Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Sistema de Gestão de Veículos - F2

47206 : Tiago Alexandre Figueiredo Pardal (47206@alunos.isel.pt)

47202 : Manuel Maria Penha Gonçalves Cavaco Henriques (47202@alunos.isel.pt)

47198 : Ricardo Filipe Matos Rocha (47198@alunos.isel.pt)

Relatório para a Unidade Curricular de Sistemas de Informação
da Licenciatura em Engenharia Informática e de Computadores

Professor : Doutor Nuno Miguel Soares Datia

09 – 05 – 2022

Resumo

No âmbito da segunda fase do trabalho prático da cadeira, este relatório tem como propósito explicitar a de desenho da aplicação de acesso à base de dados desenvolvida na primeira fase do mesmo.

Este relatório parte do pressuposto do acesso por parte do leitor ao código desenvolvido no âmbito do mesmo, não sendo assim necessário enunciá-lo em extensão, bastando apenas mencionar trechos do mesmo.

Abstract

In the scope of this assignment's second phase, this report has the purpose of exposing the design philosophy of the application, specifically the one developed in the first phase of this assignment.

This report parts with the assumption that the reader has access to the code developed, not being necessary to describe it to its full extent, only mentioning a few parts of it.

Índice

Lista de Figuras

1

Introdução

1.1 Camada Acesso a Dados

A aplicação foi desenvolvida de forma a tentar modular ao máximo o acesso aos dados e as escritas sobre a base de dados.

A aplicação desenvolvida baseia-se numa já existente da autoria do professor, fornecida como exemplo.

Sendo carregado os dados do modelo da base e dados pelo *Entity Manager*. Esta class pertencente à biblioteca do **JPA**, sendo inicializada através do *Entity Manager Factory*, vai conter os métodos necessários para iniciar as transações à base de dados e permitir interagir com a mesma.

As escritas são apenas efetuadas após a obtenção dos valores a inserir, atualizar ou até apagar, e os termos verificados quanto ao seu tipo e formato de modo a garantir que não se tentam efetuar escritas com valores errados, evitando assim acessos à base de dados desnecessários.

1.2 Jakarta Persistence

Toda a aplicação foi desenvolvida com base na *Java Persistence API*.

O **JPA** é responsável pelo acesso a dados, o mapeamento dos mesmos através do *Object-Relational Mapping* (**ORM**) e as escritas dos mesmos na base de dados.

1.2.1 ORM Mapeamento Entidades

A camada **ORM** é estabelecida através do *EclipseLink*, tendo sido criadas interfaces e classes para cada entidade da base de dados, com uso às anotações respetivas, responsáveis por dar a informação relevante ao *EclipseLink*, de forma a que este mapeie cada tabela e as suas respetivas colunas para um objecto.

2

Problema

2.1 Caso em Estudo

Recordamos aqui o caso em estudo.

No final deste capitulo encontram-se as adendas efetuadas a esta base de dados.

No presente trabalho é nos proposto o desenvolvimento de um sistema de gestão e registo de localização de automóveis e camiões.

Estando presentes neste as informações acerca dos clientes, dos veículos e das zonas a que os veículos se devem cingir, caso contrário deve ser gerado um alarme. Além dos próprios alarmes em si quando em situação de serem ativados.

2.2 Modelo EA

Na presente implementação foi considerado como vantajoso a separação entre Veículo e Condutor, de forma a facilitar a alteração do mesmo.

Para a realização desta separação foi necessário a atribuição de um **ID** para a tabela Condutor, ao que achamos que faria sentido não ser um **ID** genérico, mas sim o seu número de cartão de cidadão, **CC**, sendo que entendemos por **CC** como o número de identificação civil e não o número de cartão de cidadão, visto interessar-nos apenas um identificador do cidadão e não do cartão de cidadão do mesmo.

O estado do equipamento tem como valores: {'Activo','PausaDeAlarmes','Inactivo'}.

As coordenadas **GPS** latitude e longitude têm graus decimais e foram assim definidas com o tipo *numeric(3,1)*, permitindo assim uma casa decimal.

