Programação Orientada aos Objetos
Grupo 43

Alexandre de Freitas Ferreira Pacheco A80760
Diogo José Cruz Sobral A82523
José Pedro Milhazes Carvalho Pinto A80741

27 de Maio de 2018

A80760

A82523

A80741

Resumo

Neste semestre, no âmbito da unidade curricular de Programação Orientada aos Objetos (**POO**), foi-nos proposta a execução de um projeto prático, a fim de fazer uso da aprendizagem feita sobre este paradigma de programação e sobre a linguagem de programação *Java*.

O projeto consistiu na implementação de uma plataforma de gestão de faturas e dados relativos às atividades económicas de contribuintes individuais e empresas. Para este fim, o programa resultante deveria apresentar funcionalidades como, por exemplo, o registo de novos contribuintes, empresas e faturas, e também operações mais complexas como por exemplo a consulta das faturas correspondentes a um determinado contribuinte ou empresa, possivelmente num determinado período.

1 Arquitetura de Classes

De modo a implementar esta plataforma é necessário, naturalmente, organizar dados numa estrutura que, embora preferencialmente compacta e simples, seja de rápido e fácil acesso.

As classes elementares da nossa estrutura representam contribuintes, setores de atividade fiscal, e faturas. A classe de contribuintes é abstrata, sendo extendida pelas classes referentes às empresas e aos contribuintes individuais.

Cada uma destas classes corresponde ainda, a outra classe que define a base de dados referentes a esses objetos (no caso dos contribuintes, empresas e contribuintes individuais organizam-se na mesma base de dados).

As bases de dados referentes a cada uma das classes básicas são agrupadas numa base de dados geral.

