



Universidade do Minho - Escola de Engenharia

Relatório do trabalho prático de Arquiteturas de Software

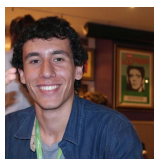
Refactoring - BetESS

Autores :

Diana Costa (A78985)



Marco Silva(A79607)



Versão 1.0
9 de Janeiro de 2019

Resumo

Neste relatório será feita uma abordagem ao segundo trabalho prático de Arquiteturas de Software, ao qual está associado o estudo aprofundado sobre *code smells*, técnicas de *refactoring* e a sua aplicabilidade. O primeiro projeto da UC (plataforma de apostas "BetESS", sem qualquer padrão de design/arquitetura) servirá de base para este estudo.

Conteúdo

1	Introdução	3
2	Contextualização	4
3	Code Smell + Técnica Refactoring	5
3.1	Bloaters	5
3.1.1	Long Method	5
3.1.1.1	Classe BetESS - Construtor	5
3.1.1.2	Classe BetESS - Método <i>fechaEvento()</i>	7
3.1.1.3	Classe AreaAdministrador - Método <i>permiteLiga()</i>	10
3.1.1.4	Classe AreaAdministrador/AreaCliente - Método <i>mostraPainel()</i>	12
3.1.1.5	Classe AreaAdministrador - Método <i>verificaDados()</i>	13
3.1.2	Long Parameter List	14
3.1.2.1	Classe Aposta - Método <i>clone()</i> e Construtores	14
3.1.2.2	Classe Database - geral	17
3.1.2.3	Classe BetESS - geral	18
3.1.2.4	Classe EventoDesportivo - Método clone e Construtores	19
3.1.2.5	Classe Jogador - Método clone e Construtor	22
3.2	Object-Orientation Abusers	24
3.2.1	Switch Statements	24
3.2.2	Classe BetESS - Método <i>fechaEvento()</i>	24
3.3	Dispensables	25
3.3.1	Data Class	25
3.3.1.1	Classe Aposta	25
3.3.1.2	Classe EventoDesportivo	27
3.3.1.3	Classe Jogador	29
3.3.2	Dead Code	30
3.3.2.1	Classe Aposta	30
3.3.2.2	Classe Equipa	31
3.3.2.3	Classe EventoDesportivo	32
3.3.2.4	Classe Liga	33
3.3.2.5	Classe Notificação	34
3.4	Couplers	35
3.4.1	Feature Envy	35
3.4.1.1	Classe BetESS - Método <i>fechaEvento()</i>	35
3.4.2	Message Chains	36
3.4.2.1	Classe BetESS - Método <i>fechaEvento()</i>	36
4	Métricas de avaliação do Refactoring	37
4.1	Testes	37
4.2	Tempos de execução	39
5	Conclusões e Sugestões	40

1 Introdução

Este trabalho prático foi elaborado no âmbito da Unidade Curricular de Arquiteturas de Software, do 4º ano do Mestrado Integrado em Engenharia Informática, com vista à deteção e correção de *code smells* através de técnicas de *refactoring*. Utilizou-se, para elaboração deste projeto, a aplicação desenvolvida no primeiro trabalho prático da unidade curricular (BetESS), aplicando-se o *refactoring* à versão sem qualquer padrão de design/arquitetura, uma vez que o grupo esperaria encontrar mais imperfeições nesta versão. Depois da aplicação das técnicas, é esperada uma melhoria da sua estrutura interna.

Assim, neste relatório, pode-se constatar o trabalho desenvolvido pelo grupo referente a três fases: uma em que se procedeu à deteção dos *code smells*, na qual o grupo percorreu com atenção todas as classes da aplicação BetESS, outra fase, em que se procedeu à correção das imperfeições encontradas, e, por fim, a elaboração/análise de testes (pré-alterações/pós-alterações) a todas as correções elaboradas. Todas estas fases implicaram a tomada de decisões pelo grupo, que envolveram, muitas vezes, um compromisso entre soluções de forma a conseguir um resultado final melhor no geral.

Desta forma, o relatório começa por contextualizar o problema, abordando o conceito, tipos e vantagens do *refactoring*, assim como as diferentes possibilidades de *code smells*, como forma de justificação para o trabalho posterior e secções seguintes. De seguida, apresentam-se as imperfeições encontradas e as respetivas soluções. Nesta secção, é mostrado, em concreto, o antes e depois do trabalho elaborado pelo grupo. Por fim, indicam-se todos os testes executados, quer por prova de conceito quer manuais, tal como uma análise de tempos de execução da aplicação antiga e modificada. O relatório termina com as conclusões, em que o grupo faz uma análise do trabalho desenvolvido, tendo em conta as dificuldades obtidas.

2 Contextualização

O *refactoring* é uma técnica disciplinada para reestruturação de código existente, alterando a sua estrutura interna sem alterar o seu comportamento externo. Assim, a estrutura interna do código é melhorada, sem qualquer modificação nas funcionalidades externas, transformando funções/métodos e repensando algoritmos. Este processo é iterativo, o que é vantajoso na medida em que pequenas soluções simples, no seu conjunto, são capazes de alterar o programa. As melhorias que advêm desta técnica são várias, mas passam sobretudo por melhorar o design do software, fazer com que o software seja fácil de entender e que esteja limpo e mais fácil para deteção de bugs.

Introduzem-se, de maneira geral, os diferentes tipos de *code smells* existentes, de forma a entender os tipos de erros que podem surgir neste trabalho prático.

- **Bloaters** - código, métodos ou classes que aumentaram para proporções gigantes, com as quais é difícil de trabalhar. Normalmente, estes *smells* não são logo detetados, mas acumulam-se com o tempo à medida que o programa evolui;
- **Object-Orientation Abusers** - todos estes *smells* são aplicações incorretas ou incompletas de princípios de programação orientada a objetos;
- **Change Preventers** - estes *smells* significam que, se for necessário alterar algo em qualquer lugar do código, também serão necessárias muitas outras alterações noutros lugares. Como resultado, o desenvolvimento de programas torna-se muito mais perigoso e caro;
- **Dispensables** - algo sem sentido e desnecessário, cuja ausência tornaria o código mais limpo, eficiente e mais fácil de entender;
- **Couplers** - os *smells* pertencentes a este grupo contribuem para o acoplamento excessivo entre as classes, ou mostram o que acontece se o acoplamento for substituído pela delegação excessiva.

De forma a resolver todas as imperfeições no código listadas acima, surgem as técnicas de *refactoring*. Dado que há uma vasta gama de técnicas, mostram-se alguns exemplos de técnicas com as quais o grupo lidou e que aparecerão nas restantes secções.

- **Extract Method** - Mover um pedaço de código para um método/função separado, e substituir o código antigo pela chamada do método;
- **Replace Temp with Query** - Mover uma expressão inteira para um método separado, e retornar o resultado da função a partir disto, isto é, consultar o método em vez de usar uma variável;
- **Preserve Whole Object** - Quando se obtêm vários valores de um objeto, e depois os mesmos valores são passados como parâmetros a um método, é preferível passar o objeto na sua totalidade;
- **Introduce Parameter Object** - Quando um método contém um grupo repetido de variáveis, substituir estes parâmetros pelo objeto;
- **Encapsulate Field** - Tornar um campo público privado, e criar métodos para o aceder;
- **Move Method** - Quando um método é mais usado noutra classe do que na sua própria, cria-se um novo método na classe em que é mais usado. Pode-se remover totalmente o método da classe onde se encontrava, ou transformá-lo numa referência no código original para o novo método;
- **Decompose Conditional** - Decompor expressões condicionais complexas para diferentes métodos.

3 Code Smell + Técnica Refactoring

3.1 Bloaters

Nesta secção apresentam-se os bloaters encontrados ao longo das classes, assim como a sua resolução e vantagens.

3.1.1 Long Method

Um long method é um método que contém demasiadas linhas de código. Geralmente, qualquer método com mais de dez linhas deve fazer com que se comecem a levantar perguntas. Com a correção deste tipo de *code smells*, conseguem-se obter classes que perduram/vivem mais tempo, já que quanto mais longo é um método, mais difícil se torna de entender e de manter. Para além disto, métodos longos são o sítio perfeito para esconder código duplicado.

