

UNIVERSIDADE DO MINHO

MIEI - 4º ANO - ESS

ARQUITETURAS DE SOFTWARE

BetESS - Refactor



Manuel Sousa - A78869



Tiago Alves - A78218



Conteúdo

1	Introdução	2
2	Refactoring BetESS	2
2.1	Code Smells	2
2.2	Técnicas de Refactoring	3
2.3	Estrutura do Código Fonte	3
2.4	PMD Source Code Analyzer	4
2.4.1	Regras Aplicadas	4
2.5	AutoRefactor	9
3	Apreciação Crítica e Conclusão	12

1 Introdução

Depois de desenvolvida a plataforma de apostas, tendo em conta os vários requisitos que foram pedidos, é necessário que essa mesma solução seja mantível ao longo do tempo. Para isto, é bastante útil que o código fonte da mesma se torne o mais legível e reutilizável possível, para que desta forma permita a qualquer tipo de desenvolvedor provocar alterações (quando necessárias) da forma mais simples possível, o que é alcançável através da técnica de **Refactoring**.

2 Refactoring BetESS

Refactoring é o processo de modificar um sistema/produto de software com o intuito de melhorar a estrutura interna do código, mas sem alterar o seu comportamento externo. O uso desta técnica evita que o código fonte tenha um ciclo de vida curto. Em grosso modo, *Refactoring* faz com que o software seja mais fácil de entender.

2.1 Code Smells

A aplicação da técnica de *Refactoring* tem como intuito principal a identificação de aquilo que são chamados *Code Smells*. Estes, são qualquer tipo de característica no código fonte de um determinado programa que possivelmente indicarão um problema maior. Desta forma, são identificados como “*smells*”, e necessitam de ser removidos.

- **Bloaters** - corresponde a código, métodos ou classes que cresceram de tal maneira que se tornou difícil manter e trabalhar com os mesmos.
- **Object-Orientation Abusers** - correspondem a uma incompleta ou incorreta aplicação dos princípios da programação orientada a objetos.
- **Change Preventers** - quando se altera algo numa parte do código, e isso terá efeitos colaterais.
- **Dispensables** - algo que é dispensável e que a sua abstenção só trará uma melhor leitura do código.
- **Couplers** - quando diversas classes tão excessivamente interligadas.

2.2 Técnicas de Refactoring

Com intuito de remover por completo os “*smells*” existentes no nosso código fonte, existem diversas técnicas que “atacam” o problema diretamente, e indicam-nos algumas maneiras de o resolver.

- **Extract Method** - mover um determinado pedaço de código para um novo método, e substituir por uma chamada a esse mesmo método.
- **Inline Method** - substituir chamadas a um método pelo conteúdo do mesmo.
- **Extract Variable** - criar variáveis cujo conteúdo seja o resultado de uma determinada expressão, tornando assim código mais legível.
- **Inline Temp** - substituir uma referência a uma variável pela sua atribuição.
- **Replace Temp with Query** - mover uma expressão para um novo método e retornar o resultado deste. É preferível elaborar uma query, em vez de usar uma variável temporária.
- **Split Temporary Variable** - dividir uma variável temporária atribuída a várias expressões num determinado método. Devem ser usadas variáveis diferentes para diferentes valores.
- **Remove Assignments to Parameters** - usar variáveis temporárias em vez de atribuir valores a variáveis passadas como parâmetros.
- **Replace Method with Method Object** - transformar um método com alguma computação numa classe para que as variáveis temporárias sejam atributos de um objeto.

Para isto, resolvemos usar algumas ferramentas que nos auxiliasssem na identificação de “*smells*” no nosso código, explicadas mais abaixo.

2.3 Estrutura do Código Fonte

Finalizada a criação da aplicação, a estrutura produzida baseou-se na divisão de código em 3 *packages*. Um *package* tratava de guardar e carregar o estado da aplicação, outro manipulava todos os objetos relativos à interface da mesma, e por fim a terceira, tratava de manipular todos os dados e fazer os cálculos necessários.

Baseados nesta estrutura, achamos mais indicado resolvemos aplicar *refactoring* a todo o código pertencente ao *package* dos dados e também ao *package* da manipulação de dados. Resolvemos não incluir o *package* da interface, visto que maior parte do código incluído é gerado de forma automática, e por sua vez não alterado de forma tão flexível.