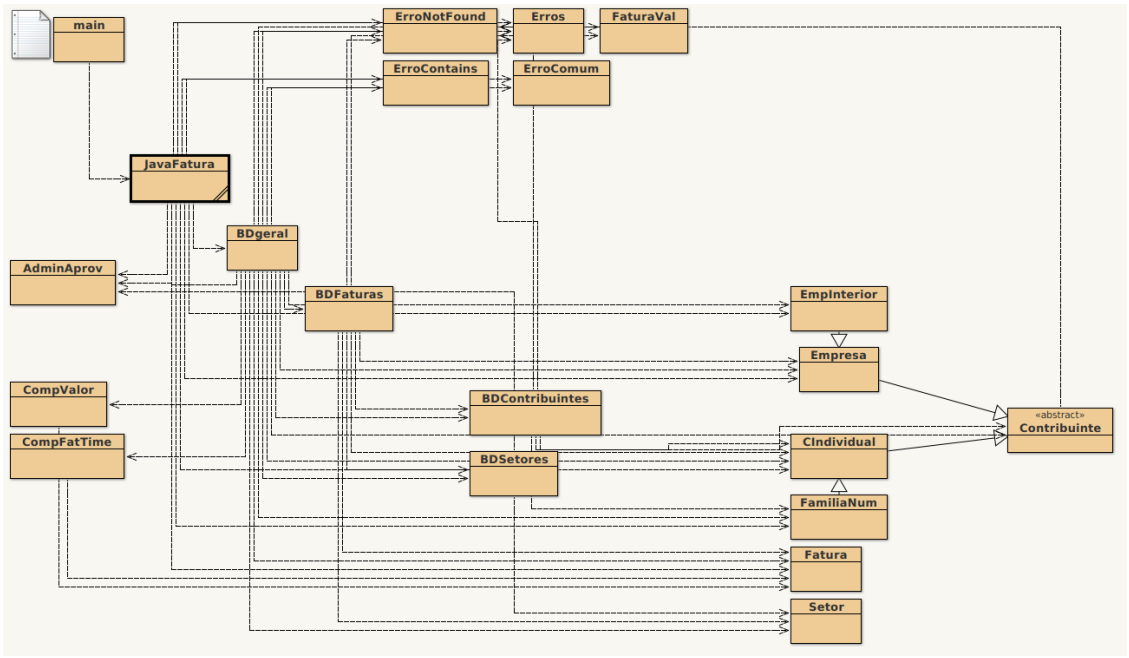


Diagrama de classes

1.1 Organização dos Objetos

Os objetos das clases fundamentais foram organizados nas suas respetivas bases de dados em *HashMaps* e *HashSets*.

```java
public class BDFaturas implements Serializable
{
    private Map<Integer,Fatura> faturas;
    private Set<Integer> faturas_porval;

    /**
     * Construtor sem argumentos.
     */
    public BDFaturas (){
        this.faturas = new HashMap<Integer,Fatura>();
        this.faturas_porval = new HashSet<Integer>();
    }
```

Base de dados de faturas

```java
public class BDSetores implements Serializable {
    private Map<String, Setor> setores;
    private Set<String> por_aprovar;

    /**
     * Construtor sem argumentos.
     */
    public BDSetores() {
        this.setores = new HashMap<String, Setor>();
        this.por_aprovar = new HashSet<>();
    }
```

Base de dados de setores de atividade fiscal

```java
public class BDContribuintes implements Serializable
{
    private Map<Integer,Contribuinte> dados;


    /**
     * Contrutor sem argumentos da classe BDContribuintes.
     */
    public BDContribuintes (){
        this.dados = new HashMap<Integer,Contribuinte>();
    }
```

Base de dados de contribuintes

Decidimos estruturar os dados desta forma devido à significativa vantagem que os *HashMaps* e *HashSets* apresentam em relação aos *TreeMaps* e *TreeSets* em operações de adição, remoção e procura.

Os setores de atividade fiscal podiam ter sido implementados na forma de, por exemplo, um par de uma *String* e um *double*, no entanto, nós optámos por lhes dedicar uma classe, por uma questão de abstração. Assim, no caso de ser necessária a implementação de funcionalidades adicionais envolvendo os setores, não será necessária uma grande alteração ao seu modo de funcionamento.

1.2 Base de dados geral

A nossa base de dados geral, para além de, como já referido, reunir os conjuntos os tipos de dados básicos, inclui também um *HashMap* com os concelhos fiscalmente bonificados (pensando na implementação dos requisitos adicionais).

Aproveitámos o facto de as classes `Empresa` e `CIndividual` serem ambas subclasses de `Contribuinte`, e assim reunimos os dois tipos de objetos em bases de dados com a mesma definição (`BDContribuintes`.

```java
public class BDgeral implements Serializable
{


    private BDContribuintes empresas;
    private BDContribuintes individuais;
    private BDFaturas faturas;
    private BDSetores setores;
    private Map<String,Double> Concelhos = new HashMap<>();
```

Base de dados geral

2 Principais Funcionalidades

De entre os requisitos apresentados nos enunciados, há alguns cuja implementação explora, de uma forma geral as qualidades da estrutura de dados por nós montada e demonstra o modo de funcionamento habitual deste tipo de operações, pelo que passamos a descrever alguns:

2.1 Dedução de Despesa

O cálculo do montante deduzido deuma despesa é calculado mediante a seguinte forma:

1. Cada fatura está associada a um setor de atividade fiscal (especificado quando a mesma é emitida, e aprovado pelo administrador quando é introduzido pela primeira vez na base de dados). A cada um destes setores corresponde uma taxa (percentual) de dedução.
2. A empresa prestadora do serviço também influencia este cálculo:
   (a) Através de um coeficiente calculado conforme o seu rendimento anual.
   (b) Caso a empresa seja do interior, é ainda bonificada com um uma percentagem adicional de dedução fiscal, conforme o seu concelho.
3. O consumidor do produto ou serviço também influencia este cálculo, através:
   (a) do seu rendimento.
   (b) do número de elementos do seu agregado familiar.

