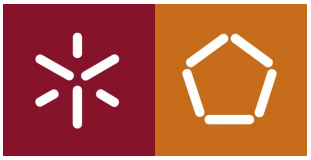


Universidade do Minho

Mestrado Integrado em Engenharia Informática

Arquiteturas de Software 18/19

BetESS - Refactoring

João Pedro Ferreira Vieira A78468
Simão Paulo Leal Barbosa A77689

4 de Janeiro de 2019

Conteúdo

1 Introdução

Este documento relata o processo de desenvolvimento do segundo trabalho prático da unidade curricular de Arquiteturas de Software.

O objetivo deste projeto passa por analisar o código desenvolvido no primeiro enunciado desta disciplina (aplicação de apostas desenvolvida sem padrões de *software*), realizar o levantamento dos *bad smells* que encontramos no mesmo e, através das técnicas de *refactoring* conhecidas e estudadas, modificar o projeto obtendo uma nova versão do mesmo.

Por fim, e usando métricas de comparação de *software*, são comparadas as duas versões do trabalho prático obtidas.

2 Refactoring

Na presente secção ilustraremos quais os *bad smells* por nós identificados em cada uma das classes existentes no nosso projeto e seus respetivos métodos, mostrando de seguida qual o processo utilizado de forma a colmatá-los.

2.1 *BetESSController*

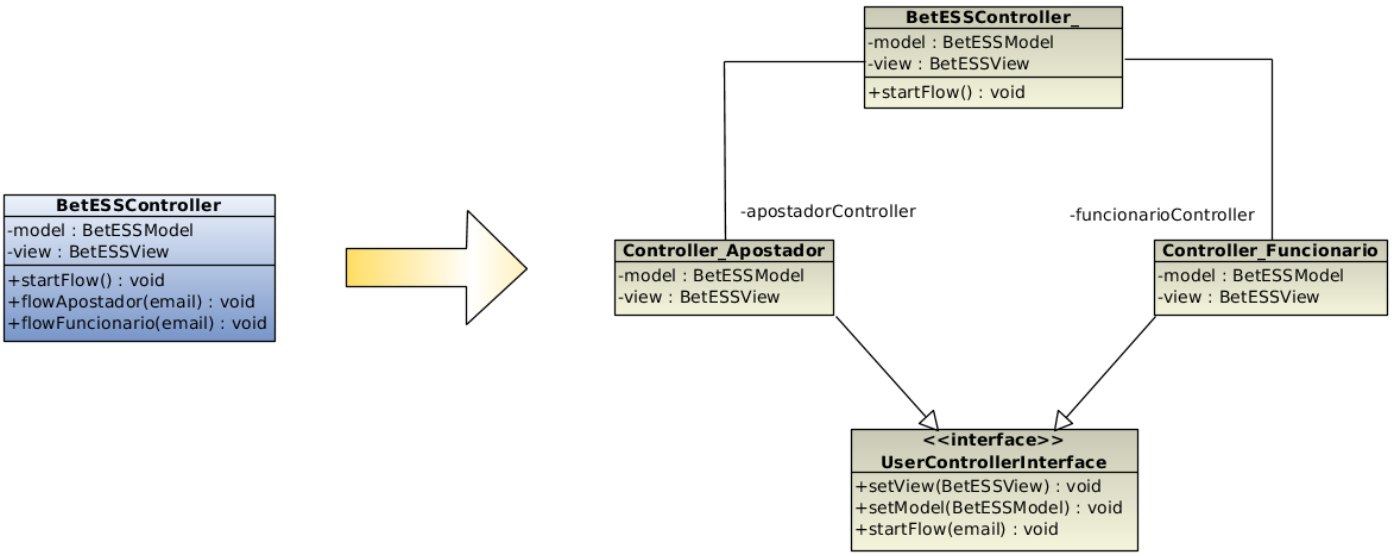
2.1.1 *Large Class*

Nesta classe que representa o controlador do modelo *MVC* por nós implementado, começámos por detetar de forma instantânea que a mesma é uma classe demasiadamente grande, ou seja, enquadra-se no tipo de *bad smell* conhecido por ***Large Class***.

De forma a resolver este problema, consideramos uma boa opção dividir o controlador elaborado por diferentes controladores, passando a existir um controlador principal (`BetESSController`), e dois controladores secundários, `Controller_Apostador` e `Controller_Funcionario`, que passam a implementar uma interface `UserControllerInterface`, e que contém as operações à disposição dos apostadores e funcionários, respetivamente.

Sendo assim, para aplicar o *refactoring* pretendido, são usadas as técnias *Extract Class* e *Extract Interface*.

O processo de transformação das classes envolvidas neste processo pode em baixo ser mais facilmente compreendido:



2.1.2 *Long Method & Duplicate Code*

No método responsável por efetuar a autenticação de um utilizador (`login()`), é possível de verificar que o mesmo tem um tamanho um pouco superior ao desejável (*Long Method*), sendo que se procedeu à extracção de um método (*Extract Method*) `loginUtilizador(int mode, String email)` para realizar partes até similares de código tendo em conta o *login* como utilizador ou funcionário (*Duplicate Code*).

O método inicialmente tomava a seguinte forma:

```
private void login(){
    Scanner scan = new Scanner(System.in);
    view.println("Insira os seus dados:");
    view.print("Email: ");
    String email = scan.nextLine();
    view.print("Password: ");
    String password = scan.nextLine();
    int login = this.model.login(email, password);
    String nome;
    switch (login) {
```

```
        case 1:
            nome = model.getUtilizador(email).getNome();
            view.println("Login como apostador efetuado com sucesso. Bem-vindo " + nome + "!");
            this.apostadorController.startFlow(email);
            break;
        case 2:
            nome = model.getUtilizador(email).getNome();
            view.println("Login como funcionário efetuado com sucesso. Bem-vindo " + nome + "!");
            this.funcionarioController.startFlow(email);
            break;
        case 0:
            view.println("Password inserida está incorreta");
            break;
        case -1:
            view.println("Não existe o utlizador com o email inserido");
            break;
        default:
            break;
    }
}
```

Com o *refactoring* efetuado e com alguns correções para uma melhor perceção do código, o resultado obtido passa a ser o seguinte:

```
private void login(){
    Scanner scan = new Scanner(System.in);
    view.println("Insira os seus dados:");
    view.print("Email: ");
    String email = scan.nextLine();
    view.print("Password: ");
    String password = scan.nextLine();
    int login = this.model.login(email, password);
    switch (login) {
        case 1:
            loginUtilizador(1,email); break;
        case 2:
            loginUtilizador(2,email); break;
        case 0:
            view.println("Password inserida está incorreta"); break;
        case -1:
            view.println("Não existe o utlizador com o email inserido"); break;
        default: break;
    }
}

// mode = 1 -> apostador, mode = 2 -> funcionário
private void loginUtilizador(int mode, String email){
    String nome = model.getUtilizador(email).getNome();
    String user = (mode == 1) ? "apostador" : "funcionário";
    view.println("Login como " + user + " efetuado com sucesso. Bem-vindo " + nome + "!");
    if (mode == 1) this.apostadorController.startFlow(email);
    else this.funcionarioController.startFlow(email);
}
```

2.1.3 *Message Chains*

É possível verificar a existência de *Message Chains*, por exemplo, no método `loginUtilizador(int mode, String email)` apresentado em cima, quando é feito `model.getUtilizador(email).getNome()` neste método ou noutros do controlador.

Para resolver este problema é utilizado *Hide Delegate*, sendo que se adiciona um método `getNomeUtilizador(String email)` à classe `BetESSModel`, que passa a ser chamado pelos métodos pertencentes ao controlador.

Outro exemplo de *Message Chains* no controlador da aplicação passa pela utilização tanto de `((Apostador) this.model.get Utilizador(email)).getSaldo()` como de `((Apostador) this.model.getUtilizador(email)).setSaldo(novo_saldo)`. Tal como realizado no exemplo anterior, é criado um método `getSaldoApostador(String email)` e um método `setSaldoApostador(String email, double valor)` em `BetESSModel`, que passam a ser utilizados nos métodos internos do controlador.

