Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Trabalho Prático Fase 2

47283 : Ricardo Duarte Cardoso Bernardino (a47283@alunos.isel.pt)

47249 : Miguel Henriques Couto de Almeida (a47249@alunos.isel.pt)

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

Professor : Doutor Nuno Miguel da Costa de Sousa Leite

10 – 6 – 2023

Resumo

No âmbito da segunda fase do trabalho prático da cadeira, este relatório tem como propósito a elaboração de uma camada de acesso à base de dados para o problema do enunciado da primeira fase, que use uma implementação JPA recorrendo aos padrões de desenho DataMapper, Repository e UnitOfWork. É também discutido o adequado uso e acesso aos dados bem como a devida utilização do processamento transacional recorrendo ao JPA para esse efeito. É também demonstrado como corretamente usar e libertar ligações e recursos conforme a sua necessidade, como também é demonstrada a implementação das restrições de integridade ou lógica de negócio a nível do JPA.

Abstract

As part of the second phase of the course's practical work, this report aims to create a database access layer for the first phase's statement problem, which uses a JPA implementation using the DataMapper, Repository and UnitOfWork. The appropriate use and access to data is also discussed, as well as the proper use of transactional processing using JPA for this purpose. It also demonstrates how to correctly use and release bindings and resources as needed, as well as the implementation of integrity constraints or business logic at the JPA level.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Lista de Abreviaturas e Siglas

EA Modelo Entidade-Associação. viii, 9

SQL Structured Query Language. 10

1

Introdução

1.1 Camada de Acesso a Dados

Na segunda fase deste trabalho, foi desenvolvida a camada de acesso a dados, que tem como objetivo principal facilitar o acesso e manipulação dos dados armazenados no sistema. Esta camada também trata do controlo transacional, garantindo que as operações são realizadas de forma consistente e segura, usando adequadamente mecanismos disponíveis no JPA, como transações e isolamento transacional, permite controlar a concorrência e garantir a integridade dos dados quando acedidos. Para esse intuito, foi utilizada uma implementação de JPA juntamente com um subconjunto dos padrões de desenho DataMapper, Repository e UnitOfWork. [3]

1.2 Jakarta Persistence

O uso da JPA simplifica consideravelmente o desenvolvimento da camada de acesso a dados, fornecendo uma interface de alto nível para a interação com a base de dados relacional. Através de anotações e configurações adequadas, é possível mapear as entidades do sistema para tabelas da base de dados, estabelecendo a correspondência entre o modelo de objetos da aplicação e o modelo relacional.

1.2.1 Padrões de desenho

Os padrões de desenho fornecem estruturas e abstrações que ajudam a organizar e simplificar o código realizado. Nesta seccção discutiremos os padrões de desenho DataMapper, Repository e UnitOfWork utilizados na camada de acesso a dados.

1.2.1.1 UnitOfWork

O *DataScope* é uma classe abstrata projetada para gerir a correta libertação de recursos do *EntityManager* e do *EntityManagerFactory* de forma eficiente e segura.

A principal finalidade do *DataScope* é garantir que cada operação de persistência ou consulta de dados seja realizada dentro de um escopo transacional controlado. Com a utilização do bloco *try-with-resources* juntamente com o DataScope, é possível garantir que os recursos do *EntityManager* sejam corretamente libertados, independentemente do resultado da operação.

A estrutura do *DataScope* é baseada no padrão de desenho *"Unit of Work"* (Unidade de Trabalho). Este cria uma sessão de trabalho isolada para cada operação, permitindo agrupar várias operações relacionadas numa única transação. Esta possibilidade ajuda a manter a consistência dos dados e evita problemas de concorrência. O uso do *DataScope* oferece várias vantagens:

- Gestão transacional: O *DataScope* inicia automaticamente uma transação aquando da sua criação e confirma-a ao finalizar o escopo. Consecutivamente esta garante que todas as operações dentro do escopo sejam tratadas como uma única transação, permitindo o controlo adequado de *commits* e *rollbacks*.
- Controlo de recursos: O *DataScope* garante que os recursos do *EntityManager* e do *EntityManagerFactory* sejam corretamente libertados, mesmo em caso de exceções. Isto evita a fuga de recursos e ajuda a manter a integridade e a eficiência do sistema.
- Isolamento de operações: Cada *DataScope* cria um novo *EntityManager* e uma nova sessão de trabalho isolada. Isto significa que as operações realizadas num *DataScope* não interferem com as operações de outros *DataScopes*, garantindo um ambiente transacional confiável e consistente.
- Controlo de exceções: O bloco *try-with-resources* do *DataScope* permite a captura e o tratamento de exceções de forma adequada. Em caso de exceção, o escopo é finalizado corretamente, garantindo que as transações sejam revertidas (*Rollback*) e os recursos sejam libertados.

1.2.1.2 DataMapper

O padrão DataMapper é responsável por mapear objetos de domínio para objetos de acesso a dados e vice-versa. Ajuda a separar as preocupações entre a camada de domínio e a camada de acesso a dados, evitando que a lógica de persistência fique espalhada por toda a aplicação.

Operações CRUD : Nos mappers da camada de acesso a dados, são implementadas as operações CRUD (Create, Read, Update e Delete) que permitem a manipulação dos dados da base de dados. Cada operação possui uma respectiva função:

- Create (Criar):

  A operação *Create* é responsável por inserir novos registos da entidade em questão na base de dados. O método *Create* cria uma instância do DataScope usando um bloco *try-with-resources*. Dentro deste bloco, o *EntityManager* é obtido a partir do *DataScope* e a transação é iniciada automaticamente. A operação de criação é realizada, ao persistir o objeto no na base de dados com a função *persist* do *EntityManager*. Em seguida, o DataScope valida o trabalho realizado e, ao finalizar o bloco *try*, o escopo é fechado automaticamente, liberando os recursos.

- Read (Ler):

  A operação de leitura é utilizada para recuperar registos específicos da base de dados. O método *Read* segue uma estrutura semelhante, onde o *DataScope* é criado usando um bloco *try-with-resources*. Dentro deste bloco, o *EntityManager* é obtido a partir do DataScope e a transação é iniciada. A operação de leitura é realizada, ao buscar uma entidade na base de dados usandoa função *find* do *EntityManager*. Após a conclusão desta operação, o DataScope valida o trabalho realizado e, ao finalizar o bloco *try*, o escopo é fechado automaticamente.

- Update (Atualizar):

  A operação de atualização é responsável por modificar os dados de um registo existente na base de dados. O método *Update* também utiliza o bloco *try-with-resources* com o *DataScope*. Dentro deste bloco, o *EntityManager* é obtido e a transação é iniciada. A operação de atualização é realizada, como ao buscar a entidade especifica na base de dados usando a função find do *EntityManager* e modificar as suas propriedades. Ao finalizar o bloco *try*, o *DataScope* valida o trabalho realizado e o escopo é fechado automaticamente.

- Delete (Excluir):

  A operação de exclusão é utilizada para remover um registos na base de dados. O método *Delete* segue a mesma estrutura dos demais métodos. O *DataScope* é criado no bloco try-with-resources, o *EntityManager* é obtido e a transação é iniciada. A operação de exclusão é realizada, ao buscar a entidade a ser removida na base de dados usando a função find do *EntityManager*, procedendo à remoção da mesma usando o método *remove* do *EntityManager* sob esse objeto. Ao finalizar o bloco *try*, o *DataScope* valida o trabalho realizado e o escopo é fechado automaticamente.

1.2.1.3 Repository

O padrão de desenho *Repository* define interfaces e métodos para a realização de consultas e operações específicas na base de dados. Esta fornece uma interface com métodos para realizar operações de CRUD (*Create*, *Read*, *Update*, *Delete*) em entidades específicas, abstraindo os detalhes da implementação de acesso a dados:

- Método getAll:

  Implementa a interface *getAll*() definida na interface *IRepository*. Utiliza o bloco *try-with-resources* com o *DataScope* para garantir o controlo transacional previamente mencionado. Dentro do bloco, o *EntityManager* é obtido e a consulta é realizada utilizando uma *NamedQuery*, definida para cada entidade na sua classe modelo, para obter a lista de todos registos da entidade em questão, ou seja todas as entradas da sua respetiva tabela na base de dados. Em seguida, o *DataScope* valida o trabalho realizado e a lista é retornada. Em caso de exceção, a mensagem de erro é exibida.

