ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

GameOn

Trabalho Prático – 2ª Fase

Daniel Caseiro
Henrique Fontes

Orientadores Afonso Remédios
Nuno Leite
Walter Vieira

Relatório GameOn (2ª Fase) realizado no âmbito de Sistemas de Informação,
do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2022/2023

Junho de 2023

Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

GameOn

Trabalho Prático – 2ª Fase

46052 Daniel André Caseiro
48295 Henrique Fontes

Orientadores:

Afonso Remédios
Nuno Leite
Walter Vieira

Relatório GameOn (2ª fase) realizado no âmbito de Sistemas de Informação,
do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2022/2023

Maio de 2023

Resumo

O projeto consiste na criação de um sistema de gestão de jogos, onde é possível criar jogadores, jogos, partidas, atribuir crachás, iniciar conversas e outras funcionalidades relacionadas.

A primeira fase do projeto resume-se à implementação do modelo de dados, incluindo todas as restrições de integridade, e a criação do código PL/pgSQL que permite criar o modelo físico, remover o modelo físico e preencher a base de dados. Também foram criadas funções, procedimentos armazenados, uma vista e gatilhos.

Na segunda e presente fase do projeto o intuito foi a manipulação de dados através da especificação JPA. Foram implementados diversos componentes que permitem aceder a processos já estabelecidos na primeira fase do projeto, tal como a adição de novas funcionalidades. Para isso, foram criados diferentes mecanismos com o objetivo de simplificar processos, tal como a criação de funções genéricas.

Conclui-se que a criação de um sistema de gestão de jogos é uma tarefa complexa, mas com a implementação adequada do modelo de dados e do código PL/pgSQL e JPA, é possível criar um sistema eficiente e escalável. Por fim, é possível constatar que a implementação de mecanismos automáticos ajuda a garantir a integridade dos dados da base de dados.

Palavras-chave: funções; base de dados; chat; crachá; integridade; jogador; PL/pgSQL; pontos; procedimento armazenado; restrições; SQL; vista; JPA.

Abstract

The project consists of creating a game management system where it is possible to create players, games, matches, assign badges, start chats, and other related functionalities.

The first phase of the project involved implementing the data model, including all integrity constraints, and creating the PL/pgSQL code that allows creating the physical model, removing the physical model, and populating the database. Functions, stored procedures, a view, and triggers were also created.

In the second and current phase of the project, the focus was on data manipulation through the JPA specification. Various components were implemented to access processes established in the first phase of the project, as well as to add new functionalities. For this purpose, different mechanisms were created to simplify processes, such as the creation of generic functions.

It can be concluded that creating a game management system is a complex task, but with the proper implementation of the data model and PL/pgSQL and JPA code, it is possible to create an efficient and scalable system. Finally, it can be observed that the implementation of automatic mechanisms helps ensure data integrity in the database.

Keywords: functions; database; chat; badge; integrity; player; PL/pgSQL; points; stored procedure; constraints; SQL; view; JPA.

Índice

Lista de Figuras

1. Introdução

Este relatório descreve o trabalho realizado com o objetivo de atingir determinados objetivos de aprendizagem. A tarefa envolve a criação de uma aplicação Java que ofereça diversas funcionalidades, conforme definido na fase 1 deste projeto.

A primeira parte do trabalho consiste em desenvolver uma aplicação capaz de aceder as funcionalidades 2d a 2l, conforme descrito na fase inicial. Essas funcionalidades envolvem a manipulação de dados relacionados a jogos, jogadores, partidas, atribuição de crachás, conversas e outras funcionalidades correlacionadas. É necessário implementar a lógica que permite interagir com essas funcionalidades de forma adequada.

Além disso, é necessário implementar a funcionalidade 2h sem recorrer a qualquer procedimento armazenado ou função PL/pgSQL. Essa funcionalidade, descrita na fase 1 do trabalho, requer uma abordagem diferente para sua execução, utilizando outros recursos disponíveis na aplicação Java.

Posteriormente, na segunda parte do trabalho, o objetivo é utilizar o mecanismo de optimistic locking para aumentar em 20% o número de pontos associados a um crachá específico. Isso é realizado fornecendo o nome do crachá e o identificador do jogo ao qual ele pertence. Se necessário, é permitido realizar alterações no esquema do banco de dados para suportar essa modificação. Além disso, é importante testar essa funcionalidade, identificando e tratando adequadamente situações de alteração concorrente conflituante que possam inviabilizar a operação.