2.1.4 *Controller_Apostador - Long Method*

Na nova classe `Controller_Apostador` criada, é possível verificar que dois métodos têm um tamanho superior ao desejável: `startFlow(String email)` e `novaAposta(String email)`.

Em `startFlow(String email)`, o estado do método era o seguinte:

```
public void startFlow(String email) {
    Menu menu = view.getMenu(2);
    Apostador a = (Apostador) this.model.getUtilizador(email);
    String opcao;
    do {
        List<String> noti = a.getNotificacoes();
        if (noti.isEmpty()) menu.show();
        else this.view.menuApostadorNotificacoes(noti.size()).show();
        Scanner scan = new Scanner(System.in);
        opcao = scan.next();
        opcao = opcao.toUpperCase();
        switch(opcao) {
            case "N" :
                if (noti.isEmpty()){ System.out.println("Opcão Inválida!"); break;}
                this.view.printNotificacoes(noti);
                a.cleanNotificacoes();
                break;
            case "E" :
                mostrarEventos();
                break;
            case "V" :
                verApostasRealizadas(email);
                break;
            case "A" :
                novaAposta(email);
                break;
            case "C" :
                imprimeSaldo(email);
                break;
            case "I" :
                carregarConta(email);
                break;
            case "S":
                break;
            default: view.println("Opcão Inválida !"); break;
        }
    } while(!opcao.equals("S"));
    view.println("Até à próxima visita " + model.getNomeUtilizador(email) + "!");
}
```

A opção tomada para fazer *refactoring* deste método passou por se fazer *Extract Method* de partes do código. A parte responsável por verificar se o apostador tem notificações (e mostrar o respetivo menu tendo isso em conta) e realizar a leitura da opção tomada passa a estar num método nomeado `private String lerOpcao(Menu menu, List<String> noti)`. Já a parte responsável por mostrar as notificações e "limpar"a caixa de notificações do apostador passa a estar em `private void verNotificacoes(List<String> noti, Apostador a)`. O resultado obtido não coloca o método principal com um tamanho aproximado de 10 linhas, mas coloca-o mais legível. Sendo assim, o resultado obtido é o seguinte:

```
public void startFlow(String email){
    Menu menu = view.getMenu(2);
    Apostador a = (Apostador) this.model.getUtilizador(email);
```

```
    String opcao;
    do {
        List<String> noti = a.getNotificacoes();
        opcao = lerOpcao(menu, noti);
        switch(opcao) {
            case "N" :
                verNotificacoes(noti, a); break;
            case "E" :
                mostrarEventos(); break;
            case "V" :
                verApostasRealizadas(email); break;
            case "A" :
                novaAposta(email); break;
            case "C" :
                imprimeSaldo(email); break;
            case "I" :
                carregarConta(email); break;
            case "S": break;
            default: view.println("Opcão Inválida !"); break;
        }
    } while(!opcao.equals("S"));
    view.println("Até à próxima visita " + model.getNomeUtilizador(email) + "!");
}

private String lerOpcao(Menu menu, List<String> noti){
    if (noti.isEmpty()) menu.show();
    else this.view.menuApostadorNotificacoes(noti.size()).show();
    Scanner scan = new Scanner(System.in);
    String opcao = scan.next().toUpperCase();
    return opcao;
}

private void verNotificacoes(List<String> noti, Apostador a){
    if (noti.isEmpty()){
        System.out.println("Opcão Inválida!");
        return;
    }
    this.view.printNotificacoes(noti);
    a.cleanNotificacoes();
}
```

Já em `novaAposta(String email)`, o estado inicial do mesmo é o seguinte:

```
private void novaAposta(String email){
    String opcao;
    view.println("Insira o id do evento em que pretende apostar:");
    Scanner scanI = new Scanner(System.in);
    int id = scanI.nextInt();
    if (!this.model.existeEvento(id)){
        view.println("O evento com o id inserido não existe");
        return;
    }
    else if (!this.model.getEvento(id).getDisponibilidade()){
        view.println("O evento escolhido não está disponível para apostas de momento!");
        return;
    }
    Apostador apostador = (Apostador) this.model.getUtilizador(email);
    view.println("Indique a quantia que pretende a apostar: (Pode apostar até " + apostador.getSaldo()
                 + " ESScoins)");
    Scanner scanD = new Scanner(System.in);
```

```
    double quantia = scanD.nextDouble();
    if (!apostador.saldoSufiente(quantia)){
        view.println("O seu saldo é insuficiente para realizar a aposta desejada");
        return;
    }
    Evento evento = this.model.getEvento(id);
    do{
        this.view.menuEquipas(evento, quantia);
        view.println("Insira a sua escolha:");
        Scanner scan = new Scanner(System.in);
        opcao = scan.next();
        opcao = opcao.toUpperCase();
        Integer[] resultados= new Integer[2];
        resultados[0]= -1;
        double[] quantiaOdd = new double[2];
        switch(opcao) {
            case "1" :
                resultados[1] = 0;
                quantiaOdd[0] = quantia;
                quantiaOdd[1]= evento.getOdds()[0];
                apostador.newAposta(resultados,quantiaOdd, evento); view.println("Aposta realizada na equipa "
                                                                                + evento.getEquipa_1() + "!");
                evento.addApostador(email);
                return;
            case "X" :
                resultados[1] = 1;
                quantiaOdd[0] = quantia;
                quantiaOdd[1]= evento.getOdds()[1];
                apostador.newAposta(resultados,quantiaOdd, evento); view.println("Aposta realizada no empate!");
                evento.addApostador(email);
                return;
            case "2" :
                resultados[1] = 2;
                quantiaOdd[0] = quantia;
                quantiaOdd[1]= evento.getOdds()[2];
                apostador.newAposta(resultados,quantiaOdd, evento); view.println("Aposta realizada na equipa "
                                                                                + evento.getEquipa_2() + "!");
                evento.addApostador(email);
                return;
            case "S":
                break;
            default:
                view.println("Opcão Inválida! Tente de novo.");
                break;
        }
    } while(!opcao.equals("S"));
}
```

Para "reparar"este método procedeu-se à utilização de *Extract Method*, até mesmo tendo em conta partes do método (que trata os 3 tipos de resultados possíveis) enquadrarem-se no *bad smell* do tipo *Duplicate Code*.

A parte do método responsável por ler o *id* do evento em que se pretende apostar deu origem ao método `lerIdEvento()`, A parte responsável por ler a quantia a apostar passou a estar em `lerQuantia(Apostador a)`. Já a parte em que se lê e trata o resultado apostado está agora em `lerAposta(Apostador apostador, Evento evento, double quantia)`, onde é utilizado o método `apostaRealizada(Apostador a, Evento e, double quantia, int i)` criado tendo em conta a existência de código duplicado e semelhante.

Desta forma, o resultado final é o seguinte:

```
private void novaAposta(String email){
    view.println("Insira o id do evento em que pretende apostar:");
    int id = lerIdEvento();
```

```
    if (id == -1) return;
    Apostador apostador = (Apostador) this.model.getUtilizador(email);
    view.println("Indique a quantia que pretende a apostar: (Pode apostar até " + apostador.getSaldo()
                + " ESScoins)");
    double quantia = lerQuantia(apostador);
    if (quantia == -1) return;
    Evento evento = this.model.getEvento(id);
    lerAposta(apostador, evento, quantia);
}

private int lerIdEvento(){
    Scanner scanI = new Scanner(System.in);
    int id = scanI.nextInt();
    if (!this.model.existeEvento(id)){
        view.println("O evento com o id inserido não existe");
        return -1;
    }
    else if (!this.model.getEvento(id).getDisponibilidade()){
        view.println("O evento escolhido não está disponível para apostas de momento!");
        return -1;
    }
    return id;
}

private double lerQuantia(Apostador a){
    Scanner scanD = new Scanner(System.in);
    double quantia = scanD.nextDouble();
    if (!a.saldoSufiente(quantia)){
        view.println("O seu saldo é insuficiente para realizar a aposta desejada");
        return -1;
    }
    return quantia;
}

private void lerAposta(Apostador apostador, Evento evento, double quantia){
    String opcao;
    do{
        this.view.menuEquipas(evento, quantia);
        view.println("Insira a sua escolha:");
        Scanner scan = new Scanner(System.in);
        opcao = scan.next().toUpperCase();
        switch(opcao) {
            case "1" :
                apostaRealizada(apostador, evento, quantia, 0); return;
            case "X" :
                apostaRealizada(apostador, evento, quantia, 1); return;
            case "2" :
                apostaRealizada(apostador, evento, quantia, 2); return;
            case "S": break;
            default: view.println("Opcão Inválida! Tente de novo."); break;
        }
    } while(!opcao.equals("S"));
}

private void apostaRealizada(Apostador a, Evento e, double quantia, int i){
    a.newAposta(new Integer[]{-1,i}, new double[]{quantia, e.getOdds()[i]}, e);
    switch (i){
        case 0:
            view.println("Aposta realizada na equipa " + e.getEquipa_1() + "!"); break;
```

```
        case 1:
            view.println("Aposta realizada no empate!"); break;
        default:
            view.println("Aposta realizada na equipa " + e.getEquipa_2() + "!"); break;
    }
    e.addApostador(a.getEmail());
}
```

