

**Trabalho prático**

**Sistemas de Informação**

**Fase 2**

|  |  |
| --- | --- |
| 47224 | André Graça |
| 49149  48459 | Diogo Guerra  Diogo Santos |

|  |  |
| --- | --- |
| docente | Walter Vieira |
|  |  |

Relatório final realizado no âmbito do trabalho prático de Sistemas de informação, do

curso de licenciatura em Engenharia Informática e de Computadores  
Semestre de Verão 2022/2023

Junho de 2023

# Resumo

Texto do resumo.

Breve descrição do projecto, dos resultados importantes e das conclusões: o objectivo é dar ao leitor uma visão global do projecto (não deve exceder uma página).

**Palavras-chave:** lista de palavras-chave, ordenadas alfabeticamente, separadas por ;.

# Abstract

Abstract text (1 page).

**Keywords:** sorted keyword list, delimited by ;.

**Índice**

[Resumo v](#_Toc417484088)

[Abstract vii](#_Toc417484089)

[Agradecimentos ix](#_Toc417484090)

[Lista de Figuras xiii](#_Toc417484091)

[Lista de Tabelas xv](#_Toc417484092)

[1. Introdução 1](#_Toc417484093)

[1.1 Nome da secção deste capítulo 1](#_Toc417484094)

[1.2 A segunda secção deste capítulo 1](#_Toc417484095)

[1.2.1 A primeira sub-secção desta secção 1](#_Toc417484096)

[1.2.2 A segunda sub-secção desta secção 1](#_Toc417484097)

[1.3 Organização do documento 2](#_Toc417484098)

[2. Formulação do Problema 3](#_Toc417484099)

[2.1 Nome da secção deste capítulo 3](#_Toc417484100)

[2.2 Análise do problema - enumeração 3](#_Toc417484101)

[2.3 Outro problema - tabela 4](#_Toc417484102)

[2.4 Expressões matemáticas 4](#_Toc417484103)

[2.5 Figuras de grande dimensão 4](#_Toc417484104)

[3. Solução Proposta - Grandes Ideias 7](#_Toc417484105)

[3.1 Nome da primeira secção deste capítulo 7](#_Toc417484106)

[3.2 A segunda secção deste capítulo 8](#_Toc417484107)

[3.2.1 A primeira sub-secção desta secção 8](#_Toc417484108)

[3.2.2 A segunda sub-secção desta secção 8](#_Toc417484109)

[3.3 Descrição detalhada da solução 8](#_Toc417484110)

[4. Avaliação Experimental 11](#_Toc417484111)

[4.1 Nome da primeira secção deste capítulo 11](#_Toc417484112)

[4.2 A segunda secção deste capítulo 11](#_Toc417484113)

[4.2.1 A primeira sub-secção desta secção 11](#_Toc417484114)

[4.2.2 A segunda sub-secção desta secção 11](#_Toc417484115)

[4.3 Análise de resultados 11](#_Toc417484116)

[5. Conclusões 13](#_Toc417484117)

[Referências 14](#_Toc417484118)

[A.1 Diagramas da Aplicação 15](#_Toc417484119)

[A.2 Modelos de dados 17](#_Toc417484120)

# 

# Lista de Figuras

[Figura 1 - Legenda da figura com o logotipo do ISEL. 1](#_Toc416101905)

[Figura 2 - Legenda da figura com o logotipo do ISEL – versão 2. 5](#_Toc416101906)

[Figura 3 – Diagrama de casos de utilização. 15](#_Toc416101907)

# Lista de Tabelas

[Tabela 1 -Um exemplo de legenda de tabela. Prazos de entrega de Projecto e Seminário, 4](#_Toc416101908)

# Introdução

# Camada de acesso a dados

Nesta secção, iremos explorar os métodos adotados para a construção de uma cada de acesso a dados. Iremos também abordar a construção das entidades em JPA.

## 2.1 Implementação do Mapper

Uma imagem com texto, captura de ecrã, Tipo de letra

Descrição gerada automaticamentePara a construção da camada de acesso a dados, optamos primeiramente pela implementação de um *Mapper* genérico. A interface Mapper tem métodos equivalentes às operações CRUD que são a create, read, update e delete para manipulação de dados.

Figura 1 - Interface Mapper

A implementação desta interface foi realizada de forma genérica, pois assim não é necessário a implementação de *Mappers*, igual ao número de entidades, visto que a class que implementa esta interface tem natureza genérica, sempre que houver necessidade de criar um Mapper, basta fornecer á class Mapper o tipo da chave primária da entidade e o tipo da entidade.