3.1.1.1 Classe BetESS - Construtor

Como é possível observar, a versão inicial do construtor contém bastante instruções o que torna este método de difícil leitura.

```
public BetESS(){
    Database d = null;
    try {
        FileInputStream fileIn = new FileInputStream("/tmp/database");
        ObjectInputStream in = new ObjectInputStream(fileIn);
        d = (Database) in.readObject();
        in.close();
        fileIn.close();
    } catch (IOException i) {
        i.printStackTrace();
    } catch (ClassNotFoundException c) {
        System.out.println("Database class not found");
        c.printStackTrace();
    }
    if (d == null){
        System.out.println("Estado da aplicação iniciado.");
        this.database = new Database();
    }
    else {
        System.out.println("Restauro da aplicação com sucesso.");
        this.database = d;
    }
}
```

Figura 1: Código original do construtor BetESS.

A melhor ação a tomar será isolar todas as instruções que estão relacionadas com a leitura do objeto de dados e reconstrução do mesmo. Isto passa pela utilização da técnica de *refactoring* "Extract Method". Desta forma, surgiu o novo método *readFile*, juntamente com o novo construtor do objeto BetESS.

```

public Database readfile(Database d){
    try {
        FileInputStream fileIn = new FileInputStream("/tmp/database");
        ObjectInputStream in = new ObjectInputStream(fileIn);
        d = (Database) in.readObject();
        in.close();
        fileIn.close();
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
    return d;
}

```

Figura 2: Código do novo método responsável pela leitura do objeto de dados.

```

public BetESS(){
    Database d = null;
    d = readfile(d);
    if (d == null){
        System.out.println("Estado da aplicação iniciado.");
        this.database = new Database();
    }
    else {
        System.out.println("Restauro da aplicação com sucesso.");
        this.database = d;
    }
}

```

Figura 3: Código do novo construtor do objeto BetESS.

3.1.1.2 Classe BetESS - Método *fechaEvento()*

Nesta secção irão ser discutidas todas as decisões tomadas no âmbito do processo de *refactoring* bem como descritas todas as fases intermédias até à obtenção da versão final. Observe-se, então, a versão inicial do método *fechaEvento*.

```
public void fechaEvento(int id_Evento, boolean ganha_casa, boolean ganha_fora, boolean empate){
    EventoDesportivo e = this.database.getEventoDesportivo(id_Evento);
    e.setGanha_casa(ganha_casa);
    e.setGanha_fora(ganha_fora);
    e.setEmpate(empate);

    for (Aposta a : this.database.getApostasEvento(e.getId_evento())){
        if ( e != null && e.getEstado().equals("Aberto")){
            boolean evento_ganha_casa = e.getGanha_casa();
            boolean evento_ganha_fora = e.getGanha_fora();
            boolean evento_empate = e.getEmpate();

            boolean aposta_ganha_casa = a.getGanha_casa();
            boolean aposta_ganha_fora = a.getGanha_fora();
            boolean aposta_empate = a.getEmpate();

            Jogador j = this.database.checkUser(a.getId_jogador());
            double saldo = j.getSaldo();
            double quant_aposta = a.getQuantia();
            double odd = -10000;
            double saldo_ant = saldo;

            if (evento_ganha_casa == aposta_ganha_casa
                && evento_ganha_fora == aposta_ganha_fora
                && evento_empate == aposta_empate)
            {
                if (evento_ganha_casa){
                    odd = e.getOdd_casa();
                    saldo += odd * quant_aposta;
                }
                else if (evento_ganha_fora){
                    odd = e.getOdd_fora();
                    saldo += odd * quant_aposta;
                }
                else if (evento_empate){
                    odd = e.getOdd_empate();
                    saldo += odd * quant_aposta;
                }
            }

            a.setEstado("Paga");
            this.database.atualizaAposta(a);

            /* lançamento de notificações */
            Notificacao n = new Notificacao(a.getId_aposta(), saldo - saldo_ant);
            Jogador jogador = this.database.checkUser(a.getId_jogador());
            jogador.adicionaNotificacao(n);
            this.database.registaJogador(jogador);

            /* atualização do saldo do cliente */
            this.database.updateSaldo(j.getEmail(), saldo);
        }
    }
    e.setEstado("Terminado");
    this.database.atualizaEventoDesportivo(e);
}
```

Figura 4: Código original do método *fechaEvento()*.

Como se pode verificar, este é um método bastante extenso, sendo aqui executadas um grande número de ações que se poderiam encontrar isoladas.

Devido à dimensão deste método, uma única aplicação da técnica de *refactoring* "Extract Method" não foi suficiente, pelo que de seguida serão descritas as secções de divisão mais gerais do método em questão.

Assim, o processo de fecho de um evento pode ser dividido em **3 grandes fases**: a **extração** dos dados do evento para posterior comparação com as apostas registadas, o **tratamento** das apostas associadas ao evento a fechar e finalmente a **atualização** do seu estado.

Apresenta-se, de seguida, as três fases referidas, associadas a métodos dedicados à execução de tais tarefas:

```
public void fecharEvento(int id_Evento, boolean ganha_casa, boolean ganha_fora, boolean empate){  
    EventoDesportivo e = getEventoDesportivo(id_Evento);  
    e.setResultadoEvento(ganha_casa, ganha_fora, empate);  
  
    trataApostasEvento(e);  
  
    atualizaDadosEvento(e);  
}
```

Figura 5: Declaração das 3 principais etapas para o fecho de um evento desportivo.

Atente-se na **primeira fase** do fecho de um evento desportivo. Como é visível na declaração do método original, o primeiro passo a tomar consiste na inserção do resultado do evento inserido pelo administrador no sistema no objeto evento. Desta forma, cada uma das apostas poderá ser facilmente validada através da informação presente neste objeto. Uma vez que, esta operação de atribuição de informação apenas procede à inserção da informação do resultado da aposta, foi criado um método na classe EventoDesportivo responsável por este carregamento. Segue-se, então, o novo método criado:

```
public void setResultadoEvento(boolean ganha_casa, boolean ganha_fora, boolean empate){  
    this.setGanha_casa(ganha_casa);  
    this.setGanha_fora(ganha_fora);  
    this.setEmpate(empate);  
}
```

Figura 6: Declaração do novo método de carregamento do resultado de uma aposta para o objeto EventoDesportivo.

Uma vez criado o novo método, passa-se para a análise da **segunda fase** do fecho de um evento. Vejamos então o método *trataApostasEvento()*, responsável por toda a lógica de verificação e pagamento das apostas associadas ao evento a ser fechado.

```
public void trataApostasEvento(EventoDesportivo e){  
    for (Aposta a : getApostasEvento(e.getId_evento())){  
        if ( e != null && e.getEstado().equals("Aberto")){  
            Jogador j = checkUser(a.getId_jogador());  
            double saldo = j.getSaldo();  
            double saldo_ant = saldo;  
  
            saldo = trataAposta(e, a, saldo);  
  
            a = atualizaDadosAposta(a);  
  
            /* lançamento de notificações */  
            lancaNotificacao(a, saldo - saldo_ant, j);  
  
            /* atualização do saldo do cliente */  
            updateSaldo(j.getEmail(), saldo);  
        }  
    }  
}
```

Figura 7: Declaração do novo método de tratamento das apostas associadas ao evento desportivo a fechar.

Este método será responsável, assim, por percorrer todas as apostas associadas ao evento desportivo e efetuar a análise das previsões de resultado, bem como lançar as notificações para cada um dos apostadores e, finalmente, atualizar o seu saldo.

Por sua vez, o processo de tratamento das apostas é ainda uma ação com um certo nível de complexidade pelo que será dividido também em métodos de menor complexidade e com responsabilidades distintas. As componentes são o tratamento da aposta e a atualização dos seus dados.

```
public double trataAposta(EventoDesportivo e, Aposta a, double saldo){
    if (verificaAposta(e, a)) {
        if (e.getGanha_casa()){
            saldo = atualizaSaldoAposta(e, a, saldo);
        }
        else if (e.getGanha_fora()){
            saldo = atualizaSaldoAposta(e, a, saldo);
        }
        else if (e.getEmpate()){
            saldo = atualizaSaldoAposta(e, a, saldo);
        }
    }
    return saldo;
}
```

Figura 8: Declaração do novo método de tratamento de uma aposta.

Este método será responsável pelo acesso aos resultados tanto da aposta como do evento em si, e verificar se estes correspondem. Conforme o resultado desta comparação, será retornado o novo valor de saldo do apostador, tendo em conta as *odds* associadas ao evento.