2.1.5 *Controller_Funcionario - Long Method*

Verifica-se que o método responsável por tornar posssível um funcionário adicionar eventos à aplicação possui um tamanho maior que o desejável (`adicionarEvento()`). Este método estava escrito da seguinte forma:

```
private void adicionarEvento(){
    Scanner scan = new Scanner(System.in);
    view.println("Insira o nome das 2 equipas envolvidas no jogo:");
    view.print("Equipa no1 : ");
    String equipa_1 = scan.nextLine();
    view.print("Equipa no2 : ");
    String equipa_2 = scan.nextLine();
    Scanner scanD = new Scanner(System.in);
    view.println("Insira as odd's para os 3 possíveis resultados:");
    view.print("Vitória do/da " + equipa_1 + ": ");
    double odd_1 = scanD.nextDouble();
    view.print("Empate: ");
    double odd_x = scanD.nextDouble();
    view.print("Vitória do/da " + equipa_2 + ": ");
    double odd_2 = scanD.nextDouble();
    view.print("O evento pode estar disponível de momento (S/N): ");
    String d = scan.nextLine().toUpperCase();
    boolean disponibilidade;
    switch(d){
        case "S": disponibilidade = true; break;
        case "N": disponibilidade = false; break;
        default: view.println("Opção inválida"); return;
    }
    String[] equipas = new String[2];
    equipas[0] =  equipa_1;
    equipas[1] =  equipa_2;
    double[] odds = new double[3];
    odds[0] =  odd_1;
    odds[1] =  odd_x;
    odds[2] = odd_2;
    this.model.addEvento(equipas,odds, disponibilidade);
    view.println("Evento adicionado com sucesso");
}
```

Tal como realizado com outros métodos que "sofriam"do mesmo problema (*Long Method*), este problema foi resolvido repartindo partes do método por novas funções (utilizando *Extract Method*). O código que lê as duas equipas envolvidas no evento está agora em `lerEquipas()`, a parte que lê as *odd's* está em `lerOdds(String equipa_1, String equipa_2)`, e, por fim, a parte do método que lê a disponibilidade com que o evento a ser adicionado deve estar está em `lerDisponibilidade()`.

O resultado final deste *refactoring* é o seguinte:

```
private void adicionarEvento(){
    String[] equipas = lerEquipas();
    double[] odds = lerOdds(equipas[0], equipas[1]);
    view.print("O evento pode estar disponível de momento (S/N): ");
    int d = lerDisponibilidade();
    if (d == -1) return;
    boolean disponibilidade = (d == 1) ? true : false;
```

```
    this.model.addEvento(equipas, odds, disponibilidade);
    view.println("Evento adicionado com sucesso");
}

private String[] lerEquipas(){
    Scanner scan = new Scanner(System.in);
    view.println("Insira o nome das 2 equipas envolvidas no jogo:");
    view.print("Equipa no1 : ");
    String equipa_1 = scan.nextLine();
    view.print("Equipa no2 : ");
    String equipa_2 = scan.nextLine();
    return new String[]{equipa_1, equipa_2};
}

private double[] lerOdds(String equipa_1, String equipa_2){
    Scanner scanD = new Scanner(System.in);
    view.println("Insira as odd's para os 3 possíveis resultados:");
    view.print("Vitória do/da " + equipa_1 + ": ");
    double odd_1 = scanD.nextDouble();
    view.print("Empate: ");
    double odd_x = scanD.nextDouble();
    view.print("Vitória do/da " + equipa_2 + ": ");
    double odd_2 = scanD.nextDouble();
    return new double[]{odd_1, odd_x, odd_2};
}

private int lerDisponibilidade(){
    Scanner scan = new Scanner(System.in);
    view.print("O evento pode estar disponível de momento (S/N): ");
    String d = scan.nextLine().toUpperCase();
    int disp;
    switch(d){
        case "S": disp = 1; break;
        case "N": disp = 2; break;
        default: view.println("Opção inválida");
                 disp = -1;
    }
    return disp;
}
```

Outro método que se tentou reduzir foi o método `flowModificar()`, utilizado para os funcionários alterarem os eventos da aplicação.

O estado inicial do método é o seguinte:

```
private void flowModificar(){
    Scanner scanI = new Scanner(System.in);
    view.print("Insira o id do evento que pretende modificar: ");
    int id = scanI.nextInt();
    if (!this.model.existeEvento(id)){
        view.println("Não exister o evento com o id inserido");
        return;
    }
    Evento evento = this.model.getEvento(id);
    Menu menu = view.getMenu(4);
    String opcao;
    do {
        menu.show();
        Scanner scan = new Scanner(System.in);
        opcao = scan.next();
        opcao = opcao.toUpperCase();
```

```
        switch(opcao) {
            case "D" :
                modificarDisponibilidadeEvento(evento);
                break;
            case "O" :
                modificarOddsEvento(evento);
                break;
            case "E" :
                modificarEquipasEvento(evento);
                break;
            case "S":
                break;
            default: view.println("Opcão Inválida !"); break;
        }
    } while(!opcao.equals("S"));
}
```

Realizou-se *Extract Method* colocando numa nova função a parte do código que lê o evento que se pretende modificar (`lerEvento()`). Para além disso, reorganizou-se o código tornando-o mais compacto e legível.

O resultado obtido é:

```
private void flowModificar(){
    int id = lerEvento();
    if (id == -1) return;
    Evento evento = this.model.getEvento(id);
    Menu menu = view.getMenu(4);
    String opcao;
    do {
        menu.show();
        Scanner scan = new Scanner(System.in);
        opcao = scan.next().toUpperCase();
        switch(opcao) {
            case "D" :
                modificarDisponibilidadeEvento(evento); break;
            case "O" :
                modificarOddsEvento(evento); break;
            case "E" :
                modificarEquipasEvento(evento); break;
            case "S": break;
            default: view.println("Opcão Inválida !"); break;
        }
    } while(!opcao.equals("S"));
}

private int lerEvento(){
    Scanner scanI = new Scanner(System.in);
    view.print("Insira o id do evento que pretende modificar: ");
    int id = scanI.nextInt();
    if (!this.model.existeEvento(id)){
        view.println("Não exister o evento com o id inserido");
        return -1;
    }
    return id;
}
```