Assim, a fórmula de cálculo do montante deduzido a cada fatura é a seguinte:

$$valor = FATURA * EMPRESA * CLIENTE$$

$$FATURA = ValorDaFatura * SetorDeAtividade$$

$$EMPRESA = 1 + CR\_empresa + Coef.Concelho + 0.005 * SA\_empresa$$

$$CLIENTE = 1 + CR\_cliente + 0.0025 * SA\_cliente + 0.01 * NElementosAgregado + COND$$

$$COND = \begin{cases} 0.05*\text{NElementosAgregado} & \text{se NElementosAgregado} \geq 4 \\ 0 & \text{se NElementosAgregado} < 4 \end{cases}$$

Todos os elementos necessários ao cálculo desta fórmula estão acessíveis a partir da base de dados geral. Esta operação é simples de implementar (em *Java*), no entanto, este exemplo serve para demonstrar a diversidade de dados armazenados na estrutura criada, e a facilidade com que os podemos consultar.

2.2 Relação dos 10 contribuintes que mais gastam

Esta é um dos requisitos mais exigentes de implementar, pelo simples facto de ser necessário consultar todos os contribuintes e todas as faturas (a não ser que o cálculo do valor total gasto por cada contribuinte estivesse precalculado, pore exemplo).

Optámos por não incluir na nossa estrutura dados auxiliares desse tipo, dado que, seguindo essa lógica, acabaríamos por tornar a estrutura extremamente redundante, ao tomar essa abordagem para todas as operações requisitadas.

Então, a abordagem que tomámos para responder a este reuisito foi:

1. Criar um *HashMap* que faz corresponder NIFs ao montante total gasto pelo respetivo contribuinte.
2. Percorrer todos os contribuintes e inserir os seus NIFs no *HashMap*.
3. Percorrer todas as faturas e somar os valores das mesmas ao valor do *HashMap* correspondente ao contribuinte para quem a fatura foi emitida.
4. Criar um *TreeMap* auxiliar com um `Comparator` que ordena decrescentemente as entradas segundo os seus valores, e nele inserem-se os valores do *HashMap*.
5. Criar uma *ArrayList* onde se inserem as chaves das primeiras 10 entradas do *TreeMap*, obtendo-se e retornando-se, nessa lista, o resultado pretendido, os NIFs dos 10 users que mais gastaram.

Embora pudessemos saltar o segundo passo deste procedimento (dado que as faturas também têm guardado o NIF do contribuinte para quem foram emitidas, decidimos não o fazer devido à possibilidade de haver menos de 10 contribuintes a pedir faturas. Neste caso, nós preenchemos o resto da lista-resultado com NIFs de contribuintes que não tenham gastos.

2.3 Top N Empresas que mais Faturam

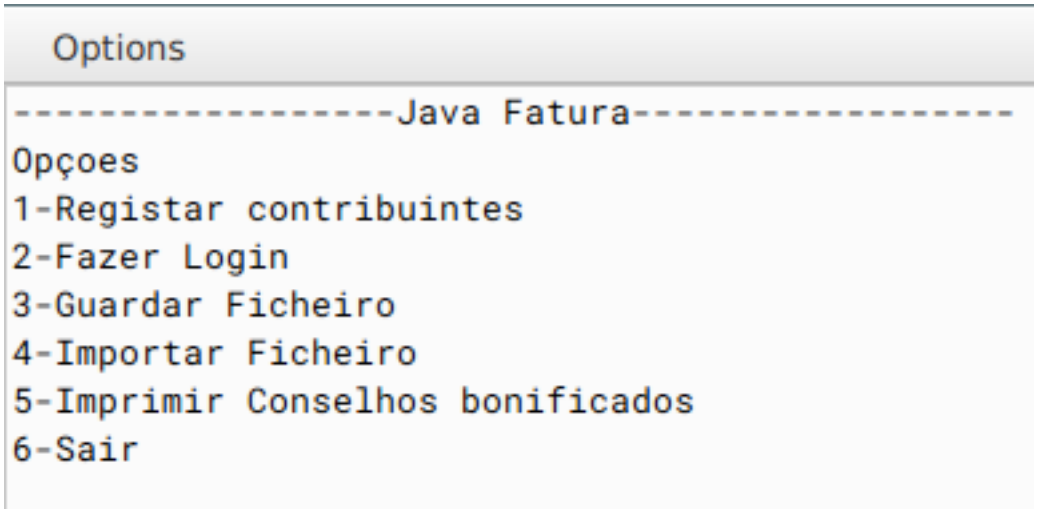
Este é outro requisito que, tal como o descrito acima, exige alguns cálculos, no entanto, faclmente executado recorrendo à API do *Java*.