No relatório, serão descritas a forma como as situações de erro foram criadas para testar a alínea anterior, bem como as estratégias utilizadas para apresentar mensagens de erro adequadas aos usuários da aplicação.

Por fim, a terceira parte do trabalho consiste em repetir a tarefa anterior, ou seja, realizar o aumento de pontos associados a um crachá, mas utilizando o controle de concorrência pessimista. Será necessário explorar as funcionalidades disponíveis nesse mecanismo para garantir a integridade e consistência dos dados durante a operação.

Este relatório fornecerá uma visão geral de cada uma dessas etapas, descrevendo os passos tomados, as decisões de implementação, os resultados obtidos e as conclusões alcançadas ao final do projeto.

2. Formulação do Problema

O objetivo principal deste projeto é a criação de uma aplicação Java para um sistema de gestão de jogos, visando enfrentar diferentes desafios e alcançar diversas metas. Os resultados esperados abrangem a criação da aplicação Java, que inclui funcionalidades como a criação de jogadores, jogos, partidas, atribuição de crachás, início de conversas e outras tarefas relacionadas.

De forma a ser construída uma aplicação que execute todos os processos necessários de forma eficaz e simples, é necessário construir uma interface intuitiva, que disponibilize as operações requisitadas para esta fase do projeto de forma clara. Para isso procura-se a implementação de mecanismos de entrada de dados, tal como de exposição de dados requisitados pelo utilizador.

A solução que procuramos passa por uma solução genérica que visa simplificar processos e diminuir substancialmente a quantidade de código escrito.

Para esta implementação será necessário aceder a funcionalidades implementadas na primeira fase do projeto através de acessos diretos à base de dados , bem como através de componentes ORM estabelecidos.

Para além da implementação do acesso às funcionalidades previamente construídas, é necessário também implementar a funcionalidade do exercício 2h) da passada fase do projeto. Esta funcionalidade visa a existência de duas implementações distintas: a utilização de procedimentos armazenados ou funções PL/pgSQL, buscando uma solução alternativa com recursos disponíveis na aplicação Java, tal como a procura de uma solução reutilizando os procedimentos armazenados que a funcionalidade original contém.

Outro dos objetivos a esclarecer é a utilização do mecanismo de *optimistic* e *pessimistic locking*. Desta forma garante-se que as transações estarão cobertas de uma camada que saiba lidar com possíveis conflitos que possam ocorrer. A utilização destes mecanismos ocorre na implementação de uma nova funcionalidade, que visa o aumento de 20% do número de pontos associados a um determinado crachá específico. Pretende-se também a realização de uma função que teste essa solução e que apresente uma mensagem de erro adequada em caso de alteração concorrente conflitante que inviabilize a operação

Resumidamente, o objetivo é desenvolver uma aplicação Java robusta, eficiente e escalável, capaz de gerir jogos de forma adequada, mantendo a integridade dos dados e proporcionando uma boa experiência para os utilizadores.

3. Solução Proposta - Grandes Ideias

A solução proposta para abordar os desafios e metas estabelecidos no enunciado do projeto de criação de uma aplicação Java para um sistema de gestão de jogos pode ser dividida em duas partes: a construção dos componentes e estrutura a ser utilizados e a resolução dos problemas propostos. A primeira parte visa a facilitação e simplificação da segunda, que depende desta para funcionar conforme o requisitado.

3.1 Mapeamento Objeto-relacional

O processo de criação das entidades JPA para o relatório envolve identificar as entidades principais do sistema e criar classes Java correspondentes a elas. Essas classes são mapeadas como entidades persistentes usando as anotações apropriadas. Os atributos são definidos e mapeados para as colunas do banco de dados, e as relações entre as entidades são estabelecidas. Em seguida, o provedor JPA é configurado no projeto através de um arquivo de configuração. Com as entidades prontas, é possível utilizar as operações de persistência fornecidas pela API JPA para interagir com o banco de dados, realizando consultas e manipulando os dados. As entidades JPA facilitam a persistência e manipulação dos dados no contexto do projeto.