- Método find:

  Implementa a interface *find*() definida na interface *IRepository*, e utiliza o *Mapper* da entidade respetiva para realizar a leitura de uma entidade com base na sua chave primária fornecida como parâmetro, chamando o método *read* deste Mapper.

- Método add:

  Implementa a interface *add*() definida na interface *IRepository*, e utiliza o *Mapper* da entidade respetiva para criar um novo registo na base de dados, chamando o método*create* deste Mapper.

- Método delete:

  Implementa a interface *delete* definida na interface *IRepository*, e utiliza o *Mapper* da entidade respetiva para excluir um registo na base de dados, chamando o método *delete* deste Mapper.

- Método save:

  Implementa a interface *save* definida na interface *IRepository*, e utiliza o *Mapper* da entidade respetiva para atualizar um registo na base de dados, chamando o método *update* deste Mapper.

1.2.2 Mapeamento Entidades

Além dos mapeamentos de cada classe, foi criada uma named query em cada uma desta, para obter todos os elementos da tabela.

1.2.2.1 Região

Para esta entidade mapeamos [2] uma classe com um atributo que chamámos nome, também ele chave primária e estabelecemos uma relação de um para muitos com a entidade Jogador, tal como tinha sido implementado na fase anterior.

1.2.2.2 Jogador

Para esta entidade mapeamos uma classe com id, username, email, estado, regiao, como atributos, tendo o id o papel de chave primária e estabelecemos uma relação de muitos para muitos com a entidade Jogador, para representar a relação de Amizade entre os jogadores, uma outra relação de volta com a classe Região, desta vez para representar a região a que pertence o Jogador, tal como tinha sido implementado na fase anterior como ainda foi estabelecida uma relação com Conversa de muitos para muitos para estabelecer a relação dos Jogadores com as diferentes Conversas a que podem pertencer.

1.2.2.3 Amizade

Para esta entidade mapeamos uma classe com dois atributos, ambos chaves, ambos id's de jogadores tendo sido portanto necessário a criação complementar de uma classe AmizadeKey para simbolizar esta chave composta.

1.2.2.4 Conversa

Para esta entidade mapeamos uma classe com dois atributos, o id da conversa e o nome da conversa onde o id da conversa ficou como chave. No entanto para estabelecer a relação dos jogadores presentes numa dada conversa, foi criada uma relação de muitos para muitos com o Jogador.

1.2.2.5 Conversa Membros

Para esta entidade mapeamos uma classe com dois atributos, ambos chaves, o id da conversa e o id do jogador, tendo sido portanto necessário a criação complementar de uma classe ConversaMembrosKey para representar esta chave composta.

1.2.2.6 Mensagem

Para esta entidade mapeamos uma classe com seis atributos,o numero da mensagem, o id da conversa, a data da mensagem, hora da mensagem, o seu texto e o id do jogador que a enviou. Foi colocado o número da mensagem como id e foram estabelecidas as relações com as outras entidades, primeiramente com a conversa, uma relação de muitos para um representada pelo o id da conversa, e uma relação muitos para um com o Jogador representada pelo id do jogador que enviou a mensagem.

1.2.2.7 Jogo

Para esta entidade mapeamos uma classe com três atributos, o id do jogo, chave, o nome do jogo e o url para uma imagem do jogo.

1.2.2.8 Compra

Para esta entidade mapeamos uma classe com quatro atributos, o id do jogo, id do jogador, sendo estes dois chaves, a data de aquisição do jogo e o preço do jogo. Visto que temos uma chave composta foi necessário mapeá-la numa outra classe que chamámos de CompraKey.

1.2.2.9 Cracha

Para esta entidade mapeamos uma classe com quatro atributos, o nome do cracha, chave, o jogo do cracha, o limite de pontos para conseguir o crachá e a imagem para

o crachá. Foi também mapeada a relação de muitos para um com o jogo, quando mapeamos esta chave estrangeira.