Outro método que merecia preocupação no que diz respeito ao tipo de *bad smell Long Method* é o método responsável por permitir a um funcionário terminar um dado evento (`terminarEvento()`), que tinha o seguinte código inicial:

```
private void terminarEvento(){
    view.println("Insira o id do evento que pretende encerrar:");
    Scanner scanI = new Scanner(System.in);
```

```
    int id = scanI.nextInt();
    int m = this.model.mudarDisponibilidadeEvento(id, false);
    if (m == 0){
        view.println("Não existe o evento com o id inserido");
        return;
    }
    else if (m == 1){
        Evento evento = this.model.getEvento(id);
        view.println("Qual o resultado com que o evento terminou?");
        view.println("1 - Vitória do/da " + evento.getEquipa_1());
        view.println("X - Empate");
        view.println("2 - Vitória do/da " + evento.getEquipa_2());
        view.print("Opção : ");
        Scanner scan = new Scanner(System.in);
        String opcao = scan.nextLine().toUpperCase();
        int resultado = -1;
        switch (opcao){
            case "1": resultado = 0; break;
            case "X": resultado = 1; break;
            case "2": resultado = 2; break;
            default: System.out.println("Resultado inválido!"); return;
        }
        Apostador apostador;
        for (String a : evento.getApostadores()){
            apostador = (Apostador) this.model.getUtilizador(a);
            apostador.eventoTerminado(evento.getId(),resultado);
        }
        view.println("Evento encerrado com sucesso");
    }
}
```

Para realizar o *refactoring* deste método, e como habitualmente realizado, recorreu-se a *Extract Method*.
A parte que imprime ao utilizador os resultados nos quais é possível terminar o evento passou a estar disponível num método da *view* (`BetESSView`), com o nome `opcoesTerminarEvento(Evento evento)`. A parte que lê o resultado final do evento e o coloca num inteiro (*int*) passa a estar no método `lerResultado()`. Por último, o código responsável por notificar os apostadores que apostaram no evento fechado ficou no método `notificarApostadores(Evento evento, int resultado)`.

Sendo assim, é em baixo apresentado o resultado final destas modificações:

```
---------------------- Controller_Funcionario ----------------------

private void terminarEvento(){
    view.println("Insira o id do evento que pretende encerrar:");
    Scanner scanI = new Scanner(System.in);
    int id = scanI.nextInt();
    int m = this.model.mudarDisponibilidadeEvento(id, false);
    if (m == 0) {view.println("Não existe o evento com o id inserido"); return;}
    else if (m == 1){
        Evento evento = this.model.getEvento(id);
        this.view.opcoesTerminarEvento(evento);
        int resultado = lerResultado();
        notificarApostadores(evento, resultado);
        view.println("Evento encerrado com sucesso");
    }
}

private int lerResultado(){
    Scanner scan = new Scanner(System.in);
    String opcao = scan.nextLine().toUpperCase();
    int resultado = -1;
    switch (opcao){
```

```
        case "1": resultado = 0; break;
        case "X": resultado = 1; break;
        case "2": resultado = 2; break;
        default: view.println("Resultado inválido!"); break;
    }
    return resultado;
}

private void notificarApostadores(Evento evento, int resultado){
    Apostador apostador;
    for (String a : evento.getApostadores()){
        apostador = (Apostador) this.model.getUtilizador(a);
        apostador.eventoTerminado(evento.getId(),resultado);
    }
}

--------------------------- BetESSView ---------------------------

public void opcoesTerminarEvento(Evento evento){
    System.out.println("Qual o resultado com que o evento terminou?");
    System.out.println("1 - Vitória do/da " + evento.getEquipa_1());
    System.out.println("X - Empate");
    System.out.println("2 - Vitória do/da " + evento.getEquipa_2());
    System.out.print("Opção : ");
}
```

2.2 *Main*

2.2.1 *Long Method*

Identificamos que o método *main* é demasiado longo (*Long Method*) e que podem ser extraídos métodos do mesmo, por isso utilizamos a técnica de refactoring *Extract Method*. O método original é o seguinte:

```
    public static void main(String[] args){

        BetESSModel model;

        model = BetESSPersistency.carregaEstado("dados.obj");

        if (model == null){
            model = new BetESSModel();
            System.out.println("A criar dados...");

            model.addEvento("CD Leganés", "Atlético Madrid", 5.25, 2.85, 1.57, true);
            model.addEvento("Real Madrid CF", "Real Valladolid", 1.19, 6.25, 11.00, true);
            model.addEvento("Valência CF", "Girona CF", 1.47, 3.90, 6.50, true);
            model.addEvento("Rayo Vallecano", "FC Barcelona", 9.00, 5.75, 1.25, true);
            model.addEvento("SD Eibar", "CD Alavés", 2.05, 3.10, 3.60, true);
            model.addEvento("CF Villareal", "UD Levante", 1.57, 4.00, 5.00, true);
            model.addEvento("Real Sociedad", "Sevilha FC", 2.65, 3.30, 2.40, true);
            model.addEvento("SD Huesca", "Getafe CF", 3.50, 3.00, 2.15, true);
            model.addEvento("Bétis Sevilha", "Celta de Vigo", 1.77, 3.60, 4.00, true);

            model.addApostador("antonio@hotmail.com", "12345", "António Silva", 15.60);
            model.addApostador("mafalda@hotmail.com", "11111", "Mafalda Castro", 5.90);
            model.addApostador("carlos@hotmail.com", "22222", "Carlos Sampaio", 3.00);
            model.addApostador("alberto@hotmail.com", "33333", "Alberto Campos", 42.30);

            model.addFuncionario("func1@gmail.com", "111", "Renato Silva");
```

```
            model.addFuncionario("func2@gmail.com", "222", "Catarina Coelho");
        }

        BetESSView view = new BetESSView();

        BetESSController control = new BetESSController();
        control.setModel(model);
        control.setView(view);

        UserControllerInterface apostadorController = new Controller_Apostador();
        apostadorController.setModel(model);
        apostadorController.setView(view);
        UserControllerInterface funcionarioController = new Controller_Funcionario();
        funcionarioController.setModel(model);
        funcionarioController.setView(view);

        control.setApostadorController(apostadorController);
        control.setFuncionarioController(funcionarioController);

        control.startFlow();

        BetESSPersistency.guardaEstado(model, "dados.obj");
    }
```

É percetível que o preenchimento do ficheiro com os dados iniciais deve ser realizado em métodos distintos, separando a adição de funcionários (`addFuncionarios(BetESSModel model)`) da adição de apostadores (`addApostadores(BetESSModel model)`) e da adição de eventos (`addEventos(BetESSModel model)`).

Sendo assim, foram realizadas as seguintes alterações:

```
    public static BetESSModel createData(BetESSModel model){
        if (model == null){
            model = new BetESSModel();
            System.out.println("A criar dados...");
            model = addEventos(model);
            model = addApostadores(model);
            model = addFuncionarios(model);
        }
        return model;
    }
    public static BetESSModel addFuncionarios(BetESSModel model) {
        model.addFuncionario("func1@gmail.com", "111", "Renato Silva");
        model.addFuncionario("func2@gmail.com", "222", "Catarina Coelho");
        return model;
    }
    public static BetESSModel addApostadores(BetESSModel model) {
        model.addApostador("antonio@hotmail.com", "12345", "António Silva", 15.60);
        model.addApostador("mafalda@hotmail.com", "11111", "Mafalda Castro", 5.90);
        model.addApostador("carlos@hotmail.com", "22222", "Carlos Sampaio", 3.00);
        model.addApostador("alberto@hotmail.com", "33333", "Alberto Campos", 42.30);
        return model;

    }
    public static BetESSModel addEventos(BetESSModel model) {
        model.addEvento("CD Leganés", "Atlético Madrid", 5.25, 2.85, 1.57, true);
        model.addEvento("Real Madrid CF", "Real Valladolid", 1.19, 6.25, 11.00, true);
        model.addEvento("Valência CF", "Girona CF", 1.47, 3.90, 6.50, true);
        model.addEvento("Rayo Vallecano", "FC Barcelona", 9.00, 5.75, 1.25, true);
        model.addEvento("SD Eibar", "CD Alavés", 2.05, 3.10, 3.60, true);
        model.addEvento("CF Villareal", "UD Levante", 1.57, 4.00, 5.00, true);
```

```
        model.addEvento("Real Sociedad", "Sevilha FC", 2.65, 3.30, 2.40, true);
        model.addEvento("SD Huesca", "Getafe CF", 3.50, 3.00, 2.15, true);
        model.addEvento("Bétis Sevilha", "Celta de Vigo", 1.77, 3.60, 4.00, true);
        return model;
    }
```