O uso do JPA Buddy auxiliou significativamente no processo de criação das entidades JPA para o relatório. O JPA Buddy é uma ferramenta que oferece recursos avançados para a geração automática de código JPA. Foi assim possível acelerar o desenvolvimento das entidades, pois são fornecidas funcionalidades como a geração automática de classes de entidade a partir do banco de dados e vice-versa. Isso economizou tempo e reduziu a quantidade de código manual necessário. Além disso, o JPA Buddy também facilitou a configuração do provedor JPA, permitindo uma integração suave com o projeto. Em resumo, o uso do JPA Buddy agilizou o processo de criação das entidades JPA, proporcionando maior produtividade e simplificando as tarefas de mapeamento e configuração.

3.2 Componentes ORM

O código apresentado inclui um Mapper genérico chamado GenericMapper, que implementa a interface IMapper. Esse Mapper genérico possui métodos para criar, ler, atualizar e excluir entidades. Ele foi projetado para ser flexível e pode ser usado com diferentes tipos de entidades e chaves. O GenericMapper utiliza o objeto DataScope e o EntityManager para gerir transações e operações de acesso ao banco de dados. Essa implementação do Mapper genérico facilita o desenvolvimento de operações CRUD, permitindo a reutilização de código e simplificando o acesso aos dados das entidades.

Além disso, o código também apresenta um Repositório genérico chamado GenericRepository, que implementa a interface IRepository. Esse Repositório genérico permite realizar operações de acesso aos dados, como adicionar, excluir, atualizar e buscar elementos. O GenericRepository utiliza o Mapper genérico GenericMapper para executar essas operações, aproveitando a flexibilidade e a funcionalidade fornecidas pelo Mapper genérico. Essa implementação do Repositório genérico simplifica o acesso aos dados das entidades, promovendo a reutilização de código e facilitando as operações básicas de acesso ao banco de dados.

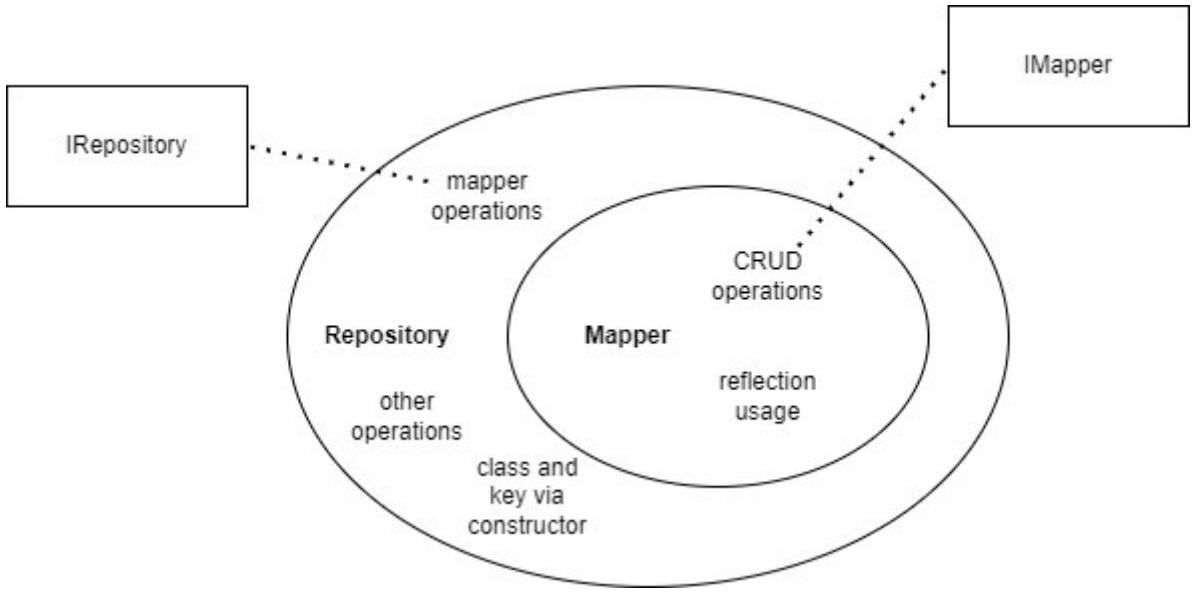

Figura 1 - Repository and Mapper Diagram

3.3 Componentes

Esta secção dedica-se à apresentação das soluções encontradas para a criação de componentes que procuram funcionar de forma consistente e regular nas implementações a ser estabelecidas.

