

UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

ARQUITETURAS DE SOFTWARE
ENGENHARIA DE SISTEMAS DE SOFTWARE

Refactoring BetESS

Autores:

Daniel Maia
Vitor Peixoto

Número:

A77531
A79175

4 de Janeiro de 2019

Conteúdo

1	Introdução	2
2	<i>Code smells</i>	3
2.1	<i>Bloaters</i>	3
2.1.1	Classes grandes	3
2.1.2	Métodos longos	3
2.1.3	<i>Data clumps</i>	7
2.2	<i>Change preventers</i>	8
2.2.1	<i>Divergent change</i>	8
2.3	<i>Dispensables</i>	9
2.3.1	Comentários	9
2.3.2	Código duplicado	10
2.3.3	Código “morto”	11
2.4	<i>Couplers</i>	11
2.4.1	<i>Feature envy</i>	11
2.4.2	<i>Message chains</i>	13
3	Métricas	14
4	Conclusão	16

1 Introdução

Ao longo da implementação e subsequente expansão de um sistema, podem ocorrer instâncias de escrita de código de qualidade dúbia. Estas podem emergir por uma variedade de razões, tais como falta de tempo, conhecimento ou simplesmente atenção. Para combater quaisquer aspectos negativos das qualidades não funcionais do código, frequentemente recorre-se à técnica de *refactoring* do código.

O *refactoring* define-se como o conjunto de técnicas cujo intuito é de limpar o código e melhorar o *design* de um sistema de *software* enquanto mantendo o seu comportamento exterior. Deste modo, espera-se obter código mais legível, extensível e menos complexo.

O objetivo deste trabalho prático é de aplicar as técnicas de *refactoring* dispostas ao projeto **BetESS** implementado anteriormente sem padrões. Serão descritos os diferentes tipos de código que pode ser melhorado, denominados de *code smells*, bem como as técnicas utilizadas para os corrigir. Serão também identificadas métricas de código através das quais o impacto das alterações efetuadas na funcionalidade do sistema. No final, será feito um balanço do trabalho efetuado, dos resultados obtidos e do possível trabalho futuro com as técnicas de *refactoring*.

2 *Code smells*

2.1 *Bloaters*

Bloaters são pedaços de código que cresceram para proporções tão grandes que se tornam difíceis de trabalhar. Estes problemas tendem a crescer com o desenvolvimento do código. Estes podem ser métodos excessivamente longos, classes com grandes proporções ou listas de parâmetros longas num determinado método, entre outras. Analisando o código do sistema, foram encontradas diversas aparições deste *code smell*:

2.1.1 Classes grandes

Analizando o código pela primeira vez há uma classe que se sobressai no que toca ao número de métodos e variáveis. Trata-se da classe `Home`, classe onde o apostador efetua as apostas nos eventos. No entanto é compreensível uma vez que é uma classe de um formulário gráfico do sistema e apresenta nove diferentes botões para efetuar uma aposta, ou seja, um para cada jogo e ainda vinte e sete métodos para efeitos gráficos ao efetuar ações no formulário.

Tudo o que foi referido é por si só um problema, tornando esta classe grande e confusa. Para piorar, ainda é aqui que são verificadas todas as condições necessárias para uma aposta ser considerada válida, sendo só depois registada a aposta na classe `BetESS`, classe responsável pelo registo dos dados do sistema. Mas iremos tratar desses problemas mais à frente.

Relativamente ainda ao problema do tamanho da classe `Home`, a única maneira possível de reduzir o número de métodos que esta contém seria através de uma remodelação da interface gráfica o que envolveria um enorme esforço para esta etapa. Tratou-se então de reduzir o tamanho da classe de outra forma: reduzindo o número de linhas de cada método.

2.1.2 Métodos longos

A classe `Home` anteriormente referida, bem como a maioria das classes de apresentação, são ricas em métodos longos, fruto de um método de produção de código rápido, mas descuidado e com evidentes lacunas em reutilização e comprehensibilidade.