2.4 PMD Source Code Analyzer

Uma das ferramentas usadas foi o *PMD*, um analisador de código fonte. Esta ferramenta, é capaz de encontrar diversos defeitos como variáveis não utilizadas, criação desnecessária de objetos, e outros. Tem suporte para várias linguagens, incluindo *Java*, a linguagem em que a nossa plataforma foi desenvolvida.

Para o funcionamento do mesmo, é passado, por linha de comandos, diversas instruções que indicam em que código fonte irá ser aplicado, bem como uma lista de regras que serão herdadas pelo *PMD*. Estas regras, escritas tipicamente num ficheiro *XML*, estão relacionadas com boas práticas em POO bem como um *design* correto.

Posto isto, iremos, no subcapítulo seguinte, explicar algumas das regras e de como estas foram aplicadas ao nosso código fonte.

2.4.1 Regras Aplicadas

É de notar que todas estas regras estão incluídas na documentação da ferramenta, ou seja, qual fosse a regra pretendida, teríamos apenas de navegar pela documentação, encontrar, e depois adicioná-la ao ficheiro *XML* o qual compilaria todas as regras que iriam ser aplicadas ao código. Algumas destas regras são:

- *ExcessiveMethodLength* - quando um método é excessivamente longo, poderá indicar que está a fazer mais do que lhe compete.
- *Potential Violation of Demeter Law* - The Law of Demeter diz que um objeto deve ter pouco conhecimento sobre outros objetos, e manter-se para si mesmo. Evita também correntes de chamadas de métodos.
- *GodClass* - quando uma classe tem demasiado conteúdo. Quando são demasiado grandes, extramamente complexas, e contêm diversos atributos.
- *SimplifyBooleanReturns* - evitar blocos condicionais quando o valor de retorno é um *boolean*. O seu valor lógico pode ser retornado.

Isto são apenas algumas amostras de uma enorme biblioteca fornecida pelo *PMD*. Posto isto, iremos então mostrar alguns exemplos dos “smells” encontrados no nosso código pelo PMD, e mostrar o resultado da resolução do mesmo. Importante denotar que “Method Chain Calls” foi o “smell” mais encontrado pelo *PMD*.

PMD report			
Problems found			
#	File	Line	Problem
1	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/Aposta.java	77	Potential violation of Law of Demeter (method chain calls)
2	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/Aposta.java	93	Potential violation of Law of Demeter (method chain calls)
3	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/Aposta.java	126	Potential violation of Law of Demeter (method chain calls)
4	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/Aposta.java	126	Potential violation of Law of Demeter (method chain calls)
5	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/Apostador.java	74	Potential violation of Law of Demeter (method chain calls)
6	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/Apostador.java	148	Potential violation of Law of Demeter (method chain calls)
7	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/Apostador.java	148	Potential violation of Law of Demeter (method chain calls)
8	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/Apostador.java	151	Potential violation of Law of Demeter (method chain calls)
9	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/BetESS.java	23	This class has too many methods, consider refactoring it.
10	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/BetESS.java	66	Potential violation of Law of Demeter (method chain calls)
11	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/BetESS.java	102	Potential violation of Law of Demeter (method chain calls)
12	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/Fase-1/Primeira-Versão/BetESS/BetESS/src/betess/business/BetESS.java	168	Avoid unused local variables such as 'a'.

Relatório PMD antes de aplicado *Refactoring*

PMD report			
Problems found			
#	File	Line	Problem
1	/Users/gcsousa/Desktop/Universidade/4º Ano/1º Semestre/Arquiteturas de Software - ESS/Trabalho Prático/BetESS-AS2018/Fase-2/BetESS-refactored/BetESS/src/betess/business/BetESS.java	23	This class has too many methods, consider refactoring it.

Relatório PMD depois de aplicado *Refactoring*

Analisado o relatório, passamos então para a resolução destes “smells”. Aqui, estão alguns exemplos mais comuns, e de como estes foram resolvidos.