Desta forma, na *main* passa a ser invocado o método `createData(BetESSModel model)`:

```
public static void main(String[] args){
    ...
        BetESSModel model;

        model = BetESSPersistency.carregaEstado("dados.obj");

        model = createData(model);

        BetESSView view = new BetESSView();
    ...
}
```

2.3 *Aposta*

2.3.1 *Long Method*

O método `toString()` existente em `Aposta` pode ser enquadrado num *Long Method*, visto que, apesar de não ser um método demasiadamente comprido, pode ser melhorado para uma melhor perceção do programador daquilo que é desejado conseguir no seu código. O método original é o seguinte:

```
public String toString() {
    String ra;
    switch (this.resultado_aposta){
        case 0: ra = "1"; break;
        case 1: ra = "X"; break;
        case 2: ra = "2"; break;
        default: ra = ""; break;
    }
    double ganhos;
    ganhos = this.odd * this.quantia;
    return "id = " + this.id + ", jogo: " + this.evento.getEquipa_1() + " X " + this.evento.getEquipa_2()
            + ", aposta realizada = " + ra + ", odd = " + this.odd
            + ", quantia = " + this.quantia + " ESScoins"
            + ", possíveis ganhos = " + ganhos + " ESScoins, estado = " + estado();
}
```

Desta forma, através de *Replace Temp with Query*, foi passado o cálculo dos possíveis ganhos de uma aposta (`ganhos = this.odd * this.quantia;`) para um método à parte, e através de *Extract Method* foi também passado para um novo método o código que calcula o resultado da aposta (`String ra;`). Assim, o resultado obtido com este *refactoring* passa a ser o seguinte:

```
public String toString() {
    return "id = " + this.id + ", jogo: " + this.evento.getEquipa_1() + " X " + this.evento.getEquipa_2()
            + ", aposta realizada = " + resultadoAposta() + ", odd = " + this.odd
            + ", quantia = " + this.quantia + " ESScoins"
            + ", possíveis ganhos = " + possiveisGanhos() + " ESScoins, estado = " + estado();
}

private String resultadoAposta(){
    switch (this.resultado_aposta){
        case 0: return "1";
        case 1: return "X";
        case 2: return "2";
        default: return "";
```

```
    }
}

private double possiveisGanhos(){
    return this.odd * this.quantia;
}
```

2.3.2 *Long Parameter List*

O construtor desta classe apresenta um *bad smell* denominado *Long Parameter List*, resultante do facto da lista de parâmetros do seu construtor apresentar um grande número de parâmetros (6):

```
    public Aposta(int id, Integer resultado_evento, Integer resultado_aposta, double quantia,
                  double odd, Evento evento) {
        this.id = id;
        this.resultado_evento = resultado_evento;
        this.resultado_aposta = resultado_aposta;
        this.quantia = quantia;
        this.odd = odd;
        this.evento = evento;
    }
```

Este problema pode ser resolvido usando a técnica *Introduce Parameter Object*, substituindo os parâmetros do tipo *Integer* e *double* por objetos únicos, da seguinte forma:

```
    public Aposta(int id, Integer[] resultados, double[] quantiaOdd, Evento evento) {
        this.id = id;
        this.resultado_evento = resultados[0];
        this.resultado_aposta = resultados[1];
        this.quantia = quantiaOdd[0];
        this.odd = quantiaOdd[1];
        this.evento = evento;
    }
```

Este tipo de *refactoring*, ainda assim, implicou a mudança do código dos métodos que utilizam o construtor de `Aposta`, de forma a adaptarem-se aos novos parâmetros do mesmo.

2.4 Apostador

2.4.1 *Long Method & Comments*

Tendo em conta o método `eventoTerminado(int idEvento, int resultado)` com a seguinte implementação:

```
public void eventoTerminado(int idEvento, int resultado){
    for (Aposta aposta : this.apostas){
        if (aposta.getIdEvento() == idEvento){
            aposta.setResultado_evento(resultado);
            if (aposta.getResultado_evento() == aposta.getResultado_aposta()){ //ganhou aposta
                this.saldo += aposta.getQuantia() * aposta.getOdd();
                this.notificacoes.add("Ganhou a aposta com o id " + aposta.getId()
                    + ", respetiva ao evento " + aposta.getEvento().getEquipa_1()
                    + " X " + aposta.getEvento().getEquipa_2()
                    + ", o seu saldo foi incrementado em " + aposta.getQuantia() * aposta.getOdd() + " ESScoins")
                        ;
            }
            else{ //perdeu aposta
                this.notificacoes.add("Perdeu a aposta com o id " + aposta.getId()
                    + ", respetiva ao evento " + aposta.getEvento().getEquipa_1()
                    + " X " + aposta.getEvento().getEquipa_2()
                    + ", na qual apostou " + aposta.getQuantia() + " ESScoins");
            }
```

```
        }
    }
}
```

É verificável a existência de comentários para perceber o contexto de se um `Apostador` ganhou ou não uma aposta a ser processada. Desta forma, é boa ideia fazer *Extract Method* e passar a condição que determina tal caso para um método separado com um nome condizente com o mesmo (`ganhouAposta(aposta)`).

Tendo em conta a realização do cálculo `aposta.getQuantia() * aposta.getOdd()` por mais do que uma vez, é utilizada a técnica *Replace Temp with Query* para passar esse cálculo para um novo método (`ganhosAposta(aposta)`).

Para além disto, é realizado *Extract Method* para separar em dois novos métodos a parte do código responsável por adicionar notificações ao `Apostador`: `adicionaNotificacaoVitoria(aposta)` e `adicionaNotificacaoDerrota(aposta)`.

O resultado obtido com estas transformações passa a ser o seguinte:

```
public void eventoTerminado(int idEvento, int resultado){
    for (Aposta aposta : this.apostas){
        if (aposta.getIdEvento() == idEvento){
            aposta.setResultado_evento(resultado);
            if (ganhouAposta(aposta)){
                this.saldo += ganhosAposta(aposta);
                adicionaNotificacaoVitoria(aposta);
            }
            else{
                adicionaNotificacaoDerrota(aposta);
            }
        }
    }
}

public boolean ganhouAposta(Aposta aposta){
    return aposta.getResultado_evento() == aposta.getResultado_aposta();
}

public double ganhosAposta(Aposta aposta){
    return aposta.getQuantia() * aposta.getOdd();
}

public void adicionaNotificacaoVitoria(Aposta aposta){
    this.notificacoes.add("Ganhou a aposta com o id " + aposta.getId()
            + ", respetiva ao evento " + aposta.getEvento().getEquipa_1()
            + " X " + aposta.getEvento().getEquipa_2()
            + ", o seu saldo foi incrementado em " + aposta.getQuantia() * aposta.getOdd() + " ESScoins");
}

public void adicionaNotificacaoDerrota(Aposta aposta){
    this.notificacoes.add("Perdeu a aposta com o id " + aposta.getId()
            + ", respetiva ao evento " + aposta.getEvento().getEquipa_1()
            + " X " + aposta.getEvento().getEquipa_2()
            + ", na qual apostou " + aposta.getQuantia() + " ESScoins");
}
```

2.4.2 *Message Chains*

É possível verificar a existência de *bad smells* do tipo *Message Chains* nos métodos em cima ilustrados (utilizados para notificar os apostadores), quando é feito `aposta.getEvento().getEquipa_1()` e `aposta.getEvento().getEquipa_2()`.