1.2.2.10 CrachaJogador

Para esta entidade mapeamos uma classe com dois atributos, o nome do cracha, o id do jogador, sendo ambos chave desta entidade, foi portanto necessário criar a classe CrachaJogadorKey para representar esta chave composta.

1.2.2.11 Partida

Para esta entidade mapeamos uma classe com sete atributos, o número da partida, a referência ao jogo, a região, data de começo e fim e o tipo de partida. Foi necessário criar uma clase para representar a chave composta entre número da partida e referência do jogo. Foram representadas as relações com outras entidades começando com a relação entre partida e jogo, muitas para um, com as pontuações multijogador e normais, uma para muitos em ambas, e as relações de especialização da entidade entre partida multijogador e normal, relações estas que foram também mapeadas de um para muitas.

1.2.2.12 Partida MultiJogador

Para esta entidade mapeamos uma classe com dois atributos, o id, chave da partida e o estado. Foi estabelecida a relação de muitas para uma de volta com a entidade Partida.

1.2.2.13 Partida Normal

Para esta entidade mapeamos uma classe com dois atributos, o id, chave da partida e o grau de dificuldade. Foi estabelecida a relação de muitas para uma de volta com a entidade Partida.

1.2.2.14 Pontuações Partida Normal

Para esta entidade mapeamos uma classe com quatro atributos, sendo três destes membros da chave composta que identifica as pontuações, constituída pelo número da partida, a referência do jogo e o id do jogador restando o atributo pontos desta tabela. Foi mapeada recorrendo ao id da partida junto com a referência para o jogo a relação de muitos para uma com a classe Partida Normal e mapeada também a relação de muitas para uma com a classe do Jogador.

1.2.2.15 Pontuações Partida MultiJogador

Para esta entidade mapeamos uma classe com quatro atributos, sendo três destes membros da chave composta que identifica as pontuações, constituída pelo número da partida, a referência do jogo e o id do jogador restando o atributo pontos desta tabela. Foi mapeada recorrendo ao id da partida junto com a referência para o jogo a relação de muitos para uma com a classe Partida Normal e mapeada também a relação de muitas para uma com a classe do Jogador.

1.2.2.16 Estatística Jogo

Para esta entidade mapeamos uma classe com quatro atributos, a chave referência do jogo, o número de partidas, número de jogadores e o total de pontos. Foram mapeadas as relações de muitos para um com o Jogo através da chave.

1.2.2.17 Estatística Jogador

Para esta entidade mapeamos uma classe com quatro atributos, a chave id do jogador, o número de jogos, número de partidas e o total de pontos. Foram mapeadas as relações de um para um com o Jogador através da chave.

2

Problema

2.1 Caso em Estudo

O caso de estudo deste trabalho permenece igual ao da primeira fase do mesmo, envolve o desenvolvimento de um sistema de gestão de jogos, jogadores e partidas para a empresa fictícia "GameOn". O objetivo é criar um sistema que registe dados de jogadores, jogos e partidas, além de fornecer funcionalidades adicionais como compra de jogos, e atribuição de crachás a jogadores.

2.2 Modelo EA

O modelo Entidade e Associação EA permaneceu igual à primeira fase, sendo que nenhuma alteração foi realizada a niveis de cardinalidade, novas entidades, novos atributos etc.

Estas são as principais tabelas e relações presentes no modelo EA, conforme definido na primeira fase deste trabalho.

