

**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

O projeto consistiu na implementação de várias funcionalidades num sistema utilizando a linguagem de programação Java e JPA (Java Persisntence API). O objetivo principal era criar um software robusto e flexível, aplicando princípios como modularidade, reutilização de código e facilidade na manutenção para poder aceder às funcionalidades implementadas na fase 1 deste trabalho.

Destacamos também a realização de testes de concorrência para garantir a eficácia das soluções implementadas. Utilizamos o optimistic locking e o pessimistic locking para controlar o acesso concorrente aos recursos e evitar inconsistências nos dados.

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

# Abstract

This report aims to document the implementation of functionalities in our project. During the system development, we sought to apply principles of modularity, code reuse, and ease of maintenance, aiming to create robust and flexible software.

In this report, we will present the implemented functionalities and the design patterns used for each of them. We will explore the benefits and logic behind the choice of each pattern, highlighting how they contributed to the efficiency and scalability of our system.

Furthermore, we will discuss the use of Java reflection, which played a crucial role in executing the stored routines in our database. We will describe how reflection allowed us to divide and generalize the code execution, making it easier to read and maintain.

The implementations and results of concurrency tests will also be addressed, demonstrating the effectiveness of the implemented solutions.

Through this report, we aim to provide a comprehensive overview of the strategies adopted during the implementation of functionalities, demonstrating how design patterns contributed to the quality of our system and the ease of its maintenance and evolution.

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

**Índice**

[Resumo iii](#_Toc137504635)

[Abstract iv](#_Toc137504636)

[Lista de Figuras viii](#_Toc137504637)

[1. Introdução 1](#_Toc137504638)

[2. Camada de acesso a dados 2](#_Toc137504639)

[2.1 Implementação do Mapper 2](#_Toc137504640)

[2.3 Implementação das entidades em JPA 4](#_Toc137504641)

[2.2.1 Entidades 4](#_Toc137504642)

[2.2.2 Entidades Fracas e relações de grau 1:N 5](#_Toc137504643)

[2.2.3 Relações de grau N:N 6](#_Toc137504644)

[3. Acesso às funcionalidades da BD 7](#_Toc137504645)

[3.1 Registo dos parâmetros das funcionalidades 7](#_Toc137504646)

[3.2 Execução de Rotinas armazenadas 9](#_Toc137504647)

[3.3 Implementação do Associar crachá usando/não stored procedures 10](#_Toc137504648)

[4. Controlo de concorrência 12](#_Toc137504649)

[4.1 Controlo de concorrência otimista 12](#_Toc137504650)

[4.2 Controlo de concorrência pessimista 13](#_Toc137504651)

[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. 13](#_Toc137504652)

[Vantagens do Controlo de Concorrência Pessimista: 13](#_Toc137504653)

[- Garante a consistência dos dados, pois apenas um processo ou thread pode modificar os recursos compartilhados por vez. 13](#_Toc137504654)

[- Evita conflitos de concorrência, eliminando a necessidade de reversões de operações e repetições. 13](#_Toc137504655)

[- É mais fácil de implementar e entender em comparação ao Controlo de Concorrência Otimista. 13](#_Toc137504656)

[Desvantagens do Controlo de Concorrência Pessimista: 13](#_Toc137504657)

[- Pode levar a um baixo nível de concorrência, uma vez que os processos ou threads são bloqueados durante a execução. 13](#_Toc137504658)

[- Pode ocorrer um aumento no tempo de espera e na contenção, especialmente em sistemas com alto volume de operações concorrentes. 13](#_Toc137504659)

[- Pode resultar em bloqueios desnecessários em cenários de baixa concorrência, impactando negativamente o desempenho do sistema. 13](#_Toc137504660)

[4.3 Análise de resultados 13](#_Toc137504661)

[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. 13](#_Toc137504662)

[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. 14](#_Toc137504663)

[5. Avisos de pre-execução 15](#_Toc137504664)

[6. Conclusões 15](#_Toc137504665)

# 

# Lista de Figuras

[Figura 1 - Interface Mapper 2](file:///D:\SI\Trab1\LEIC-42D-SI\docs\Relatorio.docx#_Toc137504369)

[Figura 2 - Implementação da class Mapper 3](file:///D:\SI\Trab1\LEIC-42D-SI\docs\Relatorio.docx#_Toc137504370)