Alguns desses métodos exageradamente grandes são os métodos para **efetuar uma aposta**. São ao todo nove métodos (um para cada botão no formulário) com o seguinte aspeto (página seguinte):

```

private void j8BetActionPerformed(java.awt.event.ActionEvent evt) {
    int res = 0;
    int val = (Integer) j8Spin.getValue();
    if(j8V.isSelected()) res = 1;
    else if (j8E.isSelected()) res = 2;
    else if (j8D.isSelected()) res = 3;
    else {
        ImageIcon icon = new ImageIcon(getClass().getClassLoader().getResource("resources/icons/warning.png"));
        JOptionPane.showMessageDialog(null, "Selecione um resultado!", "Aviso", JOptionPane.INFORMATION_MESSAGE, icon);
    }
    if(res!=0){
        boolean apostou = false;
        for (Aposta a : this.betess.getApostadores().get(apostador.getID()).getApostas())
            if(a.getEvento().getID() == jogos.get(7).getID())
                apostou = true;
        if(apostou) {
            ImageIcon icon = new ImageIcon(getClass().getClassLoader().getResource("resources/icons/forbidden.png"));
            JOptionPane.showMessageDialog(null, "Já registou uma aposta neste evento.", "Aviso", JOptionPane.INFORMATION_MESSAGE, icon);
        }
        else if(this.apostador.getESSCoins()-val >= 0){
            Aposta a = new Aposta(res, val, jogos.get(7), true);
            this.betess.getApostadores().get(apostador.getID()).efetuarAposta(a); efetua a aposta
            na classe Apostador
            ImageIcon icon = new ImageIcon(getClass().getClassLoader().getResource("resources/icons/check.png"));
            JOptionPane.showMessageDialog(null, "Aposta registrada com sucesso!", "Sucesso", JOptionPane.INFORMATION_MESSAGE, icon);
            try {
                betess.save(this.betess);
            } catch (IOException ex) {
                Logger.getLogger(Home.class.getName()).log(Level.SEVERE, null, ex);
            }
            Home home = new Home(this.betess, apostador);
            home.setVisible(true);
            this.setVisible(false);
        }
        else{
            ImageIcon icon = new ImageIcon(getClass().getClassLoader().getResource("resources/icons/forbidden.png"));
            JOptionPane.showMessageDialog(null, "Não tem saldo suficiente para realizar a aposta.", "Aviso", JOptionPane.INFORMATION_MESSAGE, icon);
        }
    }
}

```

condicionante que verifica se o apostador selecionou um palpite

condicionante que verifica se o apostador já tem apostas registradas neste evento

efetua a aposta na classe Apostador

salvar progresso e atualizar form

condicionante que verifica se o saldo do cliente lhe permite efetuar a aposta

Figura 1: Um dos nove métodos para efetuar uma aposta.

Evidentemente que um método com cerca de 40 linhas é de uma compreensão muito difícil, onde a introdução de novas condições ou funcionalidades se revela dispendiosa em termos de tempo e dinheiro investidos. Ainda para mais se esse código praticamente se repete noutras nove métodos (problema a ser resolvido mais à frente).

Assim sendo, optou-se por extrair as partes em comum aos nove métodos (a amarelo no código), ficando apenas o absolutamente necessário, ou seja, a recolha de informação do *form*.

Para além disso, temos uma porção de código responsável por gravar o progresso (utilizando persistência) e atualizar o *form*. Essa porção é também comum a todos os métodos que envolvem modificações na estrutura de dados do sistema, por isso foi extraída para um método chamado `saveNrefresh()`.

O restante processo para efetuar uma aposta encontra-se no método `efetuarAposta(Aposta a)` localizada na classe *Apostador*.

```

public class Home extends javax.swing.JFrame {
    private void jbBetActionPerformed(java.awt.event.ActionEvent evt) {
        int res = 0;
        if(j8V.isSelected()) res=1;
        else if (j8E.isSelected()) res=2;
        else if (j8D.isSelected()) res=3;
        Aposta ap = new Aposta(res, (Integer) j8Spin.getValue(), betess.getEventoAtivo(7), apostador, true);
        if(apostador.efetuarAposta(ap)) this.saveRefresh();
    }
}

public class Apostador implements Serializable{
    public boolean efetuarAposta(Aposta a){
        if(podeApostar(a)){
            apostas.add(a);
            levantarCoins(a.getValor());
            BetESS ex = new BetESS();
            ex.popupWindow(1, "Aposta registada com sucesso!", "Sucesso");
            return true;
        }
        return false;
    }

    public boolean podeApostar(Aposta apostas){
        BetESS ex = new BetESS();
        boolean saldoInsuf = escoins-aposta.getValor() < 0;
        boolean noSelc = apostas.getPalpite()==0;
        boolean jaApostou = false;
        for (Aposta ap : apostas)
            if(ap.getEventoID() == apostas.getEventoID())
                jaApostou = true;

        if(jaApostou){
            ex.popupWindow(3, "Já registou uma aposta neste evento.", "Aviso");
            return false;
        }

        else if(saldoInsuf){
            ex.popupWindow(3, "Não tem saldo suficiente para realizar a aposta.", "Aviso");
            return false;
        }

        else if(noSelc){
            ex.popupWindow(3, "Não selecionou nenhum palpite.", "Aviso");
            return false;
        }
        return true;
    }
}

```

Figura 2: *Refactoring* do processo de efetuar uma aposta.