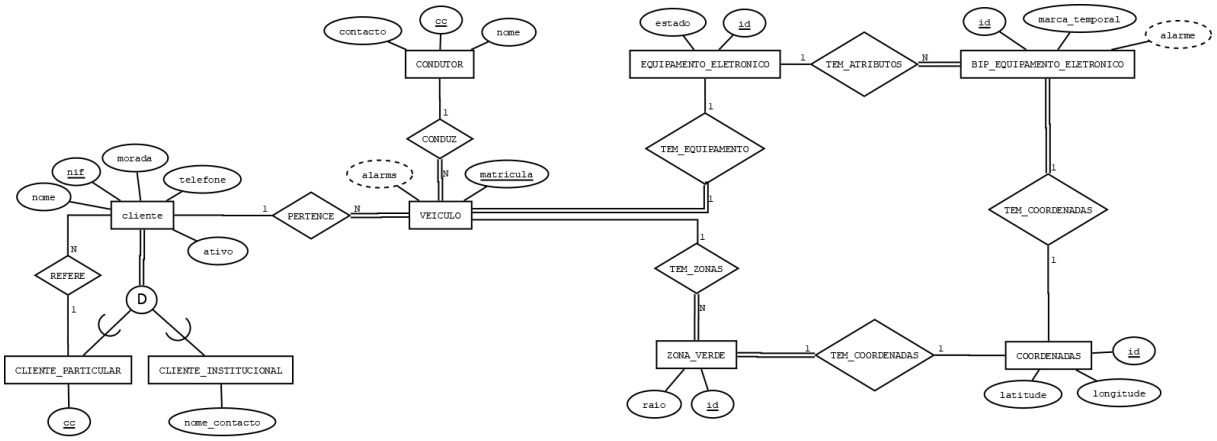

Figura 2.1: Diagrama EA

O atributo alarme é um booleano que quando ativo representa a existência de alarme naquele Bip.

Achámos a implementação de uma restrição em matricula poderia ser demasiado limitadora, assim, optou-se por não implementar nenhuma.

Existem 2 relações não presentes no modelo EA, sendo estas *Requests* e *Invalid_Requests*.

2.3 Adendas sobre Base de dados implementada na primeira fase

Ao acedermos à nossa base de dados através da aplicação, no decorrer do desenvolvimento desta fase, deparámo-nos com algumas incoerências e erros, que corrigimos.

Sendo assim, as alterações efetuadas nesta fase encontram-se referidas nesta secção.

Quanto ao modelo EA desenvolvido na primeira fase deste trabalho, atualizou-se a relação entre Zona Verde e Coordenadas, passando a relação de 1:1 para N:1, apercebendo-nos de que não existia razão para restringir as Coordenadas em relação às Zonas Verdes, possibilitando que várias Zonas Verdes tenham a mesma Coordenada.

Para além desta mudança no modelo EA, algumas mudanças foram feitas aos procedimentos e outras funções já realizadas em **SQL** e **PLPGSQL**. Quanto à criação de tabelas, mudou-se a restrição quanto aos números telefonicos, presentes nas Relações

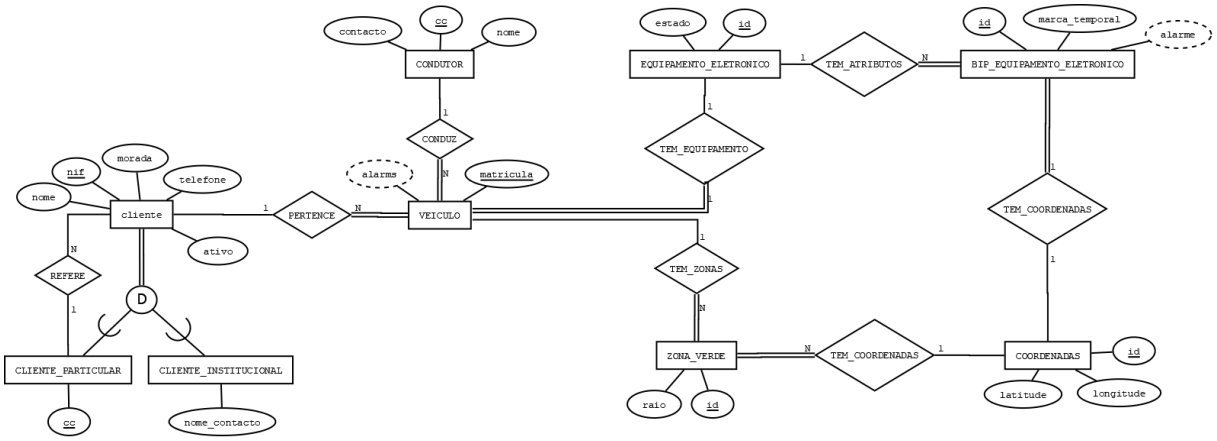

Figura 2.2: Diagrama EA atualizado

Condutor e Cliente, já que os números telefónicos podem incluir números fora do país de origem.