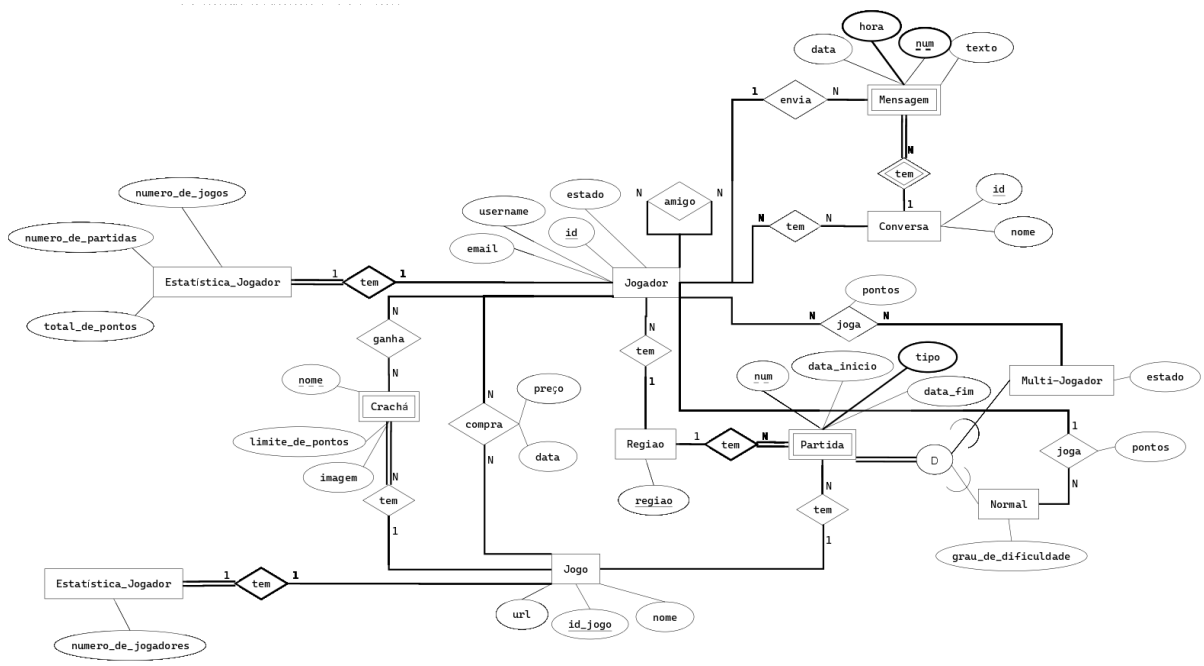

Figura 2.1: Modelo EA

2.3 Adendas sobre Base de dados implementada na primeira fase

Foram realizadas algumas adendas sob nomeadamente os scripts SQL definidos na primeira fase do trabalho, mais especificamente nos scripts das funções, e da inserção dos valores iniciais. (*functions.sql* e *insert.sql* respetivamente).

As adendas realizadas são as seguintes:

- insert.sql - O ID do utilizador administrador inserido foi alterado para o número 9999, pois o *JPA* não permite o número 0 como chave primária da tabela.

- functions.sql - A função *associarCracha* foi re-organizada em várias funções para a sua estrutura estar melhor definida, para uma leitura mais adequada da mesma, e mais importantemente para a realização da alínea 1 c) da segunda fase do trabalho. As novas funções, que apenas organizam o código já previamente realizado dentro do procedimento armazenado, são:

    1. **Função 1: `doesGameExist`**

Verifica se uma partida existe. Recebe o número da partida como parâmetro e retorna um valor booleano indicando se a partida existe na tabela "Partida". A função realiza uma consulta na tabela "Partida"procurando uma correspondência com o número da partida fornecido. Se encontrar pelo menos um registo, retorna verdadeiro, caso contrário, retorna falso.

2. **Função 2: `getGameDetails`**

   Obtém os detalhes de um jogo, incluindo a referência do jogo e o tipo de partida. Recebe o número da partida como parâmetro e retorna uma tabela contendo os campos "ref"(referência do jogo) e "tipo_partida"(tipo de partida). A função realiza uma consulta na tabela "Partida"procurando o número da partida fornecido e retorna os valores correspondentes aos campos especificados.

3. **Função 3: `doesPlayerHaveGame`**

   Verifica se um jogador possui um determinado jogo. Recebe o ID do jogador e a referência do jogo como parâmetros e retorna um valor booleano indicando se o jogador possui o jogo. A função realiza uma consulta na tabela "compra"procurando uma correspondência entre o ID do jogador e a referência do jogo. Se encontrar pelo menos um registo, retorna verdadeir, caso contrário, retorna falso.

4. **Função 4: `isBadgeInGame`**

   Verifica se um crachá está associado a um jogo específico. Recebe o nome do crachá e a referência do jogo como parâmetros e retorna um valor booleano indicando se o crachá está associado ao jogo. A função realiza uma consulta na tabela "cracha"procurando uma correspondência entre o nome do crachá e a referência do jogo. Se encontrar pelo menos um registo, retorna verdadeiro, caso contrário, retorna falso.