Dada a elevada dimensão e complexidade ciclomática do método `efetuarAposta(Aposta a)`, extraímos deste método a verificação das condições para efetuar a aposta, para o método `podeApostar(Aposta a)`. Deste modo, facilitamos a introdução de novas condicionantes ao registo de uma aposta.

Outros métodos também apresentam complexidade tão grande ou superior aos métodos de efetuar apostas, como é o caso do método que **declara o fim de um evento** e executa todas as ações consequentes, como a distribuição dos prémios e das notificações aos apostadores.

```

private void fecharButtonActionPerformed(java.awt.event.ActionEvent evt) {
    String[] equipas = split(this.eventCombo.getSelectedItem().toString(), " X ");
    ArrayList<Evento> evAtiv = new ArrayList<>();
    for(Evento e : this.betess.getEventos().values()) {
        if(e.getEstado())
            evAtiv.add(e);
    }
    for(Evento e : evAtiv){
        if(e.getEquipaC().getNome().equals(equipas[0]) && e.getEquipaF().getNome().equals(equipas[1])){
            this.betess.FinalizarEvento(e, resField.getText());
            String[] venc = split(resField.getText(),"-");
            int res;
            if(Integer.parseInt(venc[0])>Integer.parseInt(venc[1])) res = 1;
            else if(Integer.parseInt(venc[1])>Integer.parseInt(venc[0])) res = 3;
            else res = 2;
            for(Apostador a : this.betess.getApostadores().values()){
                //ArrayList<Aposta> toRen = new ArrayList<>();
                for(Aposta ap : a.getApostas()){
                    if(ap.getEvento().equals(e)){
                        if(ap.getResultado()==res){
                            if(res==1) a.adicionarESSCoins(e.getOddV()*ap.getValor());
                            else if(res==2) a.adicionarESSCoins(e.getOddE()*ap.getValor());
                            else if(res==3) a.adicionarESSCoins(e.getOddD()*ap.getValor());
                            ImageIcon icon = new ImageIcon(getClass().getClassLoader().getResource("resources/icons/check.png"));
                            JOptionPane.showMessageDialog(null, "Evento encerrado e prêmios distribuídos.", "Sucesso", JOptionPane.INFORMATION_MESSAGE, icon);
                        }
                        ap.notificaApostador();
                    }
                }
            }
        }
    }
    try {
        betess.save(betess);
    } catch (IOException ex) {
        Logger.getLogger(Admin.class.getName()).log(Level.SEVERE, null, ex);
    }
    Admin admin = new Admin(this.betess);
    admin.setVisible(true);
    this.setVisible(false);
}

```

Annotations on the code:

- Obter lista dos eventos ativos apenas**: A box around the loop that filters active events from the BetESS event map.
- Obter vencedor do evento dado o resultado.**: A box around the logic that determines the winner based on the result string ("1-2", "2-1", or "0-0"). It includes a note: "ex: 2-1 retorna 1 porque venceu a equipa da casa. 2-2 retorna 2 porque foi empate. 0-3 retorna 3 porque venceu a equipa de fora."
- Obter todos os apostadores neste evento e distribuir os prêmios respectivos, bem como adicionar notificação.**: A box around the nested loop that iterates through all bettors for the event and distributes their respective winnings.
- Salvar progresso e atualizar form**: A box around the save operation and the visibility changes at the end of the method.

Figura 3: Método responsável por encerrar um evento e ações consequentes.

Este método foi também ele decomposto em vários métodos mais simples. Na classe BetESS criámos os métodos `GetEventosAtivos()` e `finalizarEvento(String jogo, String resultado)`. O primeiro é responsável por retornar apenas os eventos declarados como ativos, resolvendo a primeira caixa amarela do método original. O segundo é o método que trata do processo todo para encerrar um evento, invocando as funções auxiliares necessárias.