O cálculo do saldo foi também extraído para um novo método *atualizaSaldoAposta*. De seguida apresenta-se a sua declaração.

```
public double atualizaSaldoAposta(EventoDesportivo e, Aposta a, double saldo){
    double odd = e.getOdd_casa();
    return saldo += odd * a.getQuantia();
}
```

Figura 9: Declaração do novo método de cálculo do novo saldo de um apostador.

Como se pode observar, apesar de o cálculo do novo saldo de um apostar não apresentar um elevado nível de complexidade, este isolamento permite uma mais fácil leitura e interpretação do código desenvolvido bem como a posterior reutilização deste módulo.

Neste momento, procedeu-se também à simplificação da condição *if* utilizando a técnica *Decompose Conditional*.

Desta forma, a condição passou a ser implementada num método que irá retornar o valor condicional a utilizar. Apresenta-se então a declaração deste novo método.

```
public boolean verificaAposta(EventoDesportivo e, Aposta a){
    return e.getGanha_casa() == a.getGanha_casa() &&
           e.getGanha_fora() == a.getGanha_fora() &&
           e.getEmpate() == a.getEmpate();
}
```

Figura 10: Declaração do novo método de cálculo da condição *if*.

Uma vez revistos todos os módulos constituintes da segunda fase de fecho de um evento desportivo, procede-se então ao registo das informações alteradas na aposta a tratar no momento.

Assim, foi desenvolvido o seguinte método responsável pela atualização do estado da aposta e consequente registo nas estruturas de dados.

```
public Aposta atualizaDadosAposta(Aposta a){
    a.setEstado("Paga");
    atualizaAposta(a);

    return a;
}
```

Figura 11: Declaração do novo método de registo das alterações feitas na aposta a tratar no momento.

Para que os utilizadores se mantenham atualizados sobre os resultados das suas apostas, são lançadas notificações na altura em que um evento desportivo é fechado. Assim, apresenta-se de seguida o módulo que é responsável pelo lançamento de notificações.

```
public void lancaNotificacao(Aposta a, double balanco, Jogador jogador){
    Notificacao n = new Notificacao(a.getId_aposta(), balanco);
    jogador.adicionaNotificacao(n);
    registaJogador(jogador);
}
```

Figura 12: Declaração do novo método destinado ao lançamento de notificações.

Finalmente, como é possível verificar na imagem 4, é necessário atualizar o saldo do cliente. Uma vez que esta operação se executa em apenas uma linha, não foi necessário qualquer extração adicional.

3.1.1.3 Classe AreaAdministrador - Método *permiteLiga()*

O método executado pelo botão de registo de uma equipa no sistema efetua a verificação se o nome da nova equipa a registar já se encontra registado no mesmo. Desta forma, este processo de verificação pode ser extraído para um novo método, que apenas retornará se o valor é válido ou não.

Assim, apresenta-se então a versão inicial (simplificada à zona onde se efetuou alterações) e final dos métodos modificados utilizando a técnica *extract method*.

```
boolean permite_liga = true;
for (Liga l: ligas){
    if (l.getNome().equals(nome_liga)){
        permite_liga = false;
        break;
    }
}
if (permite_liga){
    Liga l = new Liga(nome_liga);
    this.betess.registaLiga(l);
}
```

Figura 13: Versão inicial (limitada) da validação do nome da equipa a registar.

```
private boolean permite_liga(List<Liga> ligas, String nome_liga){  
    boolean permite_liga = true;  
    for (Liga l: ligas){  
        if (l.getNome().equals(nome_liga)){  
            permite_liga = false;  
            break;  
        }  
    }  
    return permite_liga;  
}
```

Figura 14: Versão final da validação do nome da equipa a registar.

3.1.1.4 Classe AreaAdministrador/AreaCliente - Método *mostraPainel()*

Uma vez que, nas classes representativas das *views*, é necessário remover e adicionar painéis frequentemente, e por forma a mostrar corretamente a informação requisitada pelo utilizador, foi criado um método responsável por efetuar essa ação, evitando assim a repetição de código. Para isso, recorreu-se então à técnica *extract method*.

Atente-se no exemplo de código executado quando premido o botão "Eventos Desportivos" na interface. Na sua versão inicial, este encontrava-se definido da seguinte forma:

```
private void eventos_desportivos_buttonActionPerformed(java.awt.event.ActionEvent evt) {  
    /* remoção de painéis anteriores */  
    options_panel.removeAll();  
    options_panel.repaint();  
    options_panel.revalidate();  
  
    /* alocação do respetivo painel de opções */  
    options_panel.add(eventos_desportivos_elements);  
    options_panel.repaint();  
    options_panel.revalidate();  
  
    DefaultTableModel model = (DefaultTableModel) eventos_lista.getModel();  
    model.setRowCount(0);  
  
    for (EventoDesportivo e : this.betess.getEventosDesportivos().values()){  
        String equipa_casa = this.betess.getEquipa(e.getequipa_casa()).getDesignacao();  
        String equipa_fora = this.betess.getEquipa(e.getequipa_fora()).getDesignacao();  
        model.addRow(new Object[]{e.getId_evento(), equipa_casa, equipa_fora, e.getGanha_casa(), e.getGanha_fora(), e.getEmpate(), e.getEstado()});  
    }  
}
```

Figura 15: Declaração da versão inicial do método executado com o clique no botão "Eventos Desportivos".

Uma vez que esta operação é executada múltiplas vezes nos diferentes botões do menu, esta porção de código foi extraída, ficando assim com o seguinte método e a sua invocação.

```
private void mostraPainel(JPanel c){  
    /* remoção de painéis anteriores */  
    options_panel.removeAll();  
    options_panel.repaint();  
    options_panel.revalidate();  
  
    /* alocação do respetivo painel de opções */  
    options_panel.add(c);  
    options_panel.repaint();  
    options_panel.revalidate();  
}  
  
private void eventos_desportivos_buttonActionPerformed(java.awt.event.ActionEvent evt) {  
    mostraPainel(eventos_desportivos_elements);  
  
    DefaultTableModel model = (DefaultTableModel) eventos_lista.getModel();  
    model.setRowCount(0);  
  
    for (EventoDesportivo e : this.betess.getEventosDesportivos().values()){  
        String equipa_casa = this.betess.getEquipa(e.getequipa_casa()).getDesignacao();  
        String equipa_fora = this.betess.getEquipa(e.getequipa_fora()).getDesignacao();  
        model.addRow(new Object[]{e.getId_evento(), equipa_casa, equipa_fora, e.getGanha_casa(), e.getGanha_fora(), e.getEmpate(), e.getEstado()});  
    }  
}
```

Figura 16: Declaração da versão final dos métodos executados com o clique no botão "Eventos Desportivos".

3.1.1.5 Classe AreaAdministrador - Método *verificaDados()*

O método *verificaDados()* resultou da aplicação da técnica de *refactoring* "Decompose Conditional". A sua aplicação consistiu no isolamento da condição de verificação de todos os campos de dados num método próprio, facilitando assim a leitura do código. Desta forma, observe-se a definição do método original e do modificado, com a alteração da condição *if*.

```
private void regista_evento_buttonActionPerformed(java.awt.event.ActionEvent evt) {  
    if (!combo_casa.getSelectedItem().toString().isEmpty() &&  
        !combo_fora.getSelectedItem().toString().isEmpty() &&  
        !odd_casa_field.getText().isEmpty() &&  
        !odd_fora_field.getText().isEmpty() &&  
        !odd_empate_field.getText().isEmpty()){  
  
        String c_casa = combo_casa.getSelectedItem().toString();  
        String c_fora = combo_fora.getSelectedItem().toString();  
    }  
}
```

Figura 17: Declaração da versão original do método chamado quando o botão é clicado.

```
private boolean verificaDados(){  
    return !combo_casa.getSelectedItem().toString().isEmpty() &&  
        !combo_fora.getSelectedItem().toString().isEmpty() &&  
        !odd_casa_field.getText().isEmpty() &&  
        !odd_fora_field.getText().isEmpty() &&  
        !odd_empate_field.getText().isEmpty();  
}  
  
private void regista_evento_buttonActionPerformed(java.awt.event.ActionEvent evt) {  
    if (verificaDados()){  
        String c_casa = combo_casa.getSelectedItem().toString();  
        String c_fora = combo_fora.getSelectedItem().toString();  
    }  
}
```

Figura 18: Declaração da versão final do método *verificaDados* e sua chamada no método principal.