Uma imagem com texto, captura de ecrã, Tipo de letra, software

Descrição gerada automaticamente

Figura 2 - Implementação da class Mapper

Importante notar que na figura 2 apenas está exposto a implementação do método “create”, todos os 4 métodos foram implementados de forma semelhante. As variáveis *tClass* e *tIdclass* são o tipo da entidade e o tipo da chave primária que são passados ao construtor do Mapper.

## 2.2 Implementação do Repository

## 2.3 Implementação das entidades em JPA

Nesta secção, iremos observar como converter as tabelas da base de dados em entidades do JPA. Relações de grau 1:1 não serão abordadas pois o nosso modelo EA não possui nenhuma.

### Entidades

Uma imagem com texto, captura de ecrã, diagrama, círculo

Descrição gerada automaticamentePara a conversão de entidades do Modelo EA para entidades do JPA, é necessário para cada entidade criar uma class com fields equivalentes às suas colunas, e usar anotações que ajudem a associar as colunas na base de dados aos fields da class.

Figura 3 - Entidade Jogo JPA.

Na figura 3, está representado do lado direito a entidade **Jogo** do modelo EA, que tem 3 atributos id, nome e URL, sendo nome a chave primária. No lado direito, temos o código java equivalente a entidade **Jogo**, a class tem anotações como:

- **@Entity** -> Indica que representa uma entidade.

- **@Table**(name, schema) -> Indica que tabela esta entidade representa, dado o nome da tabela e o schema onde esta se encontra.

- **@Id** -> Indica qual dos fields representa a chave primária.

- **@Column**(name) -> Indica que coluna da tabela o field representa, dado o nome da coluna, esta anotação também recebe outros valores para manter total coerência com a base dados, isto é, outras restrições que possam ter sido impostas á coluna.

Para cada um destes fields também deve ser criado um getter e um setter para fornecer a possibilidade de aceder e alterar o valor durante a execução da aplicação.

No caso de a chave primária ser composta por mais do que um atributo de deve ser feito uma class que representa essa chave, com fields que representam as colunas que formam a chave e getters e setters. Na class deve ser criado um field que é do tipo da chave primária.

### Entidades Fracas e relações de grau 1:N

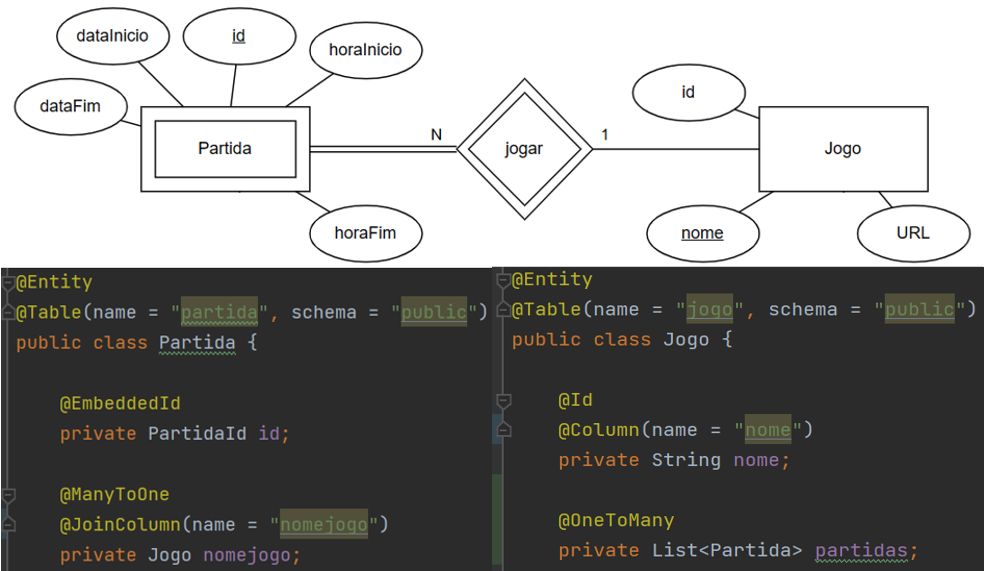
As entidades fracas do modelo EA quando convertidas para JPA, tem de ser indicado a usa dependência através das anotações do JPA. Visto que as relações de grau 1:N e entidades fracas seguem as mesmas estratégias de conversão, serão ambas abordadas nesta secção.