A segunda caixa amarela foi transformada no método `getVencedor()`. Assim pode ser utilizado por outros métodos e eliminar outro *code smell*, código duplicado (que se vai verificar mais à frente). A terceira foi extraída para o método `distribuiPremios(List<Apostador> apostadores)` e utiliza o método `ganhos()` como auxiliar para calcular o lucro obtido na aposta. O método `ganhos()` reduz também o *smell* de código duplicado, como vamos verificar mais à frente.

O resultado final após o *refactoring* deste processo pode ser visto abaixo (página seguinte):

```

public class Admin extends javax.swing.JFrame {
    private void fecharButtonActionPerformed(java.awt.event.ActionEvent evt) {
        betess.finalizarEvento(this.eventCombo.getSelectedItem().toString(), this.resField.getText());
        saveNrefresh();
    }
}

public class BetESS implements Serializable {
    public void finalizarEvento(String jogo, String res){
        String[] jogoEquipas = jogo.split(" X ");
        for(Evento e : getEventosAtivos()){
            if(e.getEquipaCasaNome().equals(jogoEquipas[0]) && e.getEquipaForaNome().equals(jogoEquipas[1])){
                e.setEstado(false);
                e.setResultado(res);
                e.distribuirPremios(getApostadores());
                popupWindow(1, "Evento encerrado e prêmios distribuídos.", "Sucesso");
            }
        }
    }
    public ArrayList<Evento> getEventosAtivos(){
        return eventos.stream()
            .filter(e -> e.getEstado())
            .collect(Collectors.toCollection(ArrayList::new));
    }
}

public class Evento implements Serializable{
    public Integer getVencedor(){
        if(resultado.equals("")) return 2;
        String[] venc = resultado.split("-");
        if(Integer.parseInt(venc[0])>Integer.parseInt(venc[1])) return 1;
        else if(Integer.parseInt(venc[1])>Integer.parseInt(venc[0])) return 3;
        else return 2;
    }
    public void distribuirPremios(ArrayList<Apostador> aps){
        for(Apostador a : aps){
            for(Aposta ap : a.getApostas()){
                if(ap.getEvento().equals(this)){
                    a.adicionarCoins(ap.ganhos(false));
                    ap.notificaApostador();
                }
            }
        }
    }
}

public class Apostador implements Serializable{
    public double ganhos(boolean hipotetico){
        if(hipotetico || palpite==evento.getVencedor()){
            if(palpite==1) return evento.getOddE()*valor;
            else if(palpite==2) return evento.getOddD()*valor;
            else if(palpite==3) return evento.getOddE()*valor;
        }
        return 0.0;
    }
}

```

Figura 4: *Refactoring* do processo de encerrar um evento.

Este processo foi aplicado para todos os métodos considerados complexos e de proporções gigantescas como os últimos dois exemplificados.

2.1.3 Data clumps

Um *Data clump* é um *code smell* detetado quando um conjunto de variáveis são passadas sempre em conjunto. São tipicamente valores primitivos que podem ser passados como um objeto, mas são frequentemente esquecidos ou imperceptíveis aos olhos do *software developer*.

No nosso caso podemos destacar os valores das *odds* dos eventos. São passados como valores sem ligação alguma entre si, mas têm tendência a andar sempre juntos, normalmente no construtor da classe **Evento** ou em parâmetros de métodos.

```

public Evento(int id, double oddV, double oddE, double oddD,
             boolean estado, String resultado, Equipa c, Equipa f){
    this.id = id;
    this.oddV = oddV;
    this.oddE = oddE;
    this.oddD = oddD;
    this.estado = estado;
    this.resultado = resultado;
    this.equipaC = c;
    this.equipaF = f;
}

```

Figura 5: Parâmetros do construtor de Evento.

Assim sendo, vamos passar as *odds* como um objeto. Para tal criámos a classe *Odds* com as três variáveis para o caso de vitória, empate ou derrota da equipa caseira e passámos a trabalhar com os três valores agregados neste novo objeto, tal como se pode ver no construtor da classe *Evento*.

```

public Evento(int id, Odds odds, boolean estado,
             String resultado, Equipa c, Equipa f){
    this.id = id;
    this.odds = odds;
    this.estado = estado;
    this.resultado = resultado;
    this.equipaC = c;
    this.equipaF = f;
}

```

Figura 6: Parâmetros do construtor de Evento após *refactoring*.