Para resolver este problema é utilizado *Hide Delegate*, sendo que se adicionam dois métodos à classe `Aposta`, `getEquipa1_Evento()` e `getEquipa2_Evento()` que passam a ser chamados pela classe `Apostador`.

```
// Em Aposta são adicionados os métodos:
public String getEquipa_1Evento(){
    return this.evento.getEquipa_1();
```

```
}

public String getEquipa_2Evento(){
    return this.evento.getEquipa_2();
}

// Em Apostador passa a ser utilizado:
aposta.getEquipa_1Evento();
aposta.getEquipa_2Evento();
```

2.4.3 *Long Parameter List*

O método *newAposta* apresenta 5 parâmetros, sendo assim, enquadra-se num *bad smell* denominado *Long Parameter List*. O método original é o seguinte:

```
    public int newAposta (Integer resultado_evento, Integer resultado_aposta, double quantia,
                          double odd, Evento evento){
        if(saldo >= quantia){
            Aposta aposta = new Aposta(this.idAposta, resultado_evento, resultado_aposta, quantia, odd, evento);
            this.apostas.add(aposta);
            this.idAposta++;
            this.saldo -= quantia;
            return 0;
        }
        return 1;
    }
```

Este problema pode ser resolvido usando a técnica *Introduce Parameter Object*, substituindo os parâmetros do tipo *Integer* e *double* por objetos únicos, da seguinte forma:

```
    public int newAposta (Integer[] resultados, double[] quantiaOdd, Evento evento){
        if(saldo >= quantiaOdd[0]){
            Aposta aposta = new Aposta(this.idAposta, resultados,quantiaOdd, evento);
            this.apostas.add(aposta);
            this.idAposta++;
            this.saldo -= quantiaOdd[0];
            return 0;
        }
        return 1;
    }
```

Para esta resolução ser possível foram também alterados os parâmetros do construtor de *Aposta*, analisado noutro capítulo. O construtor desta classe também apresenta o mesmo *bad smell*, contendo 4 parâmetros:

```
    public Apostador(String email, String password, String nome, double saldo){
        super(email,password,nome);
        this.saldo = saldo;
        this.apostas = new ArrayList<Aposta>();
        this.idAposta = 1;
        this.notificacoes = new ArrayList<String>();
    }
```

Este problema pode ser resolvido usando a mesma técnica *Introduce Parameter Object*, substituindo os parâmetros do tipo *String* por um objeto único, da seguinte forma:

```
    public Apostador(String[] dados, double saldo){
        super(dados[0],dados[1],dados[2]);
        this.saldo = saldo;
        this.apostas = new ArrayList<Aposta>();
        this.idAposta = 1;
        this.notificacoes = new ArrayList<String>();
    }
```

2.5 *BetESSModel*

2.5.1 *Dead code*

Nesta classe representativa da Camada lógica do modelo *MVC* implementado, foi obtida informação através dos recursos oferecidos pelo *IDE NetBeans* e seus *plugins*, que alguns métodos implementados por nós anteriormente não estão mais a ser usados, pelo que devem ser removidos.

Isto pode ter acontecido por a certo ponto do desenvolvimento do programa terem sido criados e usados novos métodos (técnicas) para realizar uma dada tarefa e os métodos antigos não terem sido logo removidos.

Os métodos removidos desta classe são:

- `public void addEvento(Evento evento);`
- `public int mudarOddsEvento(int idEvento, double odd_1, double odd_x, double odd_2);`
- `public int mudarEquipasEvento(int idEvento, String equipa_1, String equipa_2);`

2.5.2 *Large Class*

Sendo esta uma classe com um tamanho considerável, foi optado extrair a parte da mesma responsável por carregar e guardar o estado da camada lógica (persistência) do projeto para uma nova classe `BetESSPersistency` (utilização de *Extract Class*).

Desta forma, e com algumas modificações aos métodos mencionados e à classe `Main` que realiza o carregamento do estado no início da aplicação e guarda o seu estado no fim da mesma, a nova classe `BetESSPersistency` contém o seguinte conteúdo:

```
public class BetESSPersistency {

    public static void guardaEstado(BetESSModel model, String nomeFicheiro){
        try{
            FileOutputStream fos = new FileOutputStream(nomeFicheiro);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(model);
            oos.flush();
            oos.close();
            System.out.println("Dados guardados com sucesso");
        }
        catch (FileNotFoundException ex) {System.out.println("Dados da sessão não guardados!
                                                              Ficheiro não encontrado.");}
        catch (IOException ex) {System.out.println("Dados da sessão não guardados! Erro a aceder a ficheiro!");}
    }

    public static BetESSModel carregaEstado(String nomeFicheiro){
        try{
            FileInputStream fis = new FileInputStream(nomeFicheiro);
            ObjectInputStream ois = new ObjectInputStream(fis);
            BetESSModel m = (BetESSModel) ois.readObject();
            ois.close();
            System.out.println("Dados carregados com sucesso!");
            return m;
        }
        catch (FileNotFoundException ex) {System.out.println("Dados não importados! Ficheiro não encontrado.");}
        catch (IOException ex) {System.out.println("Dados não importados! Erro a aceder a ficheiro.");}
        catch (ClassNotFoundException ex) {System.out.println("Dados não importados! Classe não encontrada.");}
        return null;
    }

}
```

2.5.3 *Long Parameter List*

O método *addEvento* apresenta 7 parâmetros, sendo assim, enquadra-se no *bad smell* denominado *Long Parameter List*.

O método original é o seguinte:

```
public void addEvento(String equipa_1, String equipa_2, double odd_1, double odd_x, double odd_2,
                      boolean disponibilidade){
    Evento novoEvento = new Evento(id_proximoEvento, equipa_1, equipa_2, odd_1, odd_x, odd_2, disponibilidade);
    this.eventos.put(this.id_proximoEvento, novoEvento);
    this.id_proximoEvento++;
}
```

Este problema pode ser resolvido usando a técnica *Introduce Parameter Object*, substituindo os parâmetros das *equipas* e das *odds* por objetos únicos, da seguinte forma:

```
public void addEvento(String[] equipas, double[] odds, boolean disponibilidade){
    Evento novoEvento = new Evento(id_proximoEvento, equipas , odds , disponibilidade);
    this.eventos.put(this.id_proximoEvento, novoEvento);
    this.id_proximoEvento++;
}
```

O método *addApostador* apresenta 4 parâmetros e, sendo assim, também se enquadra neste *bad smell*. Originalmente, o corpo do método é o seguinte:

```
public void addApostador(String email, String password, String nome, double saldo){
    this.utilizadores.put(email, new Apostador(email, password, nome, saldo));
}
```

Este problema foi também resolvido usando a mesma técnica mencionada em cima (*Introduce Parameter Object*), substituindo os parâmetros do tipo *String* por um objeto único, da seguinte forma:

```
public void addApostador(String[] dados, double saldo){
    this.utilizadores.put(dados[0], new Apostador(dados, saldo));
}
```

Para esta resolução ser possível foram também alterados os parâmetros do construtor de *Evento* e de *Apostador*, abordados noutro capítulo.

2.6 *Evento*

2.6.1 *Long Parameter List*

No construtor desta classe podemos reconhecer um *bad smell* denominado *Long Parameter List*, face ao número de parâmetros apresentado pelo método:

```
public Evento(int id, String equipa_1, String equipa_2, double odd_1, double odd_x, double odd_2,
                  boolean disponibilidade){
    this.id = id;
    this.equipa_1 = equipa_1;
    this.equipa_2 = equipa_2;
    this.odds = new double[3];
    this.odds[0] = odd_1; this.odds[1] = odd_x; this.odds[2] = odd_2;
    this.disponibilidade = disponibilidade;
    this.apostadores = new ArrayList<>();
}
```

Este problema pode ser resolvido usando a técnica *Introduce Parameter Object*, substituindo os parâmetros das *equipas* e das *odds* por objetos únicos, da seguinte forma:

```
public Evento(int id, String[] equipas, double[] odds, boolean disponibilidade){
    this.id = id;
    this.equipa_1 = equipas[0];
    this.equipa_2 = equipas[1];
    this.odds = odds;
    this.disponibilidade = disponibilidade;
    this.apostadores = new ArrayList<>();
}
```

2.7 *BetESSView*

2.7.1 *Long Method & Comments*

É facilmente verificável que o método `initView()` presente nesta classe tem um tamanho maior do que aquele que é desejável, sendo assim, é um caso de *Long Method*. Para resolver esta problema, e face também à presença de vários comentários (*Comments*), procedeu-se à extração de métodos (*Extract Method*) para cada menu que é criado nesta função.