Quanto aos procedimentos e funções, foram atualizadas algumas verificações e foram revistos algos erros avistados durante a realização da segunda fase, nomeadamente na atualização da informação associada a um Cliente Particular, sendo necessário verificar se o Cliente é Particular. Também na função que processa os registos efetuados não eram apagados os registos que eram verificados com sucesso, permitindo que estes fossem adicionados várias vezes pelas várias chamadas que poderiam ser feitas à função.

Ainda foi possível encontrar um outro erro que possibilitava um Cliente Particular ter três ou mais Veiculos, tendo sido resolvido o problema da função que era chamada pelo *trigger*.

Outras pequenas mudanças foram feitas, sendo menores e não tendo relevância suficiente para listar cada uma nesta secção.

3

Organização modelo JPA

3.1 Organização geral

Na nossa implementação fizemos uso de *Repositories* e *Mappers*, usando interfaces para as ligar com o *JPAcontext*, que contém o *Entity Manager*, gerindo as entidades do **JPA** e as transições requisitadas.

Para o acesso às entidades, utiliza-se uma variável de ambiente.

3.2 *Repositories*

Os *Repositories*, através das interfaces para cada entidade, extendem a interface *IRepository* e implementam as seguintes funções:

- T findByKey(TK key)
- TCol find(String jpql, Object... params)
- List<T> findAll()

Para além das funções já presentes na interface *IRepository*, foram implementadas outras sobre *Repositories* especificos, com a função de implementar *Named Queries*, que

terão o objetivo de obter os objetos **JPA** das entidades a partir de parâmetros das mesmas, de forma a resolver a dificuldade com que nos deparámos a tentar aceder a entidades do **JPA** cuja a chave é gerada pelo sistema.

Quando o **ID** é gerado pelo sistema, à partida não é fácil o acesso ao objeto apartir do mesmo, visto em diversos casos sabermos alguns ou até todos os parametros do mesmo, com exceção do próprio **ID**, devido ao carater altamente imprevisivel do mesmo, principalmente devido a situações de concorrencia em que os dados não persistiram, mas o valor foi incrementado.

Começámos por usar o *findByKey* para encontrar uma entidade no **JPA** a partir da chave da mesma, mas gradualmente alterámos a mesma para a função *read* implementada através dos *Mappers*, devido à menor assinatura da chamada do método e à sua melhor apropriação a lidar com objectos singulares.

3.3 *Mappers*

Os *Mappers*, à semelhança dos *Repositories*, utilizam interfaces para cada entidade, que extendem a interface *IDataMapper*, implementando as seguintes funções:

- TK create(T entity)
- T read(TK id)
- TK update(T entity)
- TK delete(T entity)

Os *Mappers* limitam-se a implementar as funções **CRUD**, explicando a simplicidade do seu retorno e parâmetros.

4

Detalhes de Implementação

4.1 Notas Prévias

É de denotar apenas que para as implementações que retornam informação para a aplicação, com exceção às vistas, foram implementados testes, que verificam o correto comportamento das mesmas.

Alguns testes foram desenvolvidos para poderem ser corridos independentemente da informação presente na base de dados.

No entanto a grande maioria, por falta de tempo e devido à complexidade necessária para serem implementados do modo previamente necessário, correm apenas com base nos valores iniciais, sendo que na eventualidade de escritas ou de deletes da base de dados, que alterem o estado inicial da base de dados podem levar a erros nestes.

Assim sendo para correr os testes usa-se apenas o estado inicial da base de dados, para isso temos um script único que remove o modelo da base de dados, cria as tabelas, insere dados nas mesmas e cria todos os procedimentos e *triggers* necessários.

No entanto todos os testes têm o cuidado de manter o estado da base de dados antes de serem iniciados, tendo sido implementado uma função de *rollback* para isso.

Também é importante mencionar que para a implementação do processamento de registos, o retorno da função a indicar o seu sucesso ou a sua falha é um booleano gerado pelo método *executeUpdate* da *Query*, em que caso a *Query* se realize com sucesso

retorna verdadeiro, retornnando falso em caso de insucesso, não verificando se verdadeiramente o procedimento chamado através do "*CALL*" realiza qualquer alteração na base de dados.