3.3.1 Datascope

O código apresentado inclui a classe *DataScope*, que é uma subclasse da classe abstrata *AbstractDataScope*, que implementa também a interface *AutoCloseable*. A classe estende a funcionalidade do *AbstractDataScope* para criar um âmbito de dados específico.

A classe DataScope herda todos os métodos e comportamentos definidos na classe *AbstractDataScope*, como o controlo de transações, o acesso ao EntityManager e a validação do trabalho realizado. Esta também adiciona o seu próprio construtor, que chama o construtor da classe pai usando a palavra-chave *super*.

A utilização da classe *DataScope* permite criar um âmbito de dados delimitado, em que as transações são geridas automaticamente e as operações de acesso à base de dados são executadas de forma segura. Ao finalizar o âmbito, o encerramento automático é garantido pelo mecanismo *AutoCloseable*.

3.3.2 App

A classe *App r*epresenta a aplicação desenvolvida ao longo do projeto*,* oferecendo uma interface de linha de comando para possibilitar a interação com as funcionalidades do sistema de gestão de jogos, tratando exceções e apresentando resultados. Esta classe permite que o usuário execute, para além das novas funcionalidades implementadas, diversas operações realizadas na primeira fase do projeto.

3.3.3 Executor Operation

Esta classe é responsável pela execução das diversas operações do sistema de gestão de jogos. Possui um construtor que recebe um *Entity Manager* como parâmetro de forma a manter as transações consistentes e anotações que distinguem se os métodos das operações em questão são funções, vistas ou procedimentos armazenados.

As classes *ExecutorDB* e *RegisterDB* são utilizadas aqui de forma a executar as operações e registar funções e vistas relacionadas a estas operações, respectivamente.

3.3.4 Register DB

O intuito da criação da classe *Register DB* é o registo das funções e vistas relacionadas às operações. Para isso, esta classe recebe também um *Entity Manager* no construtor e é utilizada na classe previamente mencionada *Executor Operation.* De forma a realizar estes registos, utiliza-se a classe *Service* para registar estas operações, isto após serem estabelecidos os nomes e argumentos dos métodos a serem invocados.

3.3.5 Executor DB

A classe *Executor DB* encontra-se no mesmo nível da classe *Register DB* e por isso também recebe o mesmo *Entity Manager* da classe *Executor Operation,* que a invoca.

A primeira operação realizada por esta classe é o método *executeMethod()* que obtém um método através de técnicas de reflexão e através deste obtém as anotações referidas anteriormente, que distinguem se o método é uma função, vista ou procedimento armazenado. Após esta distinção o método correspondente é invocado, executa-se a função correspondente vinda da classe *Service* e apresenta-se os resultados caso estes existam.

3.3.6 Service

Esta classe é utilizada pelas classes *Register DB* e *Executor DB*. Regista-se determinada função ou vista através da classe *Register DB* e associa-se à variável *currentFunction*, que depois será executada posteriormente via *Executor DB.*

Figura 2 - Diagrama de Classes

3.4 Resolução dos problemas propostos

A realização das funcionalidades propostas pode ser dividida em três partes principais: o acesso às funcionalidades *2d)* a *2l)*, a implementação da alínea 2h em *JPA* com e sem recurso às funções e procedimentos armazenados realizados previamente em *SQL* e por último a nova operação que acrescenta 20% dos pontos da pontuação de um determinado *Cracha*.

3.4.1 Acesso às funcionalidades da Fase 1

De forma a distinguir as funções dos procedimentos, foram implementadas 3 anotações *@Function*, *@Procedure* e *@View*. Assim, quando estamos a analisar a operação em causa,

verificamos qual das anotações está presente na mesma e executamo-la da forma correta consoante o seu tipo.

3.4.1.1 Operações

3.4.1.1.1 Funções

Como foi referido anteriormente, no caso de uma função esta vai estar presente com a anotação *@Function,* identificando-a assim como uma função. Sobre a mesma transação, esta vai ser registada corretamente, evidenciando os seus parâmetros de entrada e saída, e posteriormente é executada.

Na execução da mesma, aí é feita a verificação da presença da anotação e o direcionamento para o "*executor*" correspondente. Inicialmente, foi utilizado um *HashMap* para guardar o nome das funções e as suas *query's* correspondentes. Contudo, como o registo e execução são feitos na mesma transação, criámos um variável estática *currentFunction* que vai guardar a *query* da função no registo e, aquando da sua execução, vai utilizar esta variável para efetuar a execução da mesma.