2.2 *Change preventers*

Estes *code smells* significam que uma modificação necessária numa porção do código implica várias modificações noutras locais também. O desenvolvimento do *software* torna-se muito mais complicado e dispendioso uma vez que se o tempo da modificação a implementar é superior. Ao longo da análise do nosso código, foram encontrados os seguintes *smells* deste tipo:

2.2.1 *Divergent change*

Este *smell* ocorre quando uma classe está sujeita a mudanças de diversos tipos e por diversos motivos. O código é desorganizado e de difícil compreensão, podendo até conter funcionalidades duplicadas.

Este *smell* é bastante intenso no nosso código uma vez que uma parte significativa da lógica do negócio do sistema encontra-se nas classes dos *forms*, logo, estas classes estão sujeitas a modificações por diversos motivos.

O método de resolução aplicado foi já aplicado na resolução de outros *smells*. Optámos pela decomposição dos métodos que evidenciam este *smell* por vários métodos

auxiliares nas suas classes adequadas. Podemos tomar como exemplo o processo de encerramento de um evento, na Figura 3. Este método está sujeito a diversas modificações por diversos motivos pelo que foi *refactored* para o bloco de código que se apresenta na Figura 4. O processo encontra-se agora dividido pelas suas diversas tarefas nas suas classes adequadas, assim, uma mudança numa tarefa estará relacionada à sua classe pertencente.

A redução deste *smell* deve ser equilibrada, pois a excessiva divisão das tarefas por diversas classes provoca outro *smell*, *shotgun surgery*, causado quando uma simples modificação envolve a alteração de várias classes.

2.3 *Dispensables*

São *code smells* baseados à volta de porções de código dispensáveis, cuja ausência simplificaria o código sem efeito no seu funcionamento. Encontramos os seguintes *smells* deste tipo:

2.3.1 Comentários

Comentários a um método são, por norma, boa prática, na medida em que auxilia na comprehensibilidade do código. No entanto, comentários desnecessários e excessivos são de evitar. Um método já deve ser claro o suficiente para que não precise de explicações. A sua clareza deriva de um nome do método e dos parâmetros expressivos e de uma estrutura coerente e simples.

No nosso caso, não existiam comentários, mas a sua necessidade era evidente. Exemplos disso são os nove métodos de efetuar apostas, responsáveis por todo o processo:

```
private void j1BetActionPerformed(java.awt.event.ActionEvent evt) {...39 lines}
private void j3BetActionPerformed(java.awt.event.ActionEvent evt) {...39 lines}
private void j2BetActionPerformed(java.awt.event.ActionEvent evt) {...39 lines}
private void j4BetActionPerformed(java.awt.event.ActionEvent evt) {...39 lines}
private void j5BetActionPerformed(java.awt.event.ActionEvent evt) {...39 lines}
private void j6BetActionPerformed(java.awt.event.ActionEvent evt) {...39 lines}
private void j7BetActionPerformed(java.awt.event.ActionEvent evt) {...39 lines}
private void j8BetActionPerformed(java.awt.event.ActionEvent evt) {...39 lines}
private void j9BetActionPerformed(java.awt.event.ActionEvent evt) {...39 lines}
```

Figura 7: Métodos com nomes pouco expressivos.

Ao invés disso, optamos pela separação de métodos complexos em porções simples e com nomes significativos e explicativos da função desempenhada pelo método. Tal pode ser comprovado pelos nomes dos novos métodos:

```

public boolean efetuarAposta(Aposta aposta){...10 lines}

public boolean podeApostar(Aposta aposta){...23 lines}

```

Figura 8: Nomes expressivos atribuídos aos novos métodos.

2.3.2 Código duplicado

Código duplicado causa um aumento da complexidade estrutural do sistema e do seu tamanho, consequentemente tornando o seu suporte mais dispendioso.

No nosso sistema, os métodos de efetuar aposta na classe `Home` apresentavam bastante código duplicado. Esse problema já foi resolvido por *smells* abordados anteriormente, como é possível observar. De facto, o *smell* de métodos longos, encontrado nos métodos referidos, foi resolvido recorrendo exatamente à eliminação de código duplicado.