3.1.2 Long Parameter List

Um método que possua mais de três ou quatro parâmetros, é considerado um long parameter list. Corrigir estas enumerações longas e difíceis de ler traz vantagens como código mais legível e pequeno. Outra possível vantagem é a revelação de código duplicado que passou despercebido.

3.1.2.1 Classe Aposta - Método *clone()* e Construtores

Foram encontrados **três** métodos na classe Aposta que possuíam um Long Parameter List: dois construtores e o método *clone()*:

```
public Aposta clone(){
    return new Aposta(this.getId_aposta(), this.getQuantia(), this.getId_evento(),
                      this.getId_jogador(), this.getGanha_casa(), this.getGanha_fora(),
                      this.getEmpate(), this.getEstado());
}
```

Figura 19: Método *clone()*.

```
public Aposta(int id_aposta, double quantia, int id_evento,
             String id_jogador, boolean ganha_casa,
             boolean ganha_fora, boolean empate) {

    this.id_aposta = id_aposta;
    this.quantia = quantia;
    this.id_evento = id_evento;
    this.id_jogador = id_jogador;
    this.ganha_casa = ganha_casa;
    this.ganha_fora = ganha_fora;
    this.empate = empate;
    this.estado = "Não paga";
}

public Aposta(int id_aposta, double quantia, int id_evento,
             String id_jogador, boolean ganha_casa,
             boolean ganha_fora, boolean empate, String estado) {

    this.id_aposta = id_aposta;
    this.quantia = quantia;
    this.id_evento = id_evento;
    this.id_jogador = id_jogador;
    this.ganha_casa = ganha_casa;
    this.ganha_fora = ganha_fora;
    this.empate = empate;
    this.estado = estado;
}
```

Figura 20: Construtores da classe Aposta.

Para a resolução deste **problema no *clone()***, usou-se a técnica de *refactoring* "Introduce Parameter Object", sendo necessária a construção de um construtor por cópia para a realização do mesmo. Então, a resolução do método *clone()* procedeu-se da seguinte forma:

```
@Override
public Aposta clone(){
    return new Aposta(this);
}
```

Figura 21: Novo *clone()*.

```

public Aposta(Aposta ap){
    this.id_aposta = ap.getId_aposta();
    this.quantia = ap.getQuantia();
    this.id_evento = ap.getId_evento();
    this.id_jogador = ap.getId_jogador();
    this.ganha_casa = ap.getGanha_casa();
    this.ganha_fora = ap.getGanha_fora();
    this.empate = ap.getEmpate();
    this.estado = ap.getEstado();
}

```

Figura 22: Construtor que passa a servir o clone().

Isto fez com que um dos Long Parameter List dos construtores desta classe se resolvesse sozinho, uma vez que o construtor demonstrado abaixo só existia para servir o *clone()*. Tornando-se este construtor um Dead Code, o grupo decidiu removê-lo.

```

public Aposta(int id_aposta, double quantia, int id_evento,
    String id_jogador, boolean ganha_casa, boolean ganha_fora,
    boolean empate) {
    this.id_aposta = id_aposta;
    this.quantia = quantia;
    this.id_evento = id_evento;
    this.id_jogador = id_jogador;
    this.ganha_casa = ganha_casa;
    this.ganha_fora = ganha_fora;
    this.empate = empate;
    this.estado = "Não paga";
}

public Aposta(int id_aposta, double quantia, int id_evento,
    String id_jogador, boolean ganha_casa, boolean ganha_fora,
    boolean empate, String estado) {
    this.id_aposta = id_aposta;
    this.quantia = quantia;
    this.id_evento = id_evento;
    this.id_jogador = id_jogador;
    this.ganha_casa = ganha_casa;
    this.ganha_fora = ganha_fora;
    this.empate = empate;
    this.estado = estado;
}

```

Figura 23: Construtor que servia o clone() antigo (2º construtor).

```

public Aposta(int id_aposta, double quantia, int id_evento,
    String id_jogador, boolean ganha_casa, boolean ganha_fora,
    boolean empate) {
    this.id_aposta = id_aposta;
    this.quantia = quantia;
    this.id_evento = id_evento;
    this.id_jogador = id_jogador;
    this.ganha_casa = ganha_casa;
    this.ganha_fora = ganha_fora;
    this.empate = empate;
    this.estado = "Não paga";
}

```

Figura 24: Construtor que resta na classe, para além do construtor por cópia.

Fica apenas a sobrar, então, o construtor acima com o *code smell* Long Parameter List. A resolução deste *code smell* passou por definir o construtor por omissão seguinte (substituindo o construtor que sobra, que se torna Dead Code).


```

public Aposta() {
    this.id_aposta = -1;
    this.quantia = -1;
    this.id_evento = -1;
    this.id_jogador = "None";
    this.ganha_casa = false;
    this.ganha_fora = false;
    this.empate = false;
    this.estado = "Não paga";
}

public Aposta(Aposta ap) {
    this.id_aposta = ap.getId_aposta();
    this.quantia = ap.getQuantia();
    this.id_evento = ap.getId_evento();
    this.id_jogador = ap.getId_jogador();
    this.ganha_casa = ap.getGanha_casa();
    this.ganha_fora = ap.getGanha_fora();
    this.empate = ap.getEmpate();
    this.estado = ap.getEstado();
}

```

Figura 25: Construtores finais da classe: por omissão e cópia.

Este construtor surge como *refactoring* doutras classes: na classe Database, por exemplo, existiam métodos que invocavam o construtor de Apostas, que também, naturalmente, tinham um Long Parameter List. Pensou-se em dividir a classe Apostas, diminuindo o número de atributos por classe e resolvendo este *smell* na Apostas. No entanto, o problema ia-se manter na classe Database. Assim, na classe Database, por exemplo, em vez de passar todos os argumentos ao construtor, criava-se um objeto, adicionava-se informação ao mesmo (através da invocação de sets) e passava-se o objeto como um todo. Esta técnica de *refactoring* corresponde ao "Preserve Whole Object" e encontra-se um exemplo concreto na próxima secção 3.1.2.2.

3.1.2.2 Classe Database - geral

A classe Database possuía imensos Long Parameter Lists, sendo que apenas vai ser demonstrado um exemplo que descreve todos os outros. Esta classe usava construtores das classes Aposta e EventoDesportivo, que possuíam também Long Parameter List. Consequentemente, como uma bola de neve, surgiram métodos com este tipo de *code smell* quando era invocado o construtor de qualquer uma das classes.

Demonstra-se um exemplo, e a sua resolução, de um método com este *smell* devido ao construtor da classe Aposta.

```
Aposta a = new Aposta(this.cont_apostas++, quantia, id_evento, id_jogador, ganha_casa, ganha_fora, empate);
```

Figura 26: Exemplo de long list parameter da classe Database.

A seguinte resolução usou a técnica de refactoring "Preserve Whole Object".

```
Aposta a = new Aposta();  
a.setQuantia(Double.parseDouble(quantia_field.getText()));  
a.setId_evento(id_evento);  
a.setId_jogador(user);  
a.setGanha_casa(casa_selected);  
a.setGanha_fora(foras_selected);  
a.setEmpate(empate_selected);  
this.betess.registaAposta(a);
```

Figura 27: Resolução de um long list parameter da classe Database, com o construtor por omissão construído na classe Aposta.

3.1.2.3 Classe BetESS - geral

Tal como na classe Database (3.1.2.2), são vários os Long Parameter List encontrados ao longo da classe BetESS. Estes devem-se à utilização de construtores de outras classes, como a Aposta e EventoDesportivo, que possuem construtores já com este tipo de problemas. Então, como são uma consequência dessas classes, definiu-se um construtor por omissão nas mesmas e resolveu-se o *code smell* utilizando a técnica de *refactoring* "Preserve Whole Object". Desta forma, mostra-se um exemplo deste tipo de *code smell*, assim como a sua resolução.

```
public void registaAposta(double quantia, int id_evento, String id_jogador, boolean ganha_casa, boolean ganha_fora, boolean empate){  
    this.database.registaAposta(quantia, id_evento, id_jogador, ganha_casa, ganha_fora, empate);  
}
```

Figura 28: Exemplo de long list parameter da classe BetESS.

A seguinte resolução usou a técnica de *refactoring* "preserve whole object".

```
public void registaAposta(Aposta a){  
    this.database.registaAposta(a);  
}
```

Figura 29: Resolução de um long list parameter da classe BetESS, com o construtor por omissão construído na classe Aposta.