Assim sendo, é realçado o uso das funções *registerFunction*, *executeFunction*, *createPlaceholders* e *prepareArgs*, de forma a possibilitar um registo e execução eficaz e eficiente.

Por fim, a função *displayResults* mostra os resultados obtidos da função.

3.4.1.1.2 Procedimentos

À semelhança das funções, os procedimentos também possuem uma anotação para a diferenciação dos mesmos. Deste modo, estes também serão filtrados e, ao contrário das funções, não existe necessidade de registar estes. Depois da sua filtragem através do uso da anotação, é utilizado o "*executor*" correspondente ao procedimento.

Na execução do procedimento, a função *executeProcedure* é utilizada e como o nome indica, esta serve para executar o procedimento em causa. Em adição, é utilizada a função *preparaArgs* para preparar os argumentos relativos ao procedimento.

3.4.1.1.3 Vista

No caso da Vista, esta é identificada utilizando a anotação *@View* e posteriormente executada. Primeiramente é registada a view utilizando a função *registerView* à semelhança do *registerFunction,* afetando assim o valor em *currentFunction*.

De seguida, é verificada a presença da anotação e executada a função *executeView* que vai obter o valor na variável estática *currentFunction* através da chamada da função *getResultList* retornando assim a lista correspondente à vista em causa.

Por fim, é chamada a função *displayView* que vai mostrar a lista previamente obtida.

3.4.1.2 Detalhes de Implementação - Operações

3.4.1.2.1 Função – Registo e Execução

O registo e execução de uma operação é realizado na mesma transação, ou seja, sobre o mesmo *DataScope*.

Numa fase inicial, para guardar as funções e as suas query's correspondentes, foi utilizado um *hashmap* e o seu registo era feito sobre uma única transação na inicialização do programa. Contudo, como a *query* associa a transação que foi utilizada na sua criação e esta já não existia após a primeira execução, o *hashmap* tornou-se uma ideia pouco viável.

A solução foi *currentFunction*, uma variável estática que vai ser reutilizada a cada função, assim como o registo e execução são feitos na mesma transação, nunca vamos utilizar uma *query* de uma transação que já não está ativa.

3.4.2 Implementação da alínea 2h em JPA

3.4.2.1 Utilização dos Repositórios

A implementação da alínea h), mais concretamente, o procedimento *associarCracha,* é realizado utilizando as classes *Mapper* e *Repository* que implementam *IMapper* e *IRepository* respetivamente.

Para cada operação numa dada tabela é necessário instanciar um repositório correspondente a essa tabela que quer ser verificada ou afetada. Neste caso, são utilizados os repositórios: *jogadorRepository*, *crachaRepository* e *jogoRepository*, de forma a verificar se o jogador, crachá e jogo existem.

Posteriormente, são utilizados os repositórios *partidaRepository* e *pontuacaoRepository* de forma a verificar se o jogador possui o mesmo jogo que o crachá está associado e se a pontuação do jogador é igual ou superior à pontuação associada ao crachá.

Por fim, após todas as condições estarem confirmadas, ocorre a associação do jogador ao crachá. Para isso, é instanciado o repositório *crachaJogador* de modo a realizar uma inserção na tabela cracha_jogador com o jogador e crachá em causa.

3.4.2.2 Detalhes de Implementação – Repository/Mapper

3.4.2.2.1 Instanciação de Repositórios

A instanciação de um repositório é realizada da seguinte maneira:

```
GenericRepository<Jogador, Integer> jogadorRepository = new
GenericRepository<>(Jogador.class, Integer.class);
```

Neste caso, o jogadorRepository, como o nome indica, vai ser um repositório específico da tabela Jogador. No construtor da classe *GenericRepository* é passado o tipo da classe do modelo e o tipo da chave primária associada a esse modelo.

Ainda no construtor desta classe é instanciado um mapper com os mesmos tipos passados no construtor da classe GenericRepository.

3.4.2.2.2 Instanciação de Mappers

A instanciação de um mapper é realizada da seguinte maneira:

```
private final IMapper<Tentity, Tkey> mapper;

Dentro do constructor:
this.mapper = new GenericMapper<>(entityClass, idClass);
```

Neste caso, após ser instanciado o mapper no construtor do repositório, o campo *mapper* vai ficar o com valor deste mapa criado.