5. **Função 5: `getBadgePointsLimit`**

   Obtém o limite de pontos necessário para obter um crachá específico de um jogo. Recebe o nome do crachá e a referência do jogo como parâmetros e retorna um valor inteiro indicando o limite de pontos para o crachá. A função realiza uma consulta na tabela "cracha"procurando o registro correspondente ao nome do crachá e à referência do jogo e retorna o valor do campo "limite_de_pontos".

6. **Função 6: `getPlayerPoints`**

Obtém a pontuação de um jogador numa certa partida. Recebe o ID do jogador, a referência do jogo, o número da partida e o tipo de partida como parâmetros e retorna um valor inteiro indicando a pontuação do jogador. A função verifica o tipo de partida (normal ou multi-jogador) e realiza uma consulta nas tabelas "Pontuacoes_Normal"ou "Pontuacoes_Multi_Jogador"para obter a pontuação correspondente ao jogador, referência do jogo e número da partida.

7. **Função 7: `doesPlayerAlreadyHaveBadge`**

   Verifica se um jogador já possui um crachá específico. Rrecebe o ID do jogador e o nome do crachá como parâmetros e retorna um valor booleano indicando se o jogador já possui o crachá. A função realiza uma consulta na tabela "CrachaJogador"procurando uma correspondência entre o ID do jogador e o nome do crachá. Se encontrar pelo menos um registo, retorna verdadeiro, caso contrário, retorna falso.

8. **Função 8: `assignBadgeToPlayer`**

   Atribui um crachá a um jogador. Recebe o ID do jogador e o nome do crachá como parâmetros e não retorna nenhum valor. A função insere um novo registo na tabela "CrachaJogador"com o nome do crachá e o ID do jogador, associando assim o crachá ao jogador.

3

Organização modelo JPA

3.1 Organização geral

Na nossa implementação fizemos uso de Repositories e Mappers, usando interfaces para as ligar com o nível aplicacional dos Serviços. O Entity Manager gere as entidades do JPA e o data scope gere as transições requisitadas. [1]

3.1.1 Repositories

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

3.1.1.1 Interface

- List<Tentity> getAll() throws Exception;
- Tentity find(Tkey k) throws Exception;
- void add(Tentity entity) throws Exception;
- void delete(Tentity entity) throws Exception;
- void save(Tentity e) throws Exception;

3.1.1.2 Implementação Exemplo de um Repositório

O leitor pode averiguar a implementação da interface dos Repositórios no anexo A.2.

3.1.2 Mappers

Os Mappers, à semelhança dos Repositories, utilizam interfaces para cada entidade e implementam as seguintes funções:

3.1.2.1 Interface

- Tkey create(Tentity e) throws Exception;
- Tentity read(Tkey k) throws Exception;
- void update(Tentity e) throws Exception;
- void delete(Tentity e) throws Exception;

Os Mappers limitam-se a implementar as funções CRUD, para justifcar a simplicidade da sua implementação e o seu retorno.

3.1.2.2 Implementação Exemplo de um Mapper

O leitor pode averiguar a implementação da interface dos Mappers no anexo A.1.

4

Detalhes de Implementação

4.1 Isolamento Transacional

Recorremos ao uso do DataScope para garantir isolamento operacional e transacional entre operações e acessos à camada de dados. O funcionamento do mesmo bem como os seus detalhes de implementação podem ser analisados no seu ficheiro. A.4

4.2 Níveis de Isolamento

Não foi colocado de forma explícita o nível de isolamento por operação nesta fase ao nível do JPA, assumimos portanto que neste trabalho as transações correrão com o nível de isolamento pre-definido pelo Postgres que será o Read Commited a não ser que este mesmo seja elevado pelo utilizador.

4.3 Testes

Devido à escassez de tempo o grupo não estendeu o suporte a testes nesta camada tendo ficado somente com aqueles já implementados na fase anterior. Convém realçar também que o teste à inserção de um crachá no Jogador com o uso do Lock Optimista não foi realizado num ambiente assíncrono com múltiplas transações a tentar realizar a mesma operação.