Figura 4 - Entidade Fraca Partida JPA

Na figura 4, podemos observar que partida tem um field do tipo **Jogo** com a anotação **@ManyToOne** que indica que o “N” está do lado da entidade **Partida**. Na class **Jogo** iremos ter uma lista de Partida com a anotação **@OneToMany** que representa que o jogo pode ter várias partidas. A class **Partida** irá conter, tal como mencionado anteriormente, fields equivalentes às colunas da tabela **Partida**, com getters e setters. Isto também se aplica a relações de grau 1:N.

### Relações de grau N:N

Uma imagem com texto, captura de ecrã, diagrama, Tipo de letra

Descrição gerada automaticamente

Na figura 5, podemos observar que ambas as entidades **Jogador** e **Jogo** tem um field com a anotação **@ManyToMany** que indica a relação de N:N e este tem o tipo da entidade do qual partilha essa relação. Na class que representa a relação N:N, tem de ser colocada a anotação **@ManyToOne** no field que faz referência á entidades que usufrui desta relação, logo, a class **Comprar** irá ter 2 fields, um que referencia **Jogador** e outro que referencia **Jogo**. Ambos terão as anotações **@ManyToOne** e **@JoinColumn** para indicar qual coluna é que esta ser feita a referencia.

# Acesso às funcionalidades da BD

Durante esta fase do projeto, foi fornecido um exemplo base para acessar funções e procedimentos armazenados criados em nosso banco de dados, conforme desenvolvido na fase anterior. Para implementar esse exemplo, optamos por dividir e generalizar a execução do código utilizando a reflexão em Java. Essa abordagem não apenas nos permite aplicar o conhecimento adquirido em disciplinas anteriores, mas também facilita a legibilidade e a manutenção do código, como será descrito posteriormente. Nesta secção iremos apresentar as técnicas usadas para acesso às funcionalidades da base de dados desenvolvidas na fase anterior do trabalho prático.

## 3.1 Registo dos parâmetros das funcionalidades

Para acessar uma rotina (função ou procedimento) no nosso projeto, é necessário seguir apenas dois passos simples:

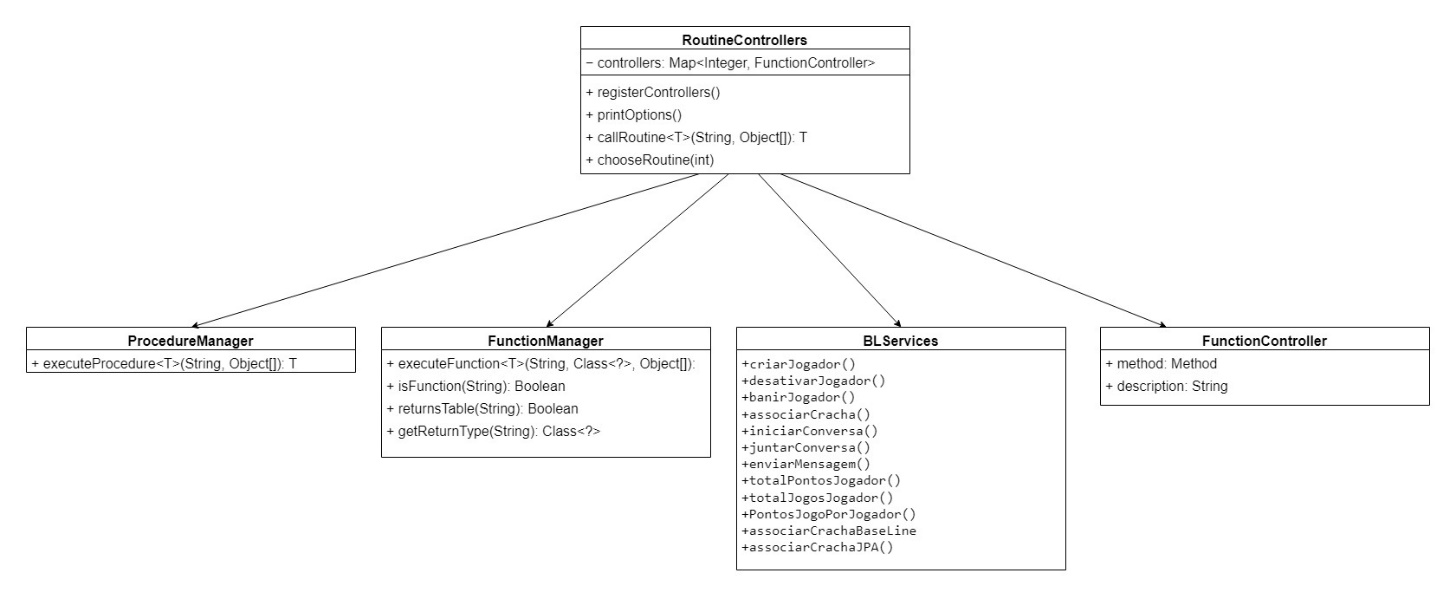
1. Registro dos parâmetros: Primeiramente, é preciso registrar os parâmetros da rotina na classe **RoutineRegisters**. Esses parâmetros podem incluir tanto os parâmetros de entrada como os de saída (caso existam). No entanto, é importante destacar que algumas funções podem não ser consideradas rotinas e dispensar esta etapa de registro (por exemplo, as funções AssociarCrachaBaseline e AssociarCrachaJPA).