No entanto, mais ocorrências de código duplicado foram detetadas no sistema. Como se pode comprovar nas classes `Home` (método `notificacoes`, linhas 1168-1179) e `Admin` (método `fecharButton`, linhas 282-295), o código patente nessas linhas é igual, pelo que podemos transferir esse código para um novo método, eliminando repetições desnecessárias. Esse método foi chamado `getVencedor()`, encontra-se na classe `Evento` e retorna 1, 2 ou 3 caso o vencedor do evento tenha sido a equipa da casa, empate, ou a equipa de fora, respetivamente.

```

public Integer getVencedor(){
    if(resultado.equals("")) return 2;
    String[] venc = resultado.split("-");
    if(Integer.parseInt(venc[0])>Integer.parseInt(venc[1])) return 1;
    else if(Integer.parseInt(venc[1])>Integer.parseInt(venc[0])) return 3;
    else return 2;
}

```

Figura 9: Método `getVencedor()` que elimina código duplicado.

Temos ainda mais ocorrências de código duplicado na obtenção dos ganhos de uma aposta: nas linhas 300-303 da classe `Admin`, são adicionados os prémios ao apostador, dependendo do seu palpite para o evento (1, 2 ou 3); nas linhas 1180-1183 da classe `Home`, calculamos os ganhos resultantes de um evento, para apresentar o resultado final de um evento ao apostador; por fim, nas linhas 84-92 da classe `MinhasApostas`, são calculados os hipotéticos ganhos que uma aposta poderá render caso o apostador acerte no seu palpite.

Estes blocos de código foram substituídos pelo método `ganhos(boolean hipotetico)` que calcula os ganhos reais gerados por uma aposta (caso `hipotetico==false`) ou os ganhos possíveis se o apostador acertar no palpite (caso `hipotetico==true`).

```

public double ganhos(boolean hipotetico){
    double d[] = {0.0d, evento.getOddV(), evento.getOddE(), evento.getOddD()};
    if(hipotetico || palpite==evento.getVencedor())
        return d[palpito]*valor;
    return d[0];
}

```

Figura 10: Método `ganhos(boolean hipotetico)` que calcula os ganhos possíveis ou efetivos de uma aposta.

Para além dos blocos de código referidos, foram ainda removidos mais alguns duplicados, embora com grau de complexidade inferior. Entre esses situam-se os métodos `saveNrefresh()` existente na maioria das classes de *forms*, responsável por simplesmente salvar o progresso e atualizar o *form* atual e o método `popupWindow(int tipo, String mensagem, String titulo)` que simplifica blocos de código semelhantes ao da imagem abaixo, para um método mais simples e rápido de escrever.

```

ImageIcon icon = new ImageIcon(getClass().getClassLoader().getResource("resources/icons/forbidden.png"));
JOptionPane.showMessageDialog(null, "Dados incorretos!", "Aviso", JOptionPane.INFORMATION_MESSAGE, icon);

```

Figura 11: Código complicado de ler e escrever, simplificado pela `popupWindow`.

2.3.3 Código “morto”

Este *code smell* define-se como uma secção de um método, um método ou até uma classe que não tem impacto no sistema de qualquer forma. Estes frequentemente resultam de correções no código após as quais foi negligenciada a limpeza do código. Nestes casos, a solução comum é simplesmente eliminar o código morto por completo.

No caso do sistema *BetESS*, tais ocorrências residem principalmente em importações de bibliotecas de classes que, ao longo da implementação, caíram em desuso e *setters* gerados que não foram e muito provavelmente não serão utilizados futuramente.

Alguns desses *setters* envolvem *setters* de variáveis imutáveis como IDs, Email, Nome da equipa, Evento de uma aposta, entre outras. Outros são *setters* de variáveis que não são alteradas no decorrer da execução como as *Odds*, Símbolo da equipa, entre outras.

2.4 *Couplers*

Este tipo de *code smells* são relativos ao excesso de acoplamento entre classes, derivando problemas de interdependência entre classes.

2.4.1 *Feature envy*

Feature envy define-se como a dependência excessiva aos dados de outro objeto que não a sua classe. Isto é, um método prefere pedir a uma classe as variáveis

necessárias para computar um método, do que pedir à classe para computar ela própria o método.

No caso do sistema *BetESS*, este *smell* está particularmente patente nas classes dos *forms*. Estas classes têm como único atributo uma instância de *BetESS* que contém todos os dados do sistema e, como todo o processo lógico é executado nestas classes, estas acabam por ter acesso a níveis profundos desta instância, acedendo a classes como *Evento*, *Apostador*, *Equipa*, etc.