- **Avoid unused local variables such as 'a' - Linha 9**

```

1 public int login( String email , String pass )
2     throws EmailErradoException , PassErradaException
3 {
4     int estatuto ;
5
6     if ( email . equals ( " betadmin@betess . pt " ) &&
7         pass . equals ( " betadmin " ) )
8     {
9         Admin a = new Admin () ;

```

```

10         this.setUser(email);
11         estatuto = 0;
12     } else {
13     ...
14 }
15
16     return estatuto;
17 }
```

1 - Before Refactoring

```

1 public int login(String email, String pass)
2             throws EmailErradoException, PassErradaException
3 {
4     int estatuto;
5
6     if (email.equals("betadmin@betess.pt") &&
7         pass.equals("betadmin"))
8     {
9         this.setUser(email);
10        estatuto = 0;
11    } else {
12    ...
13 }
14
15     return estatuto;
16 }
```

2 - After Refactoring

- Potential Violation of Demeter Law (chain calls) - Linhas 6 e 8

```

1 public void verificaVitoria(Aposta a, String user) {
2     boolean vitoria = true;
3     double oddApostada;
4     String resultado = "";
5
6     for (Evento e: a.getEventos().values()) {
7         // Guarda a odd apostada neste evento.
8         oddApostada = a.getOdds().get(e.getIdEvento());
9
10        // Procura a equipa na qual apostou baseado na odd.
11        if (e.getOddUm() == oddApostada)
12            resultado = e.getEquipaUm();
13        if (e.getOddDois() == oddApostada)
14            resultado = e.getEquipaDois();
15        if (e.getOddX() == oddApostada)
```

```

16         resultado = "EMPATE";
17
18     // Verifica se acertou no resultado.
19     if (!resultado.equals(e.getResultado()))
20         vitoria = false;
21     }
22
23     if (vitoria) this.apostadores.get(user)
24         .addTotalCoins(a.getGanhoTotal());
25 }
```

1 - Before Refactoring

```

1 public void verificaVitoria(Aposta a, String user) {
2     boolean vitoria = true;
3     double oddApostada;
4     String resultado = "";
5
6     Collection<Evento> col = a.getCollectionEventos();
7
8     for (Evento e: col) {
9         // Guarda a odd apostada neste evento.
10        int idEvento = e.getIdEvento();
11        oddApostada = a.getOddApostada(idEvento);
12
13        // Procura a equipa na qual apostou baseado na odd.
14        if (e.getOddUm() == oddApostada)
15            resultado = e.getEquipaUm();
16        if (e.getOddDois() == oddApostada)
17            resultado = e.getEquipaDois();
18        if (e.getOddX() == oddApostada)
19            resultado = "EMPATE";
20
21        // Verifica se acertou no resultado.
22        if (!resultado.equals(e.getResultado()))
23            vitoria = false;
24    }
25
26    if (vitoria)
27        this.apostadores.get(user)
28            .addTotalCoins(a.getGanhoTotal());
29 }
```

2 - After Refactoring

- Potential violation of Law of Demeter (object not created locally) - Linha 3

```

1 public BetESS editaMailUser(String email) {
2     Apostador a = this.apostadores.get(user);
3     a.setEmail(email);
4
5     this.apostadores.remove(user);
6     this.apostadores.put(email, a);
7     this.user = email;
8
9     return this;
10}

```

1 - Before Refactoring

```

1 public BetESS editaMailUser(String email) {
2     this.apostadores.get(user).setEmail(email);
3
4     this.apostadores.put(email, this.apostadores.get(user));
5     this.apostadores.remove(user);
6
7     this.user = email;
8
9     return this;
10}

```

2 - After Refactoring

Como estes, diversos foram detetados e posteriormente resolvidos. Um dos que foi detetado, foi o correspondente à regra denominada de *TooManyMethods*. Este, como o próprio nome indica, diz que determinada classe tem demasiados métodos e que, por isso, devia sofrer *refactoring* - “**This class has too many methods, consider refactoring it.**”. Este “smell” dizia respeito, então, à nossa classe principal, BetESS.

Da forma como toda a aplicação foi estruturada, baseando-nos numa classe principal que tratava de comunicar com os diversos objetos, era normal que esta tivesse mais métodos em comparação com as outras classes. Visto isto, tentamos resolver esse entrave através da divisão dessa mesma classe, mas essa solução não foi obtida com sucesso, pois na parte das interfaces, eram chamados métodos que estariam em várias partes da divisão, então não seria de todo viável haver várias instâncias de diferentes BetESS, ao invés de ter só uma. Desta forma, concluímos que a melhor solução seria, neste caso, manter a classe como está, visto que as operações que esta fazia eram apenas relacionadas com estruturas de dados e comunicação com outras classes, não existindo cálculos na mesma. Posto isto, achamos que o grau de complexidade desta classe não seria, de todo, muito elevada.