2. Criação da função de chamada: Após o registro dos parâmetros, basta criar uma função no **BLServices** que possua os argumentos necessários para serem passados à rotina armazenada, e definir o tipo de retorno. É importante lembrar que, se o tipo de retorno for complexo, será necessário criar uma classe separada para representá-lo e anotá-la corretamente com as anotações do **Jakarta Persistence**. Isso permitirá que o tipo complexo seja convertido corretamente quando proveniente da base de dados.

Essa abordagem divide o código de forma organizada e mantém a manutenção das funcionalidades novas restrita a apenas dois locais. Ao seguir esses passos simples, é possível adicionar e acessar facilmente novas funcionalidades no projeto, mantendo a clareza e a simplicidade do código.

Para possibilitar esse processo, utilizamos a reflexão do Java da seguinte maneira:

1. A aplicação instância um objeto do tipo **BLServices** que por sua vez, no seu construtor, vai chamar a função do **RoutineRegisters** para registar todos os parâmetros de entrada/saída das rotinas necessárias.
2. A aplicação instancia um objeto do tipo **RoutineControllers** e fornece-lhe a dependência dos **BLServices** para este ter acesso aos vários métodos de chamada. Por cada método encontrado que contenha a anotação **Description**, irá ser registado um controller para este, contendo a descrição encontrada e o método em si.
3. Após a instanciação do objeto **RoutineControllers**, a aplicação chama o método *printOptions*, que tem como função exibir ao usuário os vários controllers registrados. Essa exibição apresenta as opções disponíveis para execução dos métodos do **BLServices**.
4. Após a exibição das opções, a aplicação solicita ao usuário que faça a escolha entre os controllers apresentados. O usuário deve fornecer o controllerId correspondente à opção desejada.
5. Com o controllerId escolhido, a aplicação chama o método *chooseRoutine* para obter o método do **BLServices** correspondente a partir do controller selecionado, com isto, por cada parâmetro que o método tenha, é pedido ao user que lhe atribua um valor. Quando todos os argumentos estiverem recebidos, é chamado o método para executar a rotina. Esse método realiza todas as operações necessárias para a execução da rotina, como a chamada à rotina registrada e o tratamento dos resultados.

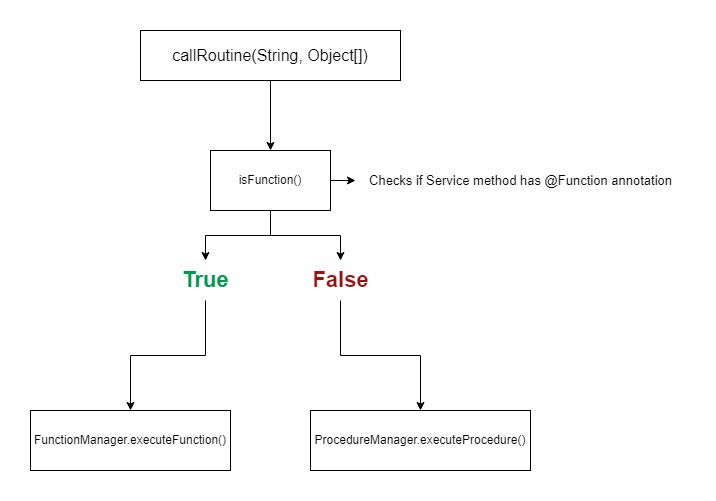


Essa abordagem permite que o usuário selecione a rotina desejada de forma interativa, facilitando a interação com o sistema e proporcionando uma execução personalizada das funcionalidades disponíveis. Além disso, o uso do controllerId garante a correta identificação da rotina a ser executada, garantindo a integridade do sistema.