Desta forma, e tendo em conta a função inicial:

```
public Menus initView() {
    Menus menus = new Menus();

    //Menu Inicial - número 1
    Opcao op11, op12, op13;
    op11 = new Opcao("Login ..................... ", "L");
    op12 = new Opcao("Efetuar registo ............ ", "R");
    op13 = new Opcao("Sair da Aplicação >>>>>>>>>> ", "S");
    List<Opcao> linhas1 = Arrays.asList(op11, op12, op13);
    Menu menuInicial = new Menu(linhas1, "BetESS - Menu Inicial");
    menus.addMenu(1, menuInicial);

    //Menu Apostador - número 2
    ...
    menus.addMenu(2, menuApostador);

    //Menu Funcionário - número 3
    ...
    menus.addMenu(3, menuFuncionario);

    //Menu Modificar Evento - número 4
    ...
    menus.addMenu(4, menuEditarEvento);

    return menus;
}
```

O resultado obtido com esta técnicas de refactoring é o seguinte:

```
public Menus initView() {
    Menus menus = new Menus();
    menus.addMenu(1, menuInicial());
    menus.addMenu(2, menuApostador());
    menus.addMenu(3, menuFuncionario());
    menus.addMenu(4, menuModificarEvento());
    return menus;
}

private Menu menuInicial(){
    ...
    return menuInicial;
}

private Menu menuApostador(){
    ...
    return menuApostador;
}

private Menu menuFuncionario(){
    ...
    return menuFuncionario;
}
```

```
private Menu menuModificarEvento(){
    ...
    return menuEditarEvento;
}
```

2.7.2 *Duplicate Code*

Nesta classe podemos encontrar dois métodos que têm muito em comum, `menuApostador()` falado no tópico anterior (que devolve o menu normal de um apostador), e `menuApostadorNotificacoes()`, que devolve o menu de um apostador quanto este tem notificações. O código de ambos os métodos pode ser visto em baixo.

```
private Menu menuApostador(){
    Opcao op21, op22, op23, op24, op25, op26;
    op21 = new Opcao("Ver eventos ................ ", "E");
    op22 = new Opcao("Ver minhas apostas ......... ", "V");
    op23 = new Opcao("Realizar nova aposta ....... ", "A");
    op24 = new Opcao("Ver saldo da conta ......... ", "C");
    op25 = new Opcao("Importar quantia ........... ", "I");
    op26 = new Opcao("Menu Inicial >>>>>>>>>>>>>>> ", "S");
    List<Opcao> linhas2 = Arrays.asList(op21, op22, op23, op24, op25, op26);
    Menu menuApostador = new Menu(linhas2, "BetESS - Menu do Apostador");
    return menuApostador;
}

public Menu menuApostadorNotificacoes(int n){
    Opcao op21, op22, op23, op24, op25, op26, op27;
    op21 = new Opcao("VER NOTIFICAÇÕES(" + n + ") ........ ", "N");
    op22 = new Opcao("Ver eventos ................ ", "E");
    op23 = new Opcao("Ver minhas apostas ......... ", "V");
    op24 = new Opcao("Realizar nova aposta ....... ", "A");
    op25 = new Opcao("Ver saldo da conta ......... ", "C");
    op26 = new Opcao("Importar quantia ........... ", "I");
    op27 = new Opcao("Menu Inicial >>>>>>>>>>>>>>> ", "S");
    List<Opcao> linhas = Arrays.asList(op21, op22, op23, op24, op25, op26, op27);
    Menu menuApostador = new Menu(linhas, "BetESS - Menu do Apostador");
    return menuApostador;
}
```

Está assim encontrado um caso de *Duplicate Code*. Para fazer o *refactoring* destes métodos realizou-se uma diminuição de cada um deles e prodeceu-se à realização de *Extract Method* pelo código em comum de ambos, respetivo às linhas em comum que os dois menus partilham. Assim, o resultado obtido é o seguinte:

```
private Menu menuApostador(){
    List<Opcao> linhas = linhasApostadorBase();
    Menu menuApostador = new Menu(linhas, "BetESS - Menu do Apostador");
    return menuApostador;
}

public Menu menuApostadorNotificacoes(int n){
    Opcao op21;
    op21 = new Opcao("VER NOTIFICAÇÕES(" + n + ") ........ ", "N");
    List<Opcao> linhas = new ArrayList<>(linhasApostadorBase());
    linhas.add(0, op21);
    Menu menuApostador = new Menu(linhas, "BetESS - Menu do Apostador");
    return menuApostador;
}

private List<Opcao> linhasApostadorBase(){
    Opcao op21, op22, op23, op24, op25, op26;
```

```
    op21 = new Opcao("Ver eventos ..... ", "E");
    op22 = new Opcao("Ver minhas apostas ..... ", "V");
    op23 = new Opcao("Realizar nova aposta ..... ", "A");
    op24 = new Opcao("Ver saldo da conta ..... ", "C");
    op25 = new Opcao("Importar quantia ..... ", "I");
    op26 = new Opcao("Menu Inicial >>>>>>>>>>>>>> ", "S");
    List<Opcao> linhas = Arrays.asList(op21, op22, op23, op24, op25, op26);
    return linhas;
}
```

3 Impacto do Refactoring

Para averiguar o impacto do *refactoring* no código do projeto desenvolvido, escolhemos algumas métricas de produto de software relacionadas com a estrutura do código, e analisamos as mesmas para o código antes e após o processo de *refactoring* efetuado.

Como forma de facilitar a análise do código desenvolvido, foi por nós utilizado o software *SourceMonitor* na sua versão *V3.5.6.334*. A informação sobre esta aplicação e sobre o seu *download* pode ser encontrada em: `http://www.campwoodsw.com/sourcemonitor.html`.

Desta forma, é criado neste programa um novo projeto, em que são adicionados dois *checkpoint's*: a versão original e a versão pós-*refactoring*. Alguns dos dados oferecidos pela aplicação que ajudam a comparar os dois códigos podem ser encontrados na tabela apresentada em seguida.

	Before *refactoring*	**After *refactoring***
Lines	1005	1128
Statements	828	933
% Comments	1.5 %	0.4 %
% Branches	18.5 %	17.6 %
Calls	378	424
Classes	12	16
Methods/Class	9.92	10
Avg Stmts/Method	5.24	4.20
Max Complexity	13	10
Max Depth	6	5
Avg Depth	2.29	2.11
Avg Complexity	1.82	1.68

De seguida são analisados alguns dos dados aqui apresentados:

- ***Lines***
  O resultado do *SourceMonitor* relativo ao número de linhas de código foi de 1005 linhas no código na versão original face a 1128 linhas na versão pós-*refactoring*. Apesar do número de linhas ter aumentado de uma versão para a outra, o mesmo pode ser considerado normal, tendo em conta o número de novos métodos criados (por exemplo através do processo de *Extract Method* ou de *Replace Temp with Query*) e o número de classes adicionadas face à versão anterior.
- ***Statements & Avg Stmts/Method***
  É possível de verificar que o número de *statements* também aumentou de 828 para 933 de uma primeira versão para a última. Tal como o número de linhas, é algo que podia ser esperado face ao número de métodos e classes que foram adicionados à versão inicial, de forma a tratar os *bad smells* por nós identificados. Ainda assim, face ao *refactoring* efetuado, foi possível diminuir o número de *statements* por método, passando de valores médios de 5.24 para 4.20.
- ***% Comments***
  Derivado da técnica *Extract Method* utilizado na remoção de comentários no código, é possível verificar a redução da percentagem de comentários de 1.5 % para 0.4 %.
- ***% Branches***
  Branches podem ser caraterizados como um conjunto de instruções utilizadas para implementar um controlo de fluxo nos *loops* e condições de um programa de *software*. Sendo assim, verifica-se que a percentagem de *branches* reduz de 18.5 % para 17.6 % na nova versão, o que, apesar de não ser uma descida gigante, acaba por ser sempre positivo e fruto do *refactoring* efetuado. Pode-se também associar uma descida destes valores a uma possível diminuição da complexidade e dificuldade de compreensão do código agora implementado. Grande parte dos controlos de fluxo utilizados no nosso projeto estavam no controlador do modelo *MVC* (`BetESSModel`) que, após *refactoring*, teve os seus métodos "renovados"e também distribuídos por mais duas classes: `Controller_Apostador` e `Controller_Funcionario`.
- ***Calls***
  Verifica-se que o número de chamadas na execução da aplicação aumenta de 378 para 424, algo normal tendo em conta a extração de métodos efetuados e a consequente chamada dos mesmos dentro de outros métodos (possivelmente do mesmo onde o seu código se encontrava).
- ***Classes***
  O número de classes do projeto aumentou de 12 para 16, ou seja, 4 novas classes foram introduzidas de uma versão para a outra. Estas 4 novas classes devem-se às técnicas de *refactoring* que foram efetuadas para remover os *bad*

smells do código original. Foram adicionadas 3 novas classes referentes às modificações efetuadas ao controlador (classes `Controller_Apostador` e `Controller_Funcionario`, e interface `UserControllerInterface`), e uma nova classe `BetESSPersistency` relativa à extração de código relacionado com a persistência dos dados que se encontrava em `BetESSModel`.