3.1.2.4 Classe EventoDesportivo - Método clone e Construtores

Esta classe seguirá a linha de raciocínio da classe Aposto (secção 3.1.2.1), pelo que não será explicada tão a fundo. Assim, nesta classe foram encontrados três métodos que possuíam um Long Parameter .ist: dois construtores e o método *clone()*:

```
public EventoDesportivo clone() {
    return new EventoDesportivo(this.getId_evento(), this.getequipa_casa(),
        this.getequipa_fora(), this.getEstado(), this.getGanha_casa(),
        this.getGanha_fora(), this.getEmpate(), this.getOdd_casa(),
        this.getOdd_fora(), this.getOdd_empate());
}
```

Figura 30: Método clone().

```
public EventoDesportivo(int id_evento, String equipa_casa,
    String equipa_fora, double odd_casa,
    double odd_fora, double empate) {
    this.id_evento = id_evento;
    this.equipa_casa = equipa_casa;
    this.equipa_fora = equipa_fora;
    this.estado = "Aberto";
    this.ganha_casa = false;
    this.ganha_fora = false;
    this.empate = false;
    this.odd_casa = odd_casa;
    this.odd_fora = odd_fora;
    this.odd_empate = empate;
}

public EventoDesportivo(int id_evento, String equipa_casa,
    String equipa_fora, String estado,
    boolean ganha_casa, boolean ganha_fora,
    boolean empate, double odd_casa, double odd_fora,
    double odd_empate) {
    this.id_evento = id_evento;
    this.equipa_casa = equipa_casa;
    this.equipa_fora = equipa_fora;
    this.estado = estado;
    this.ganha_casa = ganha_casa;
    this.ganha_fora = ganha_fora;
    this.empate = empate;
    this.odd_casa = odd_casa;
    this.odd_fora = odd_fora;
    this.odd_empate = odd_empate;
}
```

Figura 31: Construtores da classe EventoDesportivo.

Para a resolução do **problema no clone()**, usou-se a técnica de *refactoring* "Introduce Parameter Object", sendo necessária a construção de um construtor por cópia para a realização do mesmo. Então, a resolução do método *clone()* procedeu-se da seguinte forma:

```
@Override
public EventoDesportivo clone() {
    return new EventoDesportivo(this);
}
```

Figura 32: Novo clone().

```

public EventoDesportivo(EventoDesportivo ev) {
    this.id_evento = ev.getId_evento();
    this.equipa_casa = ev.getequipa_casa();
    this.equipa_fora = ev.getequipa_fora();
    this.estado = ev.getEstado();
    this.ganha_casa = ev.getGanha_casa();
    this.ganha_fora = ev.getGanha_fora();
    this.empate = ev.getEmpate();
    this.odd_casa = ev.getOdd_casa();
    this.odd_fora = ev.getOdd_fora();
    this.odd_empate = ev.getOdd_empate();
}

```

Figura 33: Construtor que passa a servir o clone().

Isto fez com que um dos Long Parameter Lists dos construtores desta classe se resolvesse sozinho, uma vez que o construtor demonstrado abaixo só existia para servir o *clone()*. Tornando-se este construtor um dead code, o grupo decidiu removê-lo.

```

public EventoDesportivo(int id_evento, String equipa_casa, String equipa_fora,
                        double odd_casa, double odd_fora, double empate) {
    this.id_evento = id_evento;
    this.equipa_casa = equipa_casa;
    this.equipa_fora = equipa_fora;
    this.estado = "Aberto";
    this.ganha_casa = false;
    this.ganha_fora = false;
    this.empate = false;
    this.odd_casa = odd_casa;
    this.odd_fora = odd_fora;
    this.odd_empate = empate;
}

/* construtor utilizado no clone */
public EventoDesportivo(int id_evento, String equipa_casa, String equipa_fora,
                        String estado, boolean ganha_casa, boolean ganha_fora,
                        boolean empate, double odd_casa, double odd_fora,
                        double odd_empate) {
    this.id_evento = id_evento;
    this.equipa_casa = equipa_casa;
    this.equipa_fora = equipa_fora;
    this.estado = estado;
    this.ganha_casa = ganha_casa;
    this.ganha_fora = ganha_fora;
    this.empate = empate;
    this.odd_casa = odd_casa;
    this.odd_fora = odd_fora;
    this.odd_empate = odd_empate;
}

```

Figura 34: Construtor que servia o clone() antigo (2º construtor).

```

public EventoDesportivo(int id_evento, String equipa_casa, String equipa_fora,
                        double odd_casa, double odd_fora, double empate) {
    this.id_evento = id_evento;
    this.equipa_casa = equipa_casa;
    this.equipa_fora = equipa_fora;
    this.estado = "Aberto";
    this.ganha_casa = false;
    this.ganha_fora = false;
    this.empate = false;
    this.odd_casa = odd_casa;
    this.odd_fora = odd_fora;
    this.odd_empate = empate;
}

```

Figura 35: Construtor que resta na classe, para além do construtor por cópia.

Fica apenas a sobrar, então, o construtor acima com o *code smell* Long Parameter List. A resolução deste *smell* passou por definir o construtor por omissão seguinte (substituindo o construtor que sobra, que se torna Dead Code).

```
public EventoDesportivo() {
    this.id_evento = -1;
    this.equipa_casa = "None";
    this.equipa_fora = "None";
    this.estado = "Aberto";
    this.ganha_casa = false;
    this.ganha_fora = false;
    this.empate = false;
    this.odd_casa = -1;
    this.odd_fora = -1;
    this.odd_empate = -1;
}

public EventoDesportivo(EventoDesportivo ev) {
    this.id_evento = ev.getId_evento();
    this.equipa_casa = ev.getequipa_casa();
    this.equipa_fora = ev.getequipa_fora();
    this.estado = ev.getEstado();
    this.ganha_casa = ev.getGanha_casa();
    this.ganha_fora = ev.getGanha_fora();
    this.empate = ev.getEmpate();
    this.odd_casa = ev.getOdd_casa();
    this.odd_fora = ev.getOdd_fora();
    this.odd_empate = ev.getOdd_empate();
}
```

Figura 36: Construtores finais da classe: por omissão e cópia.

Este construtor surge como *refactoring* das classes BetESS e Database, como já foi explicado nas secções anteriores. Estas classes possuíam invocações ao construtor desta classe, o que tornava estes métodos Long Parameter List também. Assim, resolveu-se o problemas nessas classes através da técnica de *refactoring* "Preserve Whole Object".

3.1.2.5 Classe Jogador - Método clone e Construtor

Na classe Jogador o erro manteve-se, sendo que se encontraram Long Parameter Lists num construtor e no método *clone()*, como se demonstra de seguida:

```
public Jogador clone(){
    return new Jogador(this.getEmail(), this.getNome(), this.getPassword(),
                        this.getContacto(), this.getSaldo(),
                        this.getNotificacoes());
}
```

Figura 37: Método clone().

```
public Jogador(String email, String nome, String password, String contacto,
               double saldo, List<Notificacao> notificacoes) {
    this.email = email;
    this.nome = nome;
    this.password = password;
    this.contacto = contacto;
    this.saldo = saldo;
    this.notificacoes = notificacoes;
}
```

Figura 38: Construtor da classe Jogador.

Para resolver este *smell no clone()*, o grupo usou a técnica de *refactoring* "Introduce Parameter Object", sendo necessária a construção de um construtor por cópia para a realização do mesmo. Então, a resolução do método *clone()* procedeu-se da seguinte forma:

```
@Override
public Jogador clone(){
    return new Jogador(this);
}
```

Figura 39: Novo clone().

```
public Jogador(Jogador jog){
    this.email = jog.getEmail();
    this.nome = jog.getNome();
    this.password = jog.getPassword();
    this.contacto = jog.getContacto();
    this.saldo = jog.getSaldo();
    this.notificacoes = jog.getNotificacoes();
}
```

Figura 40: Construtor que passa a servir o clone().