2.5 AutoRefactor

Esta ferramenta consiste num *plugin* para o *IDE* denominado de *Eclipse*, e o que faz é basicamente aplicar diversas regras de refactoring ao nosso código de forma automática. Como é dito no *website* da ferramenta, esta, para além de encontrar *bad smells*, também ajuda a eliminar diversos *warnings* que foram escapados por ferramentas como o *PMD*. O nosso intuito ao usar uma segunda ferramenta era o de obter uma maior confiança possível e também fazer uso de ferramentas de *refactoring* automático.

Depois de eliminados os *bad smells* apontados pelo *PMD*, resolvemos então executar o *AutoRefactor* sob o código previamente alterado. Neste poderíamos selecionar quais seriam os “smells” a serem identificados, e como seriam resolvidos. Algumas das alterações feitas por este são:

- ArrayList rather than LinkedList;
- ArrayList rather than Vector;
- AutoBoxing rather than explicit method;
- Boolean equals() rather than null check;
- Collections APIs rather than Vector pre-Collections APIs;
- Do/while rather than duplicate code;
- Do/while rather than while;
- Double primitive rather than wrapper;
- Empty test rather than size;
- End of method rather than return;
- Map.entrySet() rather than Map.keySet() and value search;
- EnumMap rather than HashMap for enum keys;
- EnumSet rather than HashSet for enum types;
- Equals on constant rather than on variable;
- Extract common code in if else statement;
- Generic list rather than raw list;

- Generic map rather than raw map;
- HashMap rather than Hashtable;
- HashMap rather than TreeMap;
- HashSet rather than TreeSet.

Visto isto, foi então aplicado o refactor ao código já alterado, e como era um pouco espectável, não ocorreram mudanças críticas. Para além de mudanças estéticas (adição de chavetas em certos *if-statements*), a única mudança mais visível foi a seguinte:

- **Map.entrySet() rather then Map.keySet() and value search**

```

1 public void alteraEstadoEvento(int idEvento) {
2     if (this.eventos.containsKey(idEvento)) {
3         this.eventos.get(idEvento).setEstado("FECHADO");
4         boolean terminada = true;
5
6         for (String apostador: this.apostas.keySet()) {
7             LinkedList<Aposta> lista =
8                 this.apostas.get(apostador);
9
10            for (Aposta a: lista) {
11                ...
12            }
13
14            this.apostadores.get(apostador)
15                .setHaNotificacoes(true);
16        }
17    }
18
19    this.eventos.remove(idEvento);
20}
```

1 - Before AutoRefactor

```

1 public void alteraEstadoEvento(int idEvento) {
2     if (this.eventos.containsKey(idEvento)) {
3         this.eventos.get(idEvento).setEstado("FECHADO");
4         boolean terminada = true;
5
6         for (Map.Entry<String, LinkedList<Aposta>> entry:
7             this.apostas.entrySet())
8         {
9             String apostador = entry.getKey();
```

```
10         LinkedList<Aposta> lista = entry.getValue();
11         for (Aposta a: lista) {
12             ...
13         }
14         this.apostadores.get(apostador)
15             .setHaNotificacoes(true);
16     }
17 }
18
19 this.eventos.remove(idEvento);
20
21 }
```

1 - After AutoRefactor

3 Apreciação Crítica e Conclusão

A realização deste tipo de trabalho prático permite-nos adquirir maiores competências ao nível da programação e estruturação de código. Os processos de *Refactoring* podem, por vezes, possuir algum grau de complexidade, pelo que é de extrema importância uma revisão a cada iteração do projeto, também para que se evitem reconstruções em fases muito adiantadas.

O desenho da solução implementada para a plataforma de apostas desenvolvida anteriormente não consistia no uso de padrões de software. Estes podem ir “contra” os conceitos de *refactor*, dadas as suas características, como por exemplo, o padrão *Facade*, por possuir uma grande quantidade de tarefas, é assumido como um *code smell*. Neste caso, existe um objeto com todas as estruturas e métodos centralizados, não sendo *Facade*, designado **BetESS**. Para evitar este *code smell*, houve uma tentativa de divisão desta classe em subclasses. No entanto, não se revelou possível tal implementação, pois apresentava uma elevada quantidade de mudanças em todas as camadas lógicas do projeto, que teriam levado a uma completa nova elaboração do mesmo.

Em suma, e dada esta dificuldade em reestruturar a camada lógica de negócio, é logicamente visível que todas as etapas de um projeto devem ser bem idealizadas, desenhadas e revistas por toda a equipa que o constitui, pois uma simples revisão pode levar uma melhor *performance* de todo o fluxo de execução.