Por fim, caso seja efetuada uma operação de *create*, *read*, *update* ou *write* (CRUD) correspondente ao repositório (*add*, *find* ...) vai ser utilizada a instância *mapper* para chamar a operação correspondente.

3.4.2.2.3 Função Auxiliar ExtractId em GenericMapper

De modo a obter o valor da chave corresponde a um tipo de um modelo em específico, por exemplo, dado um determinado jogador eu conseguir obter a sua chave, foi criada a função *extractId.* Esta função utiliza reflexão sobre a classe da entidade em causa que foi passada no constructor do mapa e vai buscar a referência para o método *getId* dessa entidade. Por sua vez, ao utilizar o método *invoke* sobre *getId* consigo obter a chave primária desse objeto dessa entidade.

Contudo, apesar da implementação ser genérica contém uma limitação. Todas as entidades têm de conter o método *getId* e todos os métodos *getId* têm de obrigatoriamente retornar a chave primária. Na nossa opinião, este é apenas um detalhe de implementação que também já é seguido pela própria abordagem de modelação de qualquer aplicação usando JPA.

3.4.3 Implementação da Alínea 2

O objetivo desta alínea é aumentar em 20% o número de pontos associados a um crachá, dados o nome do crachá e o identificador do jogo a que ele pertence.

3.4.3.1 Optimistic e Pessimistic Locking

A assinatura da função utilizada para tal é:

```
public static void crachaIncreasePoints(String idJogo, String nomeCracha, LockModeType
lockModeType)
```

Em ambas as técnicas utilizadas, *Optimistic* e *Pessimistic Locking* é usada a função *crachaIncreasePoints.* Contudo, aquando da chamada deste o *LockModeType* diferente consoante a técnica pretendemos utilizar.

A função *crachaIncreasePoints* consiste apenas numa leitura para obter o crachá utilizado o *lockmode* escolhido, seguido da verificação da existência do mesmo. Por fim, chamamos a função *increasePointsByTwentyPercent* que apenas multiplica a pontuação desse crachá por 1.2, de forma a aumentar em 20% a pontuação deste.

Por fim, está presente um *catch* que permite capturar as exceções *OptimisticLockException* e *PessimisticLockException*.

4. Avaliação Experimental

O único teste desenvolvido nesta fase é sobre a alínea 2, nomeadamente o *Optimistic* e *Pessimistic Locking*.

A ideia do teste é ao utilizar 2 *threads* diferentes e realizar a operação de incremento de 20% a pontuação de um dado crachá, isto em simultânea, esperar que ocorra uma exceção.

Assim sendo, são criados parâmetros de teste *idJogo* e *nomeCracha*. São criadas as *threads* e cada uma delas vai correr individualmente a função *crachaIncreasePoints.* De seguida, é obtido o resultado da execução de ambas as *threads* o que irá resultar numa exceção. Verificando assim, se a exceção é capturada com sucesso.

5. Conclusões

Neste relatório foi abordada a segunda fase do trabalho, na qual o foco foi o desenvolvimento de uma camada de acesso a dados utilizando uma implementação de JPA e um subconjunto dos padrões de desenho. Além disso, foi desenvolvido também uma aplicação em Java que fizesse o uso adequado dessa camada de acesso a dados.

Durante esta fase, para além da atenção dada à correta utilização do processamento transacional, por meio dos mecanismos disponíveis no JPA, também foi tido em conta a integridade dos dados e a consistência das operações. Foi enfatizada a importância de libertar adequadamente as conexões e recursos quando estes não estiverem em uso, visando a eficiência e o bom desempenho da aplicação.

Foram desenvolvidos testes adequados para identificar situações de alteração concorrente conflitante que pudessem inviabilizar a operação, que expressam mensagens de erro adequadas nesses casos.

No desenvolvimento do trabalho, foram valorizados o tratamento de erros em todas as etapas, bem como a gestão transacional, utilizando o nível de isolamento adequado de forma explícita.

Em termos de organização do código, o foco foi simplificar o código ao máximo utilizando componentes comuns de forma estruturada.

Referências

[1] Fundamentals of Database Systems (7th Edition) Ramez Elmasri, Shamkant B. Navathe Pearson Education, 2015.

[2] Transaction processing : concepts and techniques (5th Edition) Jim Gray, Andreas Reuter Morgan Kaifmann, 1993.