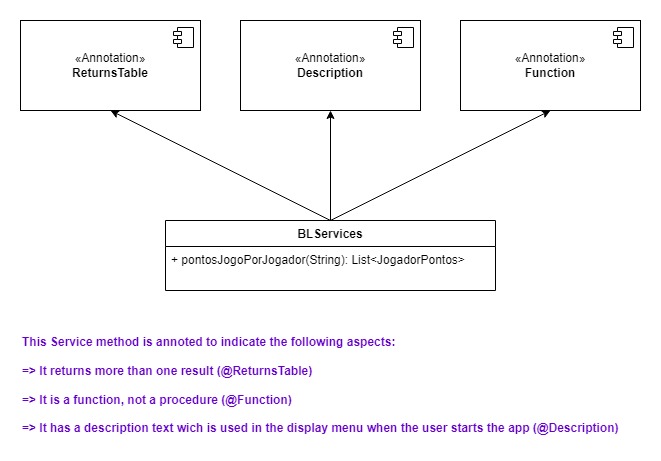
Desta forma, garantimos uma separação clara entre o registo e a lógica de execução das rotinas. Essa abordagem facilita a manutenção e a expansão do código, pois novas rotinas podem ser adicionadas apenas registrando os parâmetros e criando as funções correspondentes, mantendo a estrutura e a organização do projeto.

## Execução de Rotinas armazenadas

Cada método do **BLServices** que queira executar uma rotina, terá que chamar a função *callRoutine* que verifica se é uma função ou não e chama adequadamente o respetivo executor como representado no seguinte esquema:



Como podemos observar, o uso das nossas anotações torna fácil e fulcral para transmitir informação entre classes através de reflexão. Uma função no **BLServices** tem de ser anotada de acordo com a sua funcionalidade, como ilustrado:



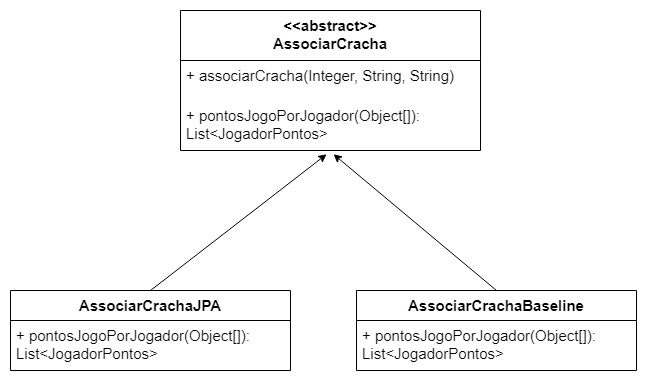
## Implementação do Associar crachá usando/não stored procedures

Para implementar a funcionalidade 2h, descrita na fase 1 deste projeto, sem utilizar procedimentos armazenados ou funções pgSql, foram criadas duas classes: AssociarCrachaJPA e AssociarCrachaBaseline.

A classe AssociarCrachaJPA foi implementada com o objetivo de realizar a funcionalidade 2h utilizando apenas o framework JPA (Java Persistence API). Nessa abordagem, foram utilizados Mappers implementados nesta fase e as funcionalidades oferecidas pelo JPA para manipulação dos dados e realização das operações necessárias como NamedQuerys. Essa classe foi desenvolvida com base nos procedimentos armazenados e funções utilizados pela funcionalidade original, adaptando-os para trabalhar com o JPA.

Já a classe AssociarCrachaBaseline foi implementada com o intuito de reutilizar os procedimentos armazenados e funções utilizados pela funcionalidade original. Essa classe utiliza o *executeFunction* do **FunctionManager** para executar a função *PontosJogoPorJogador*. Dessa forma, é possível aproveitar a lógica já existente e garantir a compatibilidade com a funcionalidade original.

Para evitar a repetição de código e promover a reutilização, optamos por usar uma classe abstrata que contém o método abstrato *PontosJogoPorJogador* que diferá por implementação.



