por implementar, métodos longos e responsabilidades invertidas ou mal organizadas em algumas classes, de notar maior complexidade no package **Service**. Para analisar as mudanças pegou-se em duas métricas:

- Lines of Code (LOC)
- Complexidade Ciclomática

Com o auxílio do plugin para IntelliJ [Metrics Reloaded](#) podemos retirar métricas de lines of code e a complexidade de cada package e comparar os valores antes e após refactor.

package	LOC	LOC(rec)	LOCp	LOCp(rec)	LOCt	LOCt(rec)
	17	1 892	17	1 892	0	0
Controllers	140	140	140	140	0	0
Models	402	402	402	402	0	0
Persistence	107	107	107	107	0	0
Service	177	177	177	177	0	0
Views	1 049	1 049	1 049	1 049	0	0
Total	1 892		1 892		0	
Average	315,33		315,33		0,00	

Figura 39 - Lines of Code do projeto original

package	LOC	LOC(rec)	LOCp	LOCp(rec)	LOCt	LOCt(rec)
	25	1 960	25	1 960	0	0
Controllers	133	133	133	133	0	0
Models	413	413	413	413	0	0
Persistence	102	102	102	102	0	0
Service	240	240	240	240	0	0
Views	1 047	1 047	1 047	1 047	0	0
Total	1 960		1 960		0	
Average	326,67		326,67		0,00	

Figura 40 - Lines of Code do projeto refactor

Comparando as linhas de código de cada projeto podemos ver que para os pacotes **Controllers**, **Persistence** e **Views** o código encurtou, enquanto que nos pacotes **Models** e **Service** as linhas aumentaram devido ao refactoring efetuado para melhorar a performance, estrutura e reduzir a complexidade do projeto.

package ▲	v(G)avg	v(G)tot
	1,00	1
Controllers	1,09	25
Models	1,15	98
Persistence	1,00	16
Service	3,44	31
Views	1,51	56
Total		227
Average	1,33	37,83

Figura 41 - Complexidade do projeto original

package ▲	v(G)avg	v(G)tot
	1,00	1
Controllers	1,04	27
Models	1,18	104
Persistence	1,00	16
Service	2,29	32
Views	1,51	56
Total		236
Average	1,30	39,33

Figura 42 - Complexidade do projeto refactor

Comparando a métrica de complexidade verificamos uma pequena subida em **Models** devido à implementação dos construtores, uma descida nos **Controllers** mas uma acentuada descida no package **Service** em que foi efetuado o refactoring de polimorfismo para separar as tarefas dos **AtivoCreator** e **APIConexao**, reduzindo de 3,44 para 2,29 a complexidade do package. Assim em média reduziu-se a complexidade do projeto de 1,33 para 1,30 de média.