Esta modificação fez com que o construtor anterior, que era um Long Parameter List, se tornasse Dead Code, resolvendo-se com a simples eliminação do método.

```

/* CONSTRUTOR */
public Jogador(String email, String nome, String password, String contacto) {
    this.email = email;
    this.nome = nome;
    this.password = password;
    this.contacto = contacto;
    this.saldo = 5; /* CONSIDERAR A OFERTA DE SALDO PARA UM NOVO UTILIZADOR */
    this.notificacoes = new ArrayList ();
}

public Jogador(String email, String nome, String password,
                String contacto, double saldo,
                List<Notificacao> notificacoes) {
    this.email = email;
    this.nome = nome;
    this.password = password;
    this.contacto = contacto;
    this.saldo = saldo;
    this.notificacoes = notificacoes;
}

```

Figura 41: Construtores originais (não aparecendo o construtor de cópia criado acima).

```

/* CONSTRUTOR */
public Jogador(String email, String nome, String password, String contacto) {
    this.email = email;
    this.nome = nome;
    this.password = password;
    this.contacto = contacto;
    this.saldo = 5; /* CONSIDERAR A OFERTA DE SALDO PARA UM NOVO UTILIZADOR */
    this.notificacoes = new ArrayList ();
}

```

Figura 42: Construtor que restou na classe (não aparecendo o construtor de cópia criado acima).

3.2 Object-Orientation Abusers

Nesta secção apresentam-se os OOAbusers encontrados ao longo das classes, assim como a sua resolução e vantagens.

3.2.1 Switch Statements

Um Switch Statement ocorre quando se tem um operador "switch" complexo, ou uma sequência de ifs. Corrigir este tipo de *code smells* aumenta a organização e legibilidade do código.

3.2.2 Classe BetESS - Método *fechaEvento()*

Detetou-se uma grande quantidade de ifs no método problemático *fechaEvento()*. No entanto, a sua resolução já se encontra explicada na secção 3.1.1.2 e a técnica de *refactoring* utilizada é intitulada "Extract Method".

```
if (evento_ganha_casa == aposta_ganha_casa
    && evento_ganha_fora == aposta_ganha_fora
    && evento_empate == aposta_empate)
{
    if (evento_ganha_casa){
        odd = e.getOdd_casa();
        saldo += odd * quant_aposta;
    }
    else if (evento_ganha_fora){
        odd = e.getOdd_fora();
        saldo += odd * quant_aposta;
    }
    else if (evento_empate){
        odd = e.getOdd_empate();
        saldo += odd * quant_aposta;
    }
}
```

Figura 43: Demonstração do encadeamento de ifs no código original.

```
if (verificaAposta(e, a)) {
    if (e.getGanha_casa()){
        saldo = atualizaSaldoAposta(e, a, saldo);
    }
    else if (e.getGanha_fora()){
        saldo = atualizaSaldoAposta(e, a, saldo);
    }
    else if (e.getEmpate()){
        saldo = atualizaSaldoAposta(e, a, saldo);
    }
}
```

Figura 44: Demonstração do encadeamento de ifs no código atualizado.

3.3 Dispensables

Nesta secção apresentam-se os dispensables encontrados ao longo das classes, assim como a sua resolução e vantagens.

3.3.1 Data Class

Uma data class refere-se a uma classe que contém apenas alguns campos públicos e métodos para os acessar (getters e setters). Estas classes não contêm qualquer funcionalidade adicional e não podem operar independentemente nos dados da mesma. Corrigir este tipo de *code smells* traz como vantagens a mais fácil deteção de código duplicado, e melhora a organização e leitura de uma classe. Isto verifica-se porque, uma vez resolvido o problema, as operações de certas informações particulares passam a estar reunidas num único sítio, em vez de espalhadas pelo código.

3.3.1.1 Classe Aposta

Tal fenómeno foi detetado na classe Aposta, como se mostra de seguida:

```
public class Aposta implements Serializable{  
  
    public int id_aposta;  
    public double quantia;  
    public int id_evento;  
    public String id_jogador;  
    public boolean ganha_casa;  
    public boolean ganha_fora;  
    public boolean empate;  
    public String estado;  
}
```

Figura 45: Campos públicos na classe Aposta.

A solução para este problema passou pelo uso da técnica de *refactoring* "Encapsulate Field", já explicada na contextualização do trabalho, e que se demonstra abaixo.

```
public class Aposta implements Serializable{  
  
    private int id_aposta;  
    private double quantia;  
    private int id_evento;  
    private String id_jogador;  
    private boolean ganha_casa;  
    private boolean ganha_fora;  
    private boolean empate;  
    private String estado;  
}
```

Figura 46: Alteração dos campos públicos para privados.

```

/* GETTERS */
public int getId_aposta() {
    return id_aposta;
}

public double getQuantia() {
    return quantia;
}

public int getId_evento() {
    return id_evento;
}

public String getId_jogador() {
    return id_jogador;
}

public boolean getGanha_casa() {
    return ganha_casa;
}

public boolean getGanha_fora() {
    return ganha_fora;
}

public boolean getEmpate() {
    return empate;
}

public String getEstado() {
    (...)
}

```

Figura 47: Especificação dos getters/setters.

Alguns destes getters/setters já existiam, mas o grupo especificou todos os necessários ao acesso das variáveis da Aposto.

3.3.1.2 Classe EventoDesportivo

O problema da secção anterior repetiu-se na classe EventoDesportivo.

```
public class EventoDesportivo implements Serializable{  
  
    public int id_evento;  
    public String equipa_casa;  
    public String equipa_fora;  
    public String estado;  
    public boolean ganha_casa;  
    public boolean ganha_fora;  
    public boolean empate;  
    public double odd_casa;  
    public double odd_fora;  
    public double odd_empate;  
}
```

Figura 48: Campos públicos na classe EventoDesportivo.

A solução para este problema passou também pelo uso da técnica de *refactoring* "Encapsulate Field", já explicada na contextualização do trabalho, e que se demonstra abaixo.

```
public class EventoDesportivo implements Serializable{  
  
    private int id_evento;  
    private String equipa_casa;  
    private String equipa_fora;  
    private String estado;  
    private boolean ganha_casa;  
    private boolean ganha_fora;  
    private boolean empate;  
    private double odd_casa;  
    private double odd_fora;  
    private double odd_empate;  
}
```

Figura 49: Alteração dos campos públicos para privados.

```

/* GETTERS */

public int getId_evento() {
    return id_evento;
}

public String getequipa_casa() {
    return equipa_casa;
}

public String getequipa_fora() {
    return equipa_fora;
}

public String getEstado() {
    return estado;
}

public boolean getGanha_casa() {
    return ganha_casa;
}

public boolean getGanha_fora() {
    return ganha_fora;
}

public boolean getEmpate() {
    (...)
}

```

Figura 50: Especificação dos getters/setters.

Alguns destes getters/setters já existiam, mas o grupo especificou todos os necessários ao acesso das variáveis do EventoDesportivo.

3.3.1.3 Classe Jogador

Mais uma vez, encontraram-se más práticas, relativas ao code smell "data class".

```
public class Jogador implements Serializable{
    public String email;
    public String nome;
    private String password;
    public String contacto;
    public double saldo;
    public List<Notificacao> notificacoes;
```

Figura 51: Campos públicos na classe Jogador.

A solução para este problema passou, mais uma vez, pelo uso da técnica de *refactoring* "Encapsulate Field", já explicada na contextualização do trabalho, e que se demonstra abaixo.

```
public class Jogador implements Serializable{
    private String email;
    private String nome;
    private String password;
    private String contacto;
    private double saldo;
    private List<Notificacao> notificacoes;
```

Figura 52: Alteração dos campos públicos para privados.

```
/* GETTERS */

public String getEmail() {
    return email;
}

public String getNome() {
    return nome;
}

public String getContacto() {
    return contacto;
}

public double getSaldo() {
    return saldo;
}

private String getPassword() {
    return password;
}
```

Figura 53: Especificação dos getters/setters.

Alguns destes getters/setters já existiam, mas o grupo especificou todos os necessários ao acesso das variáveis do Jogador.

3.3.2 Dead Code

Os Dead Codes referem-se a variáveis, parâmetros, campos, métodos ou classes que não estão a ser usadas. Remover este problema fornece vantagens como um tamanho reduzido do código e código mais simples. Assim, mostram-se os Dead Codes que o grupo detetou, que estavam no código original.