- ***Methods/Class*** A quantidade média de métodos por classe manteve-se estável, passou de 9.92 para 10, tendo em conta que foram adicionados novos métodos, mas ainda assim foram removidos alguns através da detecção de *Dead code* e foram implementadas novas classes.
- ***Max Complexity & Avg Complexity*** Estes dados de *output* devolvidos pelo *SourceMonitor* são alguns cálculos que, de certa forma, analisam a complexidade de ambas soluções. A complexidade segundo esta aplicação mede o número de caminhos de execução possíveis a partir de uma função ou método. Desta forma, é possível perceber que a complexidade máxima da aplicação baixa de 13 para 10, e a complexidade média é reduzida também de 1.82 para 1.68.
- ***Max Depth & Avg Depth*** Aapesar de não sabermos as fórmulas que levam o *SourceMonitor* a devolver os valores para `Max Depth` e `Avg Depth`, conseguimos depreender que a "profundidade"máxima que o código tem na versão inicial era de 6 e toma agora o valor de 5, e que a "profundidade"média passa de 2.29 para 2.11

Tendo em conta estes dados obtidos e as análises agora realizadas, podemos partir para analisar 3 tipos de métricas de produtos de *software* que consideramos relevantes para o estudo em questão: **Linhas de código**, **Manutenibilidade** e **Complexidade**.

3.1 Linhas de código

Linhas de código é uma métrica de *software* utilizada para medir o tamanho de um programa contando o número de linhas de texto que o código fonte do programa contém. O número de linhas é normalmente usado para prever a quantidade de esforço que é necessário atribuir a um programa de *software* para este ser desenvolvido, assim como estimar a produtividade ou manutenibilidade do produto assim que este esteja concluído.

Comparando os valores obtidos do número de linhas de código para a versão original da aplicação em causa com a versão pós-*refactoring* percebemos que o valor subiu de 1005 linhas para 1128, como relatado em cima.

Ainda assim, não podemos encarar esta mudança como algo negativo, ainda mais sabendo nós o motivo pelo qual isto acontece, face ao grande número de métodos que foram extraídos (seja por realizar *Extract Method* ou *Replace Temp with Query* por exemplo) e às classes que foram também adicionadas. Estas técnicas levaram a reduzir o número de linhas por método (como comprovado pelo *output* do *SourceMonitor*), mas a aumentar o número de linhas geral do programa.

Sendo assim, podemos ter ficado com um código maior, mas possivelmente mais fácil de compreender e de manter, tendo em conta todo o *refactoring* efetuado no mesmo.

3.2 Manutenibilidade

Manutenibilidade pode ser caraterizada como a facilidade com que um produto de *software* pode ser mantido para, por exemplo, corrigir certos defeitos ou identificá-los, implementar novos requisitos, ou maximizar a eficiência, confiabilidade e segurança do mesmo.

Na aplicação em causa neste estudo, pensamos ter obtido uma nova solução com maior poder de manutenibilidade que a primeira versão.

Por exemplo, apesar do número de classes ter aumentado, o código da aplicação ficou mais "separado"e mais fácil de editar, sendo que desta forma o código ficou mais organizado. Para além disto, o número médio de *statements* por método diminuiram (como relatado anteriormente), o que deixa a entender que o código ficou mais percetível e mais fácil de realizar alterações no mesmo. A percentagem de comentários no código fonte também desceu, o que indicia que o código e o próprio nome dos métodos tornou o código mais percetível e fácil de entender.

Podemos assim concluir que o código ficou mais extensível e legível, derivado da simplificação dos métodos e extração de classes efetuadas com o *refactoring*.

3.3 Complexidade

A complexidade é uma métrica que analisa a lógica do código fonte de um programa.

Analisamos a complexidade do código definida por *Steve McConnell* no seu livro "*Code Complete,1993*", sendo esta a definição utilizada no programa *SourceMonitor*. A complexidade máxima e média baixam na versão pós-*refactoring*, como podemos relatar pelos dados obtidos e analisados anteriormente.

Se verificarmos a evolução de `Max Complexity` e `Avg Complexity` através dos dados obtidos pelo *SourceMonitor*, podemos verificar que todas as classes ou mantiveram ou diminuiram o primeiro e o segundo valor. As classes cujos valores foram alterados têm os seus dados apresentados na seguinte tabela:

	Before *refactoring*		**After *refactoring***	
	Max Complexity	*Avg Complexity*	*Max Complexity*	*Avg Complexity*
BetESS_Controller	13	4.11	6	2.56
__Controller_Apostador	-	-	10	2.86
__Controller_Funcionario	-	-	8	2.88
__UserControllerInterface	-	-	0	0.00
BetESSModel	5	1.43	5	1.38
__BetESSPersistency	-	-	4	3.50
BetESSView	3	1.33	3	1.22
Main	7	7.00	2	1.20
Apostador	5	1.70	5	1.50
Aposta	6	1.56	6	1.45

Esta redução de valores deve-se à utilização de técnicas como *Extract Class*, *Extract Method* ou *Replace Temp with Query*.

Pela tabela agora apresentada, podemos concluir que as principais classes que levaram as estas alterações foram a classe `BetESS_Controller` e a classe `Main`.

- ***BetESS_Controller***
  Podemos observar que a complexidade máxima do total da aplicação era 13, devido à complexidade máxima desta mesma classe. Após *refactoring*, verificamos que a classe passa a ter uma complexidade máxima de valor 6 e uma complexidade média que baixa para 2.56 a partir de 4.11, tendo em conta as classes e os métodos extraídos. Podemos notar que o valor da complexidade máxima da classe `Controller_Apostador` implementada após *refactoring* toma o valor de 10, que passa a ser o valor máximo para toda a aplicação.
- ***Main***
  Esta classe apresentava uma complexidade máxima de valor 7 e uma complexidade média numa ordem de 7.00. Após o *refactoring*, com os métodos que foram extraídos da sua função `main` e também algumas partes do código relativas à persistência da aplicação que também passaram a ser incluídas em `BetESSPersistency`, os valores da complexidade máxima e média passam a ser de 2 e 1.20, respetivamente.

Para além disto, as medidas do *SourceMonitor* tal como *Max Depth* e *Avg Depth* ajudam também a concluir que a complexidade da nova versão é inferior à versão original, ajudando a compreender que o código agora modificado tem uma "profundidade"inferior à original.

4 Conclusão

Podemos concluir, derivado das métricas definidas e analisadas, que o *refactoring* efetuado ao código da primeira versão teve uma grande influência na qualidade do código do programa.

Apesar do tamanho do código aumentar (como podemos deduzir tendo em conta o aumentar do número de linhas e do número de classes), o código fonte ficou mais fácil de compreender e diminuiu a sua complexidade.

Desta forma, pode-se concluir que a partir da identificação de *bad smells* no código e da realização da sua refatorização através das respetivas técnicas de *refactoring* estudadas e aprendidas, podemos desenvolver código "melhor", com maior manutenibilidade, menor complexidade, e mais fácil de perceber.