A função associarCracha pertencente à classe base usa a *pontosJogoPorJogador* da classe que implementar esta. Para além disso usa Mappers, descritos acima, para fazer pesquisas simples como verificar se o jogador e jogo com o id e nome respetivos fornecidos existem ou não. Após chamar a função abstrata, faz um varrimento para encontrar a pontuação do jogador com o respetivo id para verificar se este tem pontos suficientes para o crachá. O mapper é usado também para fazer a tarefa final que é a inserção na tabela **TEM** que representa a atribuição do crachá ao jogador.

# Controlo de concorrência

O Controlo de Concorrência Otimista e o Controlo de Concorrência Pessimista. Ambos os métodos visam resolver problemas relacionados à concorrência em sistemas multiutilizador, onde múltiplos processos ou threads tentam aceder e modificar recursos compartilhados de forma concorrente. Neste relatório, analisaremos os princípios subjacentes a cada um desses métodos, bem como suas vantagens e desvantagens.

## 4.1 Controlo de concorrência otimista

O Controlo de Concorrência Otimista baseia-se no princípio de permitir que múltiplos processos ou threads acessem recursos compartilhados sem restrições, assumindo que os conflitos de concorrência são raros. Durante a execução, cada processo ou thread realiza suas operações sem bloqueios, mas verifica se ocorreram conflitos antes de efetuar as alterações definitivas. Caso haja conflitos, o processo ou thread revertirá as alterações e repetirá a operação com base na nova informação obtida. A deteção de conflitos é geralmente realizada através de técnicas como controle de versões em que cada vez que é feita uma escrita de um processo, a versão, como um inteiro, é incrementada, assim se for feita uma escrita de outro processo no meio é necessário verificar se o valor da versão é mesma do processo anterior, caso contrário é feito um roolback.

Vantagens do Controlo de Concorrência Otimista:

- Permite um alto nível de concorrência, pois os processos ou threads não são bloqueados durante a execução.

- Evita bloqueios desnecessários, melhorando o desempenho do sistema em cenários de baixa concorrência.

- Reduz o tempo de espera e aumenta a taxa de aceitação de operações concorrentes, maximizando a utilização dos recursos compartilhados.

Desvantagens do Controlo de Concorrência Otimista:

- Pode haver um aumento no número de operações rejeitadas ou revertidas devido a conflitos de concorrência.

- A implementação correta do algoritmo de deteção de conflitos pode ser complexa e requerer uma cuidadosa consideração dos casos de uso.

- Pode ocorrer a necessidade de efetuar compensações e repetições de operações, o que pode introduzir um overhead adicional no sistema.

## 4.2 Controlo de concorrência pessimista

## O Controlo de Concorrência Pessimista adota uma abordagem mais cautelosa, assumindo que os conflitos de concorrência são frequentes e, portanto, bloqueia os processos ou threads para evitar acessos simultâneos a recursos compartilhados. Cada processo ou thread adquire um bloqueio exclusivo antes de realizar operações nos recursos, garantindo assim a exclusividade e consistência dos dados. Esse bloqueio pode ser implementado usando mecanismos como semáforos, mutexes ou transações.

## Vantagens do Controlo de Concorrência Pessimista:

## - Garante a consistência dos dados, pois apenas um processo ou thread pode modificar os recursos compartilhados por vez.

## - Evita conflitos de concorrência, eliminando a necessidade de reversões de operações e repetições.

## - É mais fácil de implementar e entender em comparação ao Controlo de Concorrência Otimista.

## Desvantagens do Controlo de Concorrência Pessimista:

## - Pode levar a um baixo nível de concorrência, uma vez que os processos ou threads são bloqueados durante a execução.

## - Pode ocorrer um aumento no tempo de espera e na contenção, especialmente em sistemas com alto volume de operações concorrentes.

## - Pode resultar em bloqueios desnecessários em cenários de baixa concorrência, impactando negativamente o desempenho do sistema.

## 4.3 Análise de resultados

## No caso de erro de concorrência, é um atraso deliberado entre a leitura e a atualização do objeto crachá “c” no banco de dados. Isso criará uma situação em que outro processo ou thread pode modificar o objeto "c" antes que a transação atual seja confirmada, resultando em uma exceção de concorrência.

## Ao executar o teste, a exceção de concorrência (OptimisticLockException) será lançada quando a transação atual tentar confirmar as alterações e perceber que o objeto foi modificado por outra fonte antes da confirmação.

# Conclusões