Para resolver este problema foram analisadas as instâncias dos objetos consultados e atualizados no decorrer de cada método, **realocando os métodos** ou uma porção destes para a classe mais “correta”. Sigamos o exemplo abaixo: antes recolhíamos a variável email e efetuávamos a comparação neste método; na versão sem *feature envy*, a comparação é efetuada na classe onde se encontra o email, retornando o resultado da comparação.

```
antes
if(a.getEmail().equals(email))

depois
if(ap.checkEmail(email))
```

Figura 12: Verificar se dois emails são iguais.

Outro exemplo: o método **efetuarAposta**. Construído para reduzir o tamanho e eliminar código duplicado nos métodos de efetuar apostas na classe de *forms* *Home*, este método (bem como grande parte dos métodos extraídos durante todo o processo de *refactoring*) foi alocado na classe *BetESS*, recebendo como argumento o seu apostador e a aposta a efetuar.

```
public boolean efetuarAposta(Aposta a, Apostador ap){
    if(podeApostar(a,ap)){
        ap.efetuarAposta(a);
        popupWindow(1, "Aposta registada com sucesso!", "Sucesso");
        return true;
    }
    return false;
}
```

Figura 13: **efetuarAposta** na classe *BetESS*.

Porém tornou-se evidente que a classe correta deste método é *Apostador*, uma vez que as apostas são registadas na instância do seu apostador. É redundante colocar um método na classe *BetESS* que pede ao *Apostador* para registrar uma aposta, quando o método pode ir diretamente para a classe *Apostador*. Abaixo temos o método realocado para a nova classe:

```

public boolean efetuarAposta(Aposta aposta){
    if(podeApostar(aposta)){
        apostas.add(aposta);
        levantarCoins(aposta.getValor());
        BetESS ex = new BetESS();
        ex.popupWindow(1, "Aposta registada com sucesso!", "Sucesso");
        return true;
    }
    return false;
}

```

Figura 14: `efetuarAposta` na classe *Apostador*.

2.4.2 Message chains

Uma *message chain* define-se como uma instância na qual um objeto efetua um pedido a outro objeto, que por sua vez efetua um pedido a outro e assim em diante. Estas correntes implicam uma dependência desnecessária entre a instância e as classes da corrente. Este *smell* origina da negligência do *software developer* relativamente ao respeito da Lei de *Demeter*.

Estes *code smells* são resolvidos pela Lei de *Demeter* que decorre do princípio “*tell, don't ask*” através de criação de métodos que funcionam como **intermediários**, responsáveis por transmitir a informação desejada, de uma classe para a outra, de modo a um objeto conhecer apenas o seu objeto imediato. Estas cadeias também tendem a serem desnecessárias se os métodos forem criados na classe mais adequada.

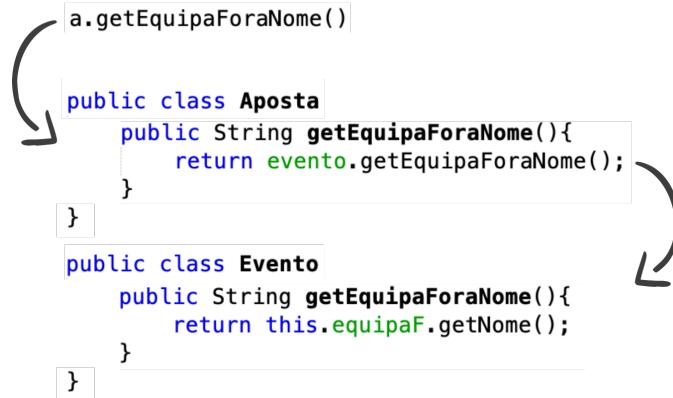
Métodos como os apresentados abaixo foram construídos de modo a reduzir a dependência entre classes causada pelas cadeias de chamadas.

antes

`a.getEvento().getEquipaF().getNome()`

depois

`a.getEquipaForaNome()`



```

public class Aposta {
    public String getEquipaForaNome(){
        return evento.getEquipaForaNome();
    }
}

public class Evento {
    public String getEquipaForaNome(){
        return this.equipaF.getNome();
    }
}

```

Figura 15: Mesmo processo, mas respeitando a Lei de *Demeter* agora.