Ainda mais, não se implementou a periodicidade neste procedimento nem no que apaga os registos inválidos, sendo que a ideia de implementação passaria pelo uso de uma *class* chamada *Timer* presente no **java.util**, em que após a sua instanciação, utilizaria-se o método *scheduleAtFixedRate*, em que se criaria um objecto do tipo *TimerTask*, implementando a função *run* que esta *class* abstrata contém, em que se iria chamar o procedimento pretendido. O tempo para executar o task seria sempre o mesmo, sendo passado como último argumento. O *delay* passado seria de 0, já que se pretende executar o procedimento no período definido, sem qualquer atraso.

Para uso em testes foi efetuada uma reimplementação da função equals de modo a comparar os atributos dos objectos em vez do hash code do objecto em si.

4.2 Atualização da Informação de Cliente Particular

O *create*, *update* e *delete* foram implementados manualmente, sem uso a procedures, embora estes tenham sido desenvolvidos na primeira fase do trabalho.

Assim, fazemos uso dos *Mappers* para efetuar estas operações nos Clientes Particulares.

4.3 Total Alarmes

A funcionalidade para obter o total de alarmes foi implementada através da chamada da função já realizada na primeira fase.

A chamada da função foi efetuada através da chamada de uma outra função *alarm_number*. Esta chamada faz-se através de um *Named Stored Procedure* e do respetivo mapeamento do mesmo na classe **Bip**.

Os parâmetros foram passados por ordem, sendo obtido o retorno a partir do terceiro parâmetro, visto este ter sido estipulado no terceiro parâmetro.

4.4 Criar Veiculo com Zona Verde sem procedimento

A funcionalidade de criar um veiculo e uma zona verde associada ao mesmo, caso os parâmetros para isso forem passados, encontra-se implementado do seguinte modo:

Em primeiro lugar aquando da obtenção dos valores por parte do utilizador, questionamos o utilizador se pretende acrescentar uma Zona Verde.

Se assim pretender, pedimos-lhe os parâmetros para a mesma. A partir daqui é chamada uma função que verifica os parametros obtidos, nomeadamente se os mesmos são válidos.

Se os parametros forem todos válidos, incluindo os da Zona Verde, efetuamos a criação de um Veiculo e de uma Zona Verde, caso os parâmetros para Zona Verde não sejam válidos, ou não tenham sido passados corretamente inserimos apenas Veiculo.

Na presente implementação não pretendemos criar um Condutor, Cliente ou Equipamento se os mesmos não existirem, assim limitamo-nos a cancelar a criação de Veiculo se os mesmos não existirem, efetuando esta verificação através de *reads*, sem tentarmos efetuar escritas inválidas na base de dados.

Neste seguimento, deparámo-nos com um problema derivado da nossa implementação de Coordenadas, que têm um **ID** gerado, assim para verificar-mos se o **ID** já existe na base de dados, desenvolvemos uma *Named Query* que nos retorna um objeto Coordenadas com base em latitude e longitude.

Assim, ao verificar-mos se já existe aquela coordenada na base de dados, evitamos a criação de coordenadas extra.

Devido à *Named Query* ter sido implementada da forma como foi, não retorna **null** caso não encontre nada para os parametros passados. Para resolver isto, fizemos uso de um *try catch* que verifica se foram encontradas Coordenadas para os parâmetros passados.

Se exitirem, prossegue para a criação de uma Zona Verde com as Coordenadas já existentes.

Se não existir, vai ser lançada uma exceção *NoResultException*, que apanhamos com o *catch*, e corremos o código para a criação de uma Coordenada nova, criando em seguida a Zona Verde com a nova Coordenada.

Se pretendêssemos também poderíamos ter desenvolvido uma implementação que cria se condutor, equipamento e cliente, questionando o utilizador acerca dos dados dos mesmos caso o utilizador assim deseja se.

4.5 Implementações com Procedimentos

As restantes implementações foram efetuadas com recurso a *Native Queries*, sendo passados os parâmetros por ordem, à semelhança de como foram passados para as *Named*

Stored Procedures, tal como mencionado na secção anterior, não sendo desta vez recebido qualquer parâmetro como retorno.

As *Native Queries* não requerem qualquer mapeamento, em oposição às *Named queries*, visto que nas *Named Queries* criávamos o *Query* de chamada com um determinado nome e procedíamos a chamá-lo, aqui nas *Named Queries* efetuamos logo a *Query*.

Em procedimentos que possam ser passados parâmetros a **null**, usamos a técnica de overload, de forma a evitar enviar parâmetros a **null**. Poderia ter sido efetuado o mesmo apenas com uma função, mas preferimos a implementação com 2 funções com o mesmo nome, mas com parâmetros diferentes.