[Figura 3 - Entidade Jogo JPA. 4](file:///D:\SI\Trab1\LEIC-42D-SI\docs\Relatorio.docx#_Toc137504371)

[Figura 4 - Entidade Fraca Partida JPA 5](file:///D:\SI\Trab1\LEIC-42D-SI\docs\Relatorio.docx#_Toc137504372)

[figura 5 - Relação de grau N:N JPA 6](file:///D:\SI\Trab1\LEIC-42D-SI\docs\Relatorio.docx#_Toc137504373)

[figura 6 - UML do RoutineRegisters.java 7](file:///D:\SI\Trab1\LEIC-42D-SI\docs\Relatorio.docx#_Toc137504374)

[figura 7 - UML do RoutineControllers.java 8](file:///D:\SI\Trab1\LEIC-42D-SI\docs\Relatorio.docx#_Toc137504375)

[figura 8 - Esquema da função callRoutine 9](#_Toc137504376)

[figura 9 - Esquema das anotações 10](#_Toc137504377)

[figura 10 - UML do AssociarCracha.java 11](#_Toc137504378)

# Introdução

Este relatório tem como objetivo documentar a implementação das funcionalidades no nosso projeto. Durante o desenvolvimento do sistema, procuramos aplicar os princípios de modularidade, reutilização de código e facilidade na manutenção, visando a criação de um software robusto e flexível.

Neste relatório, apresentaremos as funcionalidades implementadas e os padrões de design utilizados para cada uma delas. Exploraremos os benefícios e a lógica por trás da escolha de cada padrão, destacando como eles contribuíram para a eficiência e a escalabilidade do nosso sistema.

Além disso, abordaremos a utilização da reflexão do Java, que desempenhou um papel fundamental na execução das rotinas armazenadas na nossa base de dados. Descreveremos como a reflexão nos permitiu dividir e generalizar a execução do código, facilitando sua leitura e manutenção.

Também serão discutidos as implementações e resultados dos testes de concorrência, demonstrando a eficácia das soluções implementadas.

Com este relatório, esperamos fornecer uma visão abrangente das estratégias adotadas durante a implementação das funcionalidades, demonstrando como os padrões de design contribuíram para a qualidade do nosso sistema e para a facilidade de sua manutenção e evolução.

# 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.

## 2.3 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

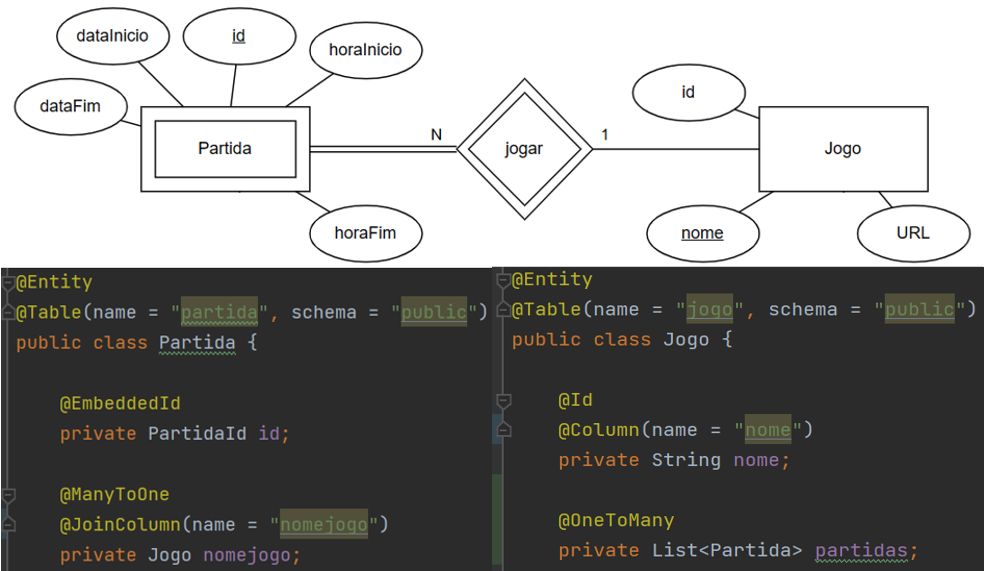
As entidades fracas do modelo EA quando convertidas para JPA, tem de ser indicado a usa dependência através das anotações do JPA. Visto que as relações de grau 1:N e entidades fracas seguem as mesmas estratégias de conversão, serão ambas abordadas nesta secção.