3 Métricas

Para determinar o impacto das alterações do código na funcionalidade do sistema, foi necessário calcular um conjunto de métricas através das quais se possa demonstrar que a funcionalidade inicialmente implementada se mantém intacta. Tendo desenvolvido uma interface gráfica que já demonstra a maioria desta por defeito simplifica significativamente este esforço.

Definiu-se primeiramente um conjunto de ações a efetuar no sistema para testar a sua funcionalidade:

- Execução de *login* má sucedida.
- Execução de *login* como apostador.
- Verificação do saldo após o *login*.
- Execução de uma aposta.
- Verificação do saldo após a execução da aposta.
- Logout da conta do apostador.
- Registo de um novo apostador.
- Execução, como o novo apostador, de uma aposta no mesmo evento, mas para um resultado diferente.
- Execução de *login* como administrador.
- Finalização do evento no qual os apostadores anteriores apostaram, escolhendo o resultado no qual o primeiro apostador apostou.
- Criação de um novo evento.
- Finalização de um evento no qual os apostadores não apostaram.
- Nova execução de *login* do primeiro apostador.
- Nova execução de *login* do novo apostador, que deve receber uma notificação que perdeu a aposta.

Este conjunto de ações faz uso de todas as funcionalidade presentes no programa. Como tal, caso os resultados obtidos da sua execução no programa atualizado forem idênticos ao do programa original, é possível admitir que as alterações efetuadas aumentaram a legibilidade e a limpeza do código, mantendo no entanto a funcionalidade do sistema.

De facto, nota-se que cada um destes atos produz o mesmo efeito sobre ambas versões do sistema:

- Uma tentativa de *login* má sucedida produz uma notificação que informa o utilizador deste facto.

- Ao fazer *login*, um apostador é levado para a janela dos Eventos Ativos.
- Ao executar a aposta, o sistema notifica o apostador a confirmar a ação, é subtraído o valor da aposta à conta do apostador e é adicionada uma entrada na lista de apostas efetuadas.
- Após efetuar a aposta, o apostador não pode efetuar uma nova aposta no mesmo evento.
- Efetuar o *logout* leva o utilizador de volta ao menu de *login*.
- Ao selecionar o botão 'Registo', o utilizador é levado a um menu onde especifica o seu nome, email e palavra passe, bem como a quantidade inicial de *BetESScoins* da conta.
- Ao finalizar um evento, o administrador deve receber uma notificação a confirmar a ação.
- Ao criar um evento, o administrador deve receber uma notificação a confirmar a ação.
- Ao executar um novo *login*, o primeiro apostador deve receber uma notificação o evento em que apostou terminou e que ganhou a aposta. Não deve receber qualquer notificação relativa ao evento em que não apostou.
- O primeiro apostador deve verificar que o evento em que apostou já não se encontra disponível, que o valor dos ganhos da aposta foi adicionado à conta e que foi removida a respetiva entrada da lista de apostas efetuadas. Ainda mais, deverá ser possível verificar que outro evento terminou e um terceiro foi adicionado.
- Ao executar um novo *login*, o segundo apostador deve receber uma notificação o evento em que apostou terminou e que perdeu a aposta. Não deve receber qualquer notificação relativa ao evento em que não apostou.
- O segundo apostador deve verificar que o evento em que apostou já não se encontra disponível, que o valor de *BetESScoins* da conta se manteve e que foi removida a respetiva entrada da lista de apostas efetuadas. Ainda mais, deverá ser possível verificar que outro evento terminou e um terceiro foi adicionado.

4 Conclusão

Comparando o código do sistema antes e depois do *refactoring*, nota-se uma melhoria significativa na qualidade geral do código, sendo este bastante mais limpo e fácil de navegar. Deste modo, torna também mais simples o processo de alterar ou adicionar funcionalidades caso se torne necessário, reduzindo possíveis custos futuros, quer monetários, quer em termos de tempo despendido.

Concluindo, o *refactoring* da plataforma *BetESS* revelou-se bastante produtivo. Desta forma obteve-se uma base sólida no que toca à deteção de uma variedade de *code smells* e métodos para os combater. A aprendizagem obtida na implementação destes foi também essencial para aumentar a extensibilidade e manutenibilidade do *software* desenvolvido.

Referências

- [1] Alexander Shvets, Gerhard Frey, Marina Pavlova. *Refactoring*. SourceMaking, 2018.
- [2] André L. Ferreira, João Luís Sobral. *Slides de Arquiteturas de Software*. Universidade do Minho, 2018.