Referências

[1] S. Sudarshan Abraham Silberschatz Henry F. Korth, *Database System Concepts 7th ed.* McGrawHill Education, 2020.

[2] Shamkant B. Navathe R. Elmasri, *Fundamentals of Database System 7th ed.* Pearson Education, 2016.

[3] Andreas Reuter Jim Gray, *Transaction Processing: Concepts and Techniques 5th ed.* Morgan Kaifmann, 1993.

Anexo

No presente anexo estão presentes o código que foi sendo mencionado ao longo do relatório.

A.1 Exemplo de Implementação de Mapper

```
1  public class MapperCompra{
2
3     private EntityManagerFactory emf;
4     private EntityManager em;
5
6     public CompraKey create(Compra v) throws Exception {
7        try (DataScope ds = new DataScope()) {
8           EntityManager em = ds.getEntityManager();
9           em.flush();
10          em.persist(v);
11          ds.validateWork();
12          return v.getId();
13       } catch (Exception e) {
14          System.out.println(e.getMessage());
15          throw e;
16       }
17    }
18
```

```
19  public Compra read(Integer jogador, String jogo) throws Exception {
20    try (DataScope ds = new DataScope()) {
21      if (jogo == null && jogador == null) return null;
22      EntityManager em = ds.getEntityManager();
23      em.flush();
24      CompraKey cmId = new CompraKey();
25      cmId.setJogador(jogador);
26      cmId.setJogo(jogo);
27      Compra a = em.find(Compra.class, cmId,LockModeType.PESSIMISTIC_READ )
        ;
28      ds.validateWork();
29      return a;
30    }
31    catch(Exception e)
32    {
33      System.out.println(e.getMessage());
34      throw e;
35    }
36  }
37
38  public void update(Compra a) throws Exception {
39    try (DataScope ds = new DataScope()) {
40      EntityManager em = ds.getEntityManager();
41      em.flush();
42      Compra a1 = em.find(Compra.class,
43            a.getId(),LockModeType.PESSIMISTIC_WRITE );
44      if (a1 == null)
45        throw new java.lang.IllegalAccessException("Entidade inexistente");
46      a1.setPreco(a.getPreco());
47      a1.setData(a.getData());
48      ds.validateWork();
49    }
50    catch(Exception e) {
51      System.out.println(e.getMessage());
52      throw e;
53    }
54  }
55
```

```
56    public void delete(Compra v) throws Exception {
57      try (DataScope ds = new DataScope()) {
58        EntityManager em = ds.getEntityManager();
59        em.flush();
60        Compra v1 = em.find(Compra.class, v.getId(), LockModeType.
        PESSIMISTIC_WRITE);
61        if (v1 == null)
62          throw new java.lang.IllegalAccessException("Entidade inexistente");
63        em.remove(v1);
64        ds.validateWork();
65      } catch (Exception e) {
66        System.out.println(e.getMessage());
67        throw e;
68      }
69    }
70
71    public CompraKey firstId() throws Exception {
72      emf = Persistence.createEntityManagerFactory("projetoBase");
73      em = emf.createEntityManager();
74
75      try {
76        CompraRepository rv = new CompraRepository();
77        return rv.getAll().get(0).getId();
78      } catch (Exception e) {
79        System.out.println(e.getMessage());
80        throw e;
81      } finally {
82        em.close();
83        emf.close();
84      }
85    }
86  }
```