1. São criados dois *HashMaps*, para armazenar o montante total faturado e o montante total deduzido de cada empresa.
2. É percorrida a base de dados das empresas, e para cada empresa é percorrida a sua lista de faturas emitidas. Para cada uma dessas faturas, é calculada a dedução do valor, sendo estes dois valores acumulados numa variável local. Ao fim de processar as faturas de cada empresa, os valores resultantes são armazenados nos respetivos *HashMaps* auxiliares.
3. É criado um *TreeMap* com um `Comparator` para organizar decrescentemente pelo valor entradas que fazem corresponder o NIF de cada empresa ao montante total faturado.
4. É criado ainda um *HashMap* onde se guardam os N primeiros elementos do *TreeMap* preenchido no passo anterior, respondendo então ao requisito, devolvendo as N empresas que mais faturaram, e o respetivo montante total resultante de deduções.

3 Interação com o Utilizador

O modo de interação com o programa que adotámos foi uma simples **CLI** (*Command Line Interface*). Funciona da seguinte forma: quando o programa inicia, aparece um menu principal, e conforme as ações tomadas, podem ser abertos sub-menus dedicados a um conjunto de operações, como por exemplo o menu de admin, que permite utilizar funcionalidades reservadas aos administradores da plataforma, ou o menu das empresas, que permite emitir faturas, por exemplo.

Tomámos a decisão de não armazenar na base de dados a palavra passe do administrador, estando esta *hard-coded* no programa, por motivos de segurança.



Menu inicial

4 Implementação de Novas Funcionalidades

O paradigma de programação que temos seguido, bem como a linguagem que utilizámos neste projeto, facilitam a produção de software de maneira que seja fácil a implementação de novas ferramentas e funcionalidades. Um bom exemplo é a segunda remessa de requisitos do enunciado deste projeto (envolvendo os "concelhos do interior"e as famílias numerosas).

Bastou criar `EmpInterior` e `FamiliaNum`, subclasses de `Empresa` e `CIndividual`, respetivamente. Bastou criar em ambas um método referente à redução de imposto em vigor, e na das empresas implementar um campo referente ao concelho a que a empresa pertence.

Fosse necessário, seria possível implementar ainda mais e mais complexos modelos de cálculo da dedução das aturas, ou criação de situações ou casos particulares dos já definidos, graças à reutilização de código possibilitada pela hierarquia de classes e herança de métodos.

5 Conclusão

Atingidos os objetivos deste projeto, não pudemos deixar de reparar na facilidade de implementação deste tipo de plataformas obedecendo ao paradigma de programação orientada aos objetos e utilizando linguagens como o *Java* , principalmente em termos de abstração, reutilização de código e modularidade, em oposição a linguagens como C, por exemplo, em que teríamos de "reinventar a roda"para obter um produto final com as mesmas funcionalidades.

A possiblididade de criar classes abstratas e interfaces permite que o programa seja escrito de forma a ser extremamente facilitada a alteração e, melhor ainda, a adição de funcionalidades, devido à flexibilidade que existe entre a herança ou alteração e criação de novos métodos e atributos em subclasses.

Outro ponto forte deste modelo de trabalho é a segurança, visto que com o mínimo de cuidados na escrita do código e estruturação do programa, a estrutura de dados mantida pelo programa fica invulnerável a eventuais alterações por parte de terceiros.

Num projeto futuro, poderemos melhorar ainda, fazendo melhor uso da hierarquia de classes e das interfaces, como por exemplo, criar uma interface `BaseDeDados` que seja implementada pelas bases de dados que nós utilizámos, e certamente têm muitas características em comum.