3.3.2.1 Classe Aposta

Foram vários os Dead Codes descobertos para a presente classe, que se apresentam de seguida, rodeados a vermelho.

```
/* SETTERS */

public void setId_aposta(int id_aposta) {
    this.id_aposta = id_aposta;
}

public void setQuantia(double quantia) {
    this.quantia = quantia;
}

public void setId_evento(int id_evento) {
    this.id_evento = id_evento;
}

public void setId_jogador(String id_jogador) {
    this.id_jogador = id_jogador;
}

public void setGanha_casa(boolean ganha_casa) {
    this.ganha_casa = ganha_casa;
}

public void setGanha_fora(boolean ganha_fora) {
    this.ganha_fora = ganha_fora;
},
```

Figura 54: Dead Code, rodeado a vermelho.

A resolução deste problema focou-se na simples eliminação do código em excesso.

```
/* SETTERS */

public void setGanha_casa(boolean ganha_casa) {
    this.ganha_casa = ganha_casa;
}

public void setGanha_fora(boolean ganha_fora) {
    this.ganha_fora = ganha_fora;
}

public void setEmpate(boolean empate) {
    this.empate = empate;
}
```

Figura 55: Eliminação do Dead Code.

3.3.2.2 Classe Equipa

Os Dead Codes encontrados na classe Equipa são mostrados abaixo, rodeados a vermelho.

```
/* GETTERS */
public String getId_liga() {
    return id_liga;
}

public String getDesignacao() {
    return designacao;
}

/* SETTERS */
public void setId_liga(String id_liga) {
    this.id_liga = id_liga;
}

public void setDesignacao(String designacao) {
    this.designacao = designacao;
}
```

Figura 56: Dead Code, rodeado a vermelho.

A resolução deste problema centrou-se na simples eliminação do código em excesso.

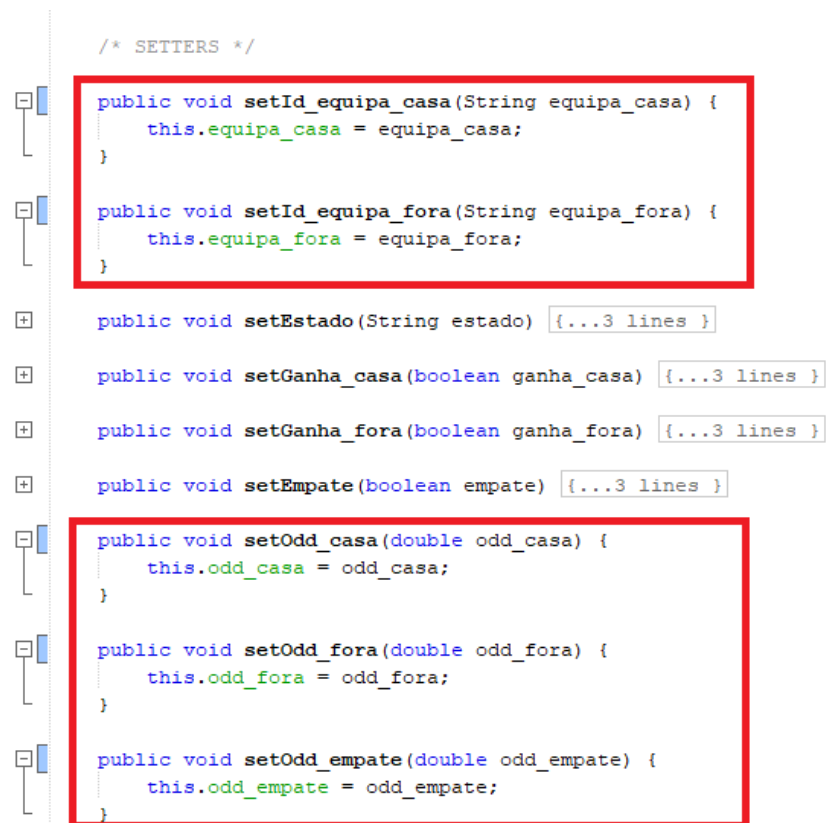
```
/* GETTERS */
public String getId_liga() {
    return id_liga;
}

public String getDesignacao() {
    return designacao;
}
```

Figura 57: Eliminação do Dead Code.

3.3.2.3 Classe EventoDesportivo

Os Dead Codes presentes na classe EventoDesportivo são demonstrados de seguida, rodeados a vermelho.



The image shows a snippet of Java code from the `EventoDesportivo` class. The code is organized into sections by comments. Two sections are highlighted with red rectangular boxes, indicating dead code. The first box encloses two setter methods: `setId_equipa_casa` and `setId_equipa_fora`. The second box encloses three setter methods: `setOdd_casa`, `setOdd_fora`, and `setOdd_empate`. Other visible methods include `setEstado`, `setGanha_casa`, `setGanha_fora`, and `setEmpate`, each followed by a placeholder indicating they consist of 3 lines of code.

```
/* SETTERS */

public void setId_equipa_casa(String equipa_casa) {
    this.equipa_casa = equipa_casa;
}

public void setId_equipa_fora(String equipa_fora) {
    this.equipa_fora = equipa_fora;
}

public void setEstado(String estado) {...3 lines }

public void setGanha_casa(boolean ganha_casa) {...3 lines }

public void setGanha_fora(boolean ganha_fora) {...3 lines }

public void setEmpate(boolean empate) {...3 lines }

public void setOdd_casa(double odd_casa) {
    this.odd_casa = odd_casa;
}

public void setOdd_fora(double odd_fora) {
    this.odd_fora = odd_fora;
}

public void setOdd_empate(double odd_empate) {
    this.odd_empate = odd_empate;
}
```

Figura 58: Dead Code, rodeado a vermelho.

A solução deste problema passou pela simples eliminação do código em excesso.

```
/* SETTERS */

public void setEstado(String estado) {
    this.estado = estado;
}

public void setGanha_casa(boolean ganha_casa) {
    this.ganha_casa = ganha_casa;
}

public void setGanha_fora(boolean ganha_fora) {
    this.ganha_fora = ganha_fora;
}

public void setEmpate(boolean empate) {
    this.empate = empate;
}
```

Figura 59: Eliminação do Dead Code.

3.3.2.4 Classe Liga

O Dead Code originável na classe Liga é demonstrado de seguida, rodeado a vermelho.

```
public String getNome() {  
    return nome;  
}  
  
public void setNome(String nome) {  
    this.nome = nome;  
}  
  
public Liga clone() {  
    return new Liga(this.getNome());  
}
```

Figura 60: Dead Code, rodeado a vermelho.

A solução deste problema passou pela simples eliminação do código em excesso.

```
public String getNome() {  
    return nome;  
}  
  
public Liga clone() {  
    return new Liga(this.getNome());  
}
```

Figura 61: Eliminação do Dead Code.

3.3.2.5 Classe Notificação

O Dead Code originável na classe Notificação é demonstrado de seguida, rodeado a vermelho.

```
/* SETTERS */  
  
public void setId_aposta(int id_aposta) {  
    this.id_aposta = id_aposta;  
}  
  
public void setBalanco(double balanco) {  
    this.balanco = balanco;  
}  
  
public void marcarLida() {  
    this.status = "Lida";  
}
```

Figura 62: Dead Code, rodeado a vermelho.

A resolução deste problema focou-se na simples eliminação do código em excesso.

```
public void marcarLida() {  
    this.status = "Lida";  
}
```

Figura 63: Eliminação do Dead Code.

3.4 Couplers

Nesta secção apresentam-se os Couplers encontrados ao longo das classes, assim como a sua resolução e vantagens.

3.4.1 Feature Envy

Uma Feature Envy é quando um método acessa mais a informação de outro objeto do que a sua própria informação. A correção deste tipo de "erros" traz vantagens como redução de código duplicado (se a código de gestão da informação estiver num sítio central) e melhor organização do código (métodos para gestão de dados devem ser próximos aos dados em si).

3.4.1.1 Classe BetESS - Método *fechaEvento()*

Antes de ser feita qualquer alteração no método, note-se que a maioria das alterações sobre os dados são feitas ao nível do objeto *database*, o que evidencia uma má localização da declaração do método. Desta forma, e para resolução deste *smell* o método foi transferido para a classe *Database*, uma vez que as operações efetuadas são maioritariamente sobre dados presentes nesta. Esta técnica de *refactoring* é denominada "Move Method".

```
EventoDesportivo e = this.database.getEventoDesportivo(id_Evento);
e.setGanha_casa(ganha_casa);
e.setGanha_fora(ganha_fora);
e.setEmpate(empate);

for (Aposta a : this.database.getApostasEvento(e.getId_evento())){
```

Figura 64: Observação dos acessos sucessivos do método a objetos fora da BetESS.