Listagem A.1: Mapper Exemplo

A.2 Exemplo de Implementação Repositorio

```
1  public class CompraRepository implements IRepository<Compra, CompraKey> {
```

```
2
3   public List<Compra> findAll() {
4     EntityManagerFactory emf = Persistence.createEntityManagerFactory("
      ProjetoBase");
5     EntityManager em = emf.createEntityManager();
6     return em.createNamedQuery("Compra.getAll", Compra.class).getResultList()
      ;
7   }
8   @Override
9   public List<Compra> getAll() throws Exception {
10    try (DataScope ds = new DataScope()) {
11      EntityManager em = ds.getEntityManager();
12      List<Compra> l = em.createNamedQuery("Compra.getAll", Compra.class).
      getResultList();
13      ds.validateWork();
14      return l;
15    }
16    catch(Exception e)
17    {
18      System.out.println(e.getMessage());
19      throw e;
20    }
21  }
22
23  @Override
24  public Compra find(CompraKey id) throws Exception {
25    MapperCompra m = new MapperCompra();
26    try {
27      return m.read(id.getJogador(),id.getJogo());
28    } catch (Exception e) {
29      System.out.println(e.getMessage());
30      throw e;
31    }
32  }
33
34  @Override
35  public void add(Compra entity) throws Exception {
36    MapperCompra mcp = new MapperCompra();
```

```
37          try {
38              mcp.create(entity);
39          } catch (Exception e) {
40              System.out.println(e.getMessage());
41              throw e;
42          }
43      }
44
45      @Override
46      public void delete(Compra entity) throws Exception {
47          MapperCompra mcp = new MapperCompra();
48          try {
49              mcp.delete(entity);
50          } catch (Exception e) {
51              System.out.println(e.getMessage());
52              throw e;
53          }
54      }
55
56      @Override
57      public void save(Compra entity) throws Exception {
58          MapperCompra mcp = new MapperCompra();
59          try {
60              mcp.update(entity);
61          } catch (Exception e) {
62              System.out.println(e.getMessage());
63              throw e;
64          }
65      }
66  }
```

Listagem A.2: Repositório Exemplo

A.3 Exemplo de Implementação da Interface AbstractDataScope

```
1  package Dal;
2
```

```
3  import jakarta.persistence.EntityManager;
4  import jakarta.persistence.EntityManagerFactory;
5  import jakarta.persistence.Persistence;
6
7  public abstract class AbstractDataScope implements AutoCloseable {
8
9    protected class Session {
10     private EntityManagerFactory ef;
11     private EntityManager em;
12     private boolean ok = true;
13
14   }
15
16   boolean isMine = true;
17   boolean voted = false;
18
19   private static final ThreadLocal<Session> threadLocal = ThreadLocal.withInitial(()
       -> null);
20
21   public AbstractDataScope() {
22     if (threadLocal.get()==null) {
23       EntityManagerFactory emf = Persistence.createEntityManagerFactory("
       projetoBase");
24       EntityManager em = emf.createEntityManager();
25       Session s = new Session();
26       s.ef = emf;
27       s.ok = true;
28       s.em = em;
29       threadLocal.set(s);
30       em.getTransaction().begin();
31       isMine = true;
32     }
33     else
34       isMine = false;
35   }
36
37   public EntityManager getEntityManager() {return threadLocal.get().em;}
38
```

```
39    @Override
40    public void close() throws Exception {
41      // TODO Auto-generated method stub
42      if (isMine) {
43        if(threadLocal.get().ok && voted)  {
44          threadLocal.get().em.getTransaction().commit();
45        }
46        else
47          threadLocal.get().em.getTransaction().rollback();
48        threadLocal.get().em.close();
49        threadLocal.get().ef.close();
50        threadLocal.remove();
51        //ou:
52        //threadLocal.set(null);
53      }
54      else
55      if (!voted)
56        cancelWork();
57
58    }
59
60    public void validateWork() {
61      voted = true;
62    }
63
64    public void cancelWork() {
65      threadLocal.get().ok = false;
66      voted = true;
67    }
68  }
```

Listagem A.3: Interface DataScope

A.4 Exemplo de Implementação da DataScope

```
1  public class DataScope extends AbstractDataScope implements AutoCloseable {
2
3    public DataScope() throws Exception {
```

```
4         super();
5     }
6
7     public List<Compra> getAllCompras() throws Exception {
8         return new CompraRepository().getAll();
9     }
10
11    public Compra findCompra( Integer jogador, String jogo) throws Exception {
12        return new MapperCompra().read(jogador, jogo);
13    }
14
15    public CrachaJogador findCrachaJogador(String cracha, int jogador) throws
          Exception {
16        return new MapperCrachaJogador().read(cracha, jogador);
17    }
18 }
```

Listagem A.4: DataScope