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

figura 5 - Relação de grau N:N JPA

Na figura 5, é possível observar que as entidades Jogador e Jogo possuem um campo com a anotação @ManyToMany, indicando uma relação de N para N entre elas. Esse campo possui o tipo da entidade com a qual compartilha essa relação.

Na classe que representa a relação N para N, é necessário adicionar a anotação @ManyToOne no campo que faz referência às entidades que utilizam essa relação. Portanto, a classe Comprar terá dois campos, um que referencia a entidade Jogador e outro que referencia a entidade Jogo. Ambos os campos terão as anotações @ManyToOne e @JoinColumn para indicar qual coluna está sendo referenciada.

# 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. Registo 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 registo (por exemplo, as funções AssociarCrachaBaseline e AssociarCrachaJPA).

figura 6 - UML do RoutineRegisters.java

2. Criação da função de chamada: Após o registo 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.

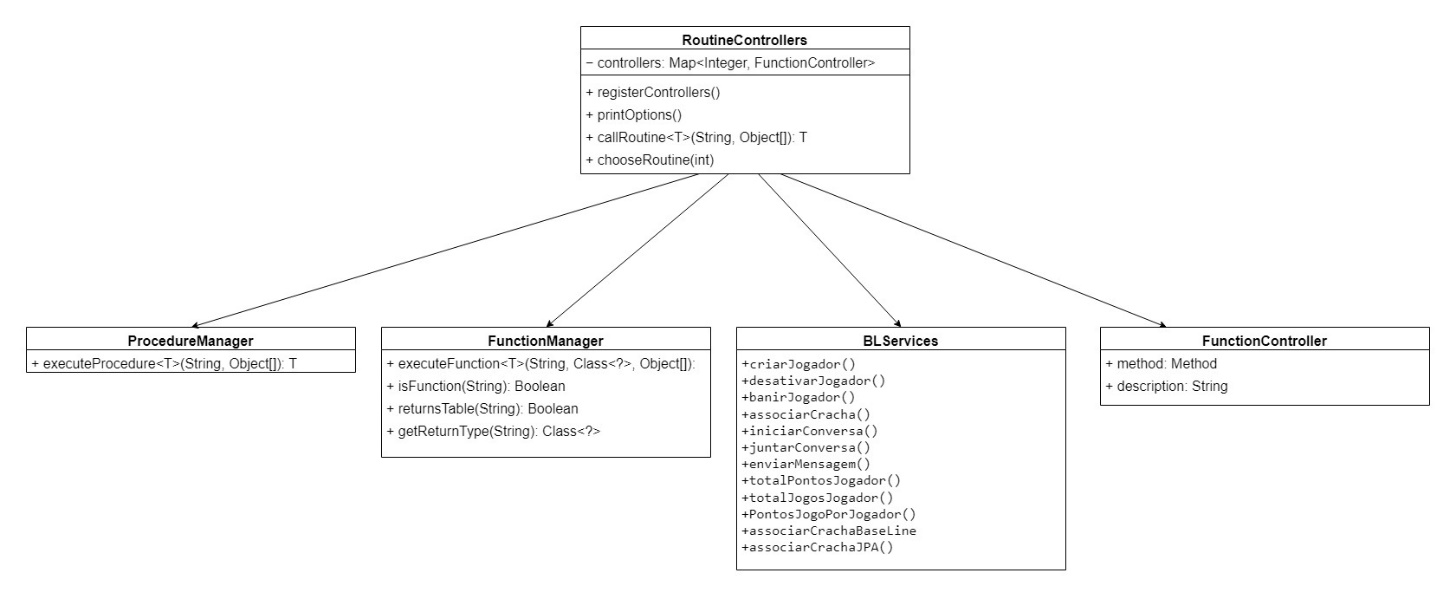


figura 7 - UML do RoutineControllers.java

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:

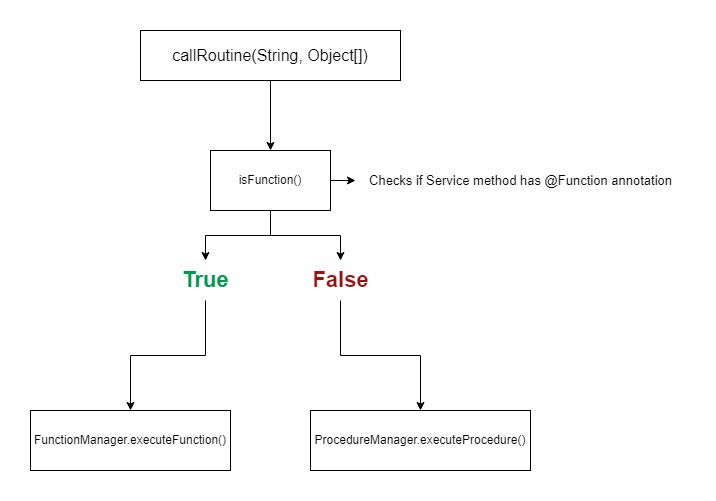


figura 8 - Esquema da função callRoutine

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:

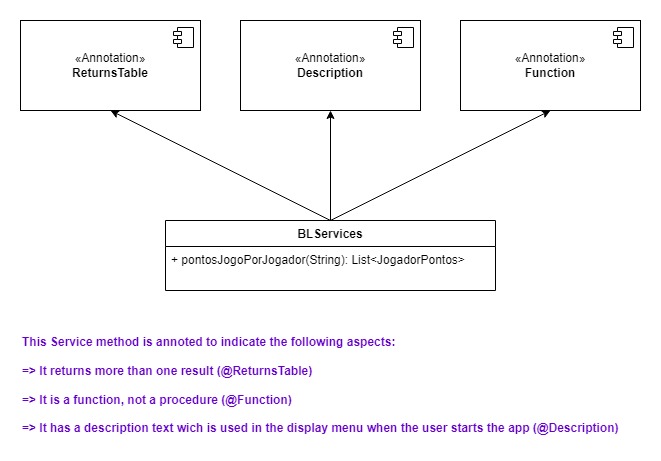


figura 9 - Esquema das anotações

## 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.

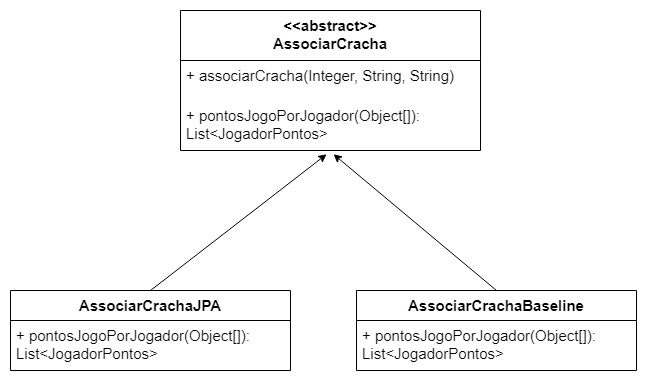


figura 10 - UML do AssociarCracha.java

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.

# Avisos de pre-execução

Para executar o projeto deve fazer as seguintes alterações:

1. Comentar as definições dos transaction level dos procedimentos armazenados.
2. Compilar com o seguinte comando: -parameters para que durante a execução, quando lhe for pedido os argumentos de cada função, através de reflexão, ser apresentado o nome do parâmetro original.

# Conclusões

Ao concluir este trabalho, foi possível alcançar os objetivos propostos de implementar funcionalidades específicas em um projeto Java. Durante o desenvolvimento, aplicamos princípios de modularidade, reutilização de código e facilidade na manutenção para criar um software robusto e flexível.

Ao utilizar algumas técnicas como reflexão e genéricos, conseguimos estruturar o código de forma organizada, facilitando sua compreensão e manutenção. Através de classes abstratas, evitamos repetição de código e permitimos a criação de objetos de forma mais flexível. Já a Reflexção permitiu-nos acessar e executar funções e procedimentos armazenados na base de dados de maneira genérica e escalável.

Em resumo, este trabalho nos permitiu aprofundar nossos conhecimentos em programação Java, mais concreatamente a usar a JPA (Java Persistance API) e técnicas de mapeamento de entidades. Através da aplicação desses conceitos, conseguimos desenvolver um software com funcionalidades robustas, escaláveis e de fácil manutenção.