3.4.2 Message Chains

Message Chains consiste no sucessivo encadeamento de chamadas a métodos. Corrigir este *smell* traz vantagens como redução de dependências entre as classes do encadeamento e redução de código em massa.

3.4.2.1 Classe BetESS - Método *fechaEvento()*

Inicialmente, foram detetadas Message Chains no método *fechaEvento()* em que se verificava o encadeamento de mensagens entre objetos desnecessariamente. A resolução deste *code smell* ocorreu juntamente com a modificação descrita na secção 3.4.1.1 pelo que não há nada de relevante a discutir. Também, o facto de termos movido o método na secção anterior reduziu bastante as Message Chains, uma vez que se aproximou o método dos dados.

```
public void fechaEvento(int id_Evento, boolean ganha_casa, boolean ganha_fora, boolean empate){  
    EventoDesportivo e = this.database.getEventoDesportivo(id_Evento);  
    e.setGanha_casa(ganha_casa);  
    e.setGanha_fora(ganha_fora);  
    e.setEmpate(empate);  
  
    for (Aposta a : this.database.getApostasEvento(e.getId_evento())){  
        if ( e != null && e.getEstado().equals("Aberto")){  
            boolean evento_ganha_casa = e.getGanha_casa();  
            boolean evento_ganha_fora = e.getGanha_fora();  
            boolean evento_empate = e.getEmpate();  
  
            boolean aposta_ganha_casa = a.getGanha_casa();  
            boolean aposta_ganha_fora = a.getGanha_fora();  
            boolean aposta_empate = a.getEmpate();  
  
            Jogador j = this.database.checkUser(a.getId_jogador());  
            double saldo = j.getSaldo();  
            double quant_aposta = a.getQuantia();  
            double odd = -10000;  
            double saldo_ant = saldo;
```

Figura 65: Observação do encadeamento de métodos.

4 Métricas de avaliação do Refactoring

4.1 Testes

Uma vez que esta aplicação é baseada na interação com o utilizador, os testes para a verificação de que o comportamento da aplicação não foi alterado pelo processo de *refactoring* consistirão na execução de uma ação e posterior observação do comportamento da mesma. Este comportamento passa por ver se os dados são guardados corretamente como no programa original, e se as alterações são efetuadas corretamente a todos os objetos que deveriam.

De seguida, será apresentada a sequência de ações executada na aplicação como verificação de que os resultados se encontram corretos. Utilizaram-se criações de objetos, e verificaram-se que estavam a ser guardadas, e recorreu-se ao processo de fechar um evento, já que é um método que alberga quase a totalidade das funcionalidades do programa.

A interface mostra a 'Área de administração' com um menu lateral contendo: Ver Jogadores, Eventos Desportivos (selecionado), Jogadores Bloqueados, Apostas, Nova Equipa e Nova Liga. O formulário principal, 'Novo Evento Desportivo', contém campos para: Equipa Casa (FC Porto), Equipa Fora (SC Braga), Odd Casa (1.45), Odd Fora (2) e Odd Empate (1.6). Um botão 'Registrar Evento' está no canto inferior direito, e 'Terminar Sessão' no canto inferior esquerdo.

Figura 66: Criação de um evento desportivo.

A interface mostra a 'Área de administração' com o menu lateral. O formulário principal, 'Eventos Desportivos', apresenta uma tabela com as seguintes colunas: Identificador, Equipa Casa, Equipa Fora, Ganha Casa, Ganha Fora, Empate e Estado. A primeira linha de dados mostra: 1, FC Porto, SC Braga, com os botões de vitória desativados e o botão de empate ativo. Abaixo da tabela, há botões 'Registrar Evento Desportivo' e 'Fechar Evento', além de 'Terminar Sessão' no canto inferior esquerdo.

Figura 67: Verificação da criação do evento desportivo.

A interface mostra a 'Área autenticada' com o menu lateral contendo: Apostar (selecionado), Minhas apostas, Créditos, Notificações e Editar Perfil. O formulário principal, 'Bem-vindo à BetESS', apresenta uma tabela com as seguintes colunas: Evento número, Casa, Fora, Odd Casa, Odd Fora e Odd Empate. A primeira linha de dados mostra: 1, FC Porto, SC Braga, com as odds 1.45, 2 e 1.6. Abaixo da tabela, há campos para 'Quantia' (2) e '€ (euros)', e 'Ganhos possíveis: 2.90'. Há também botões de seleção para Casa (selecionado), Fora e Empate, e um botão 'Apostar'. O botão 'Terminar sessão' está no canto inferior esquerdo.

Figura 68: Criação de uma aposta.

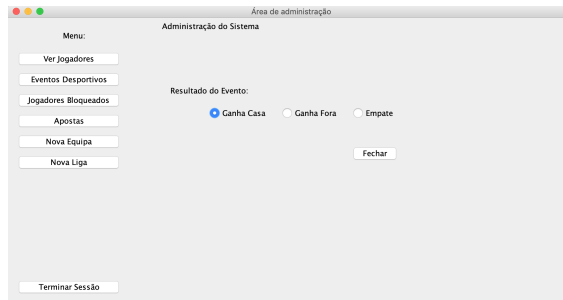


Figura 69: Registo do resultado de fecho do evento.



Figura 70: Visualização da notificação da aposta após o fecho do evento.

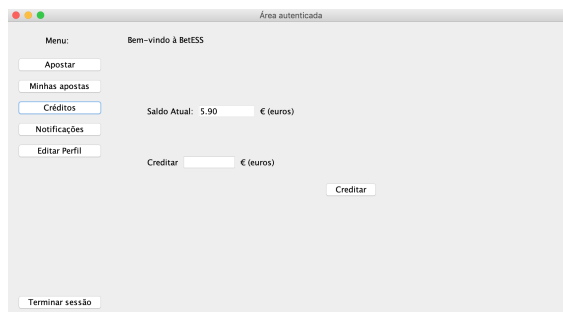


Figura 71: Verificação da atualização do saldo do apostador.

Assim, observa-se que o processo de fecho de um evento continua a produzir os resultados esperados após o processo de *refactoring*.

Adicionalmente, de forma a provar que a execução de *refactoring* sobre *code smells* como Dead Code, foram feitos testes mais conceituais, isto é, na IDE onde o grupo desenvolveu o programa, ao remover um pedaço de código (método Dead Code), não deve dar erro de sintaxe em nenhuma outra classe. Todos estes testes passaram com distinção.

4.2 Tempos de execução

Uma vez que, o principal objetivo do processo de *refactoring* é a melhoria da qualidade do código e não a melhoria do desempenho, foi apenas testado se as funcionalidades do programa se mantiveram inalteradas. Devido à natureza deste processo, existe a possibilidade de uma ligeira perda de desempenho devido às sucessivas chamadas a métodos distintos. Por outro lado, a evolução dos compiladores e a simplicidade das tarefas a executar neste sistema, faz com que esta baixa no desempenho não seja de todo significativa.

5 Conclusões e Sugestões

A resolução deste segundo trabalho prático foi bastante importante e enriquecedora, pois permitiu aos membros do grupo perceber e interiorizar melhor os conceitos abordados nas aulas da Unidade Curricular de Arquiteturas de Software.

Deste modo, foram aprofundados conceitos relativos a *code smells* e as consequentes técnicas de *refactoring*. O grupo sente que evoluiu, apesar de alguma dificuldade inicial, pois era requerido um estudo prévio de cada tipo de *code smell*, e era necessário que o comportamento externo do programa não se alterasse. A partir do momento que se entendeu o poder e o alcance do *refactoring* na legibilidade, simplicidade e melhoria de desempenho da aplicação, o grupo iniciou este processo iterativo, facilmente executando o que era pedido.

Em suma, é feita uma apreciação positiva relativamente ao trabalho realizado, visto que a realização de todas as etapas propostas foram conseguidas com sucesso. O grupo conseguiu tirar partido dos conhecimentos adquiridos neste projeto, sentido-se capaz de, num contexto futuro, aplicar os conceitos subjacentes de forma eficaz. É evidente que, num outro contexto (como um projeto de grandes dimensões), seria benéfico que fossem utilizadas ferramentas de análise de código fonte ao invés da procura manual dos *code smells*. Ainda assim, as capacidades que o grupo adquiriu permitem que, no futuro, se corrijam e previnam erros antes sequer de eles existirem!