Relatório de Arquitetura - Sistema de Reserva de Espaços Físicos em uma Universidade

Disciplina: Orientação a Objetos
Professor: Andre Luiz Peron Martins Lanna
Alunos: Pedro Ramos Sousa Reis (222031680), Rafael Rodrigues Alencar (222026528), Tiago Sousa Nepomuceno (222009732), Isadora Quaresma Oliveira (222025155)

Este relatório detalha as decisões de arquitetura, os padrões de Orientação a Objetos e as estratégias de tratamento de erros aplicadas no desenvolvimento do Sistema de reserva de espaços físicos em uma Universidade.

1. Arquitetura e Padrões de Orientação a Objetos

O sistema foi estruturado utilizando conceitos fundamentais de Programação Orientada a Objetos (POO) para garantir um código limpo, reutilizável e de fácil manutenção. Os principais padrões aplicados foram Herança, Polimorfismo e Associações.

1.1. Herança

A herança foi utilizada para criar hierarquias de classes, promovendo o reuso de código e estabelecendo relações do tipo "é um".

Hierarquia de `Usuario`

A estrutura de usuários é um exemplo de herança.

- **`Usuario` (Classe Abstrata):** Funciona como a superclasse para todos os tipos de usuários. Ela define os atributos e comportamentos comuns, como `nome`, `email`, `telefone` e `senha`. O método `getTipoUsuario()` foi declarado como `abstract` para forçar as subclasses a implementarem sua própria identificação.
- **`Aluno`, `Professor` e `ServidorAdministrativo` (Classes Concretas):** Herdam de `Usuario` e adicionam atributos e comportamentos específicos.
    - `Aluno` **é um** `Usuario` que possui `matricula`, `curso` e `semestre`.
    - `Professor` **é um** `Usuario` que possui `matriculaInstitucional` e `cargo`.
    - `ServidorAdministrativo` **é um** `Usuario` que possui `cargoAdministrativo` e `departamento`.

Hierarquia de `EspacoFisico`

De forma similar, a herança foi aplicada aos espaços físicos.

- **`EspacoFisico` (Classe Abstrata):** Define os atributos comuns a todos os espaços, como `nome`, `capacidade`, `localizacao` e `equipamentos`.
- **`SalaAula`, `Laboratorio` e `Auditorio` (Classes Concretas):** Herdam de `EspacoFisico`, representando tipos específicos de espaços que podem ser agendados.

1.2. Polimorfismo

- **Exemplo no Cadastro:** No método `cadastrarUsuario()`, um objeto `Usuario` é instanciado. Dependendo da escolha do usuário, este objeto pode ser um `new Aluno(...)`, `new Professor(...)` ou `new ServidorAdministrativo(...)`. Independentemente do tipo real, a variável é do tipo `Usuario`. Isso permite que o método `cadastroUsuario.cadastrarUsuario(usuario)` seja chamado, passando um objeto que pode assumir "várias formas" (polimorfismo).
- **Exemplo na Busca:** O método `buscarPorMatricula` no `GerenciadorUsuarios` percorre uma `List<Usuario>`. Dentro do loop, ele utiliza o operador `instanceof` para verificar a forma real do objeto (`Aluno` ou `Professor`) e executar a lógica de comparação de matrícula apropriada para cada tipo.
- **Exemplo com Métodos Abstratos:** O método `getTipoUsuario()` na classe `Usuario` é abstrato. Cada subclasse (Aluno, Professor, etc.) fornece sua própria implementação. Isso permite chamar `usuario.getTipoUsuario()` em qualquer objeto do tipo `Usuario` e obter a resposta correta ("Aluno", "Professor", etc.), o que é usado para exibir os detalhes na busca.

1.3. Associações

As associações definem como os objetos de diferentes classes se relacionam e interagem.

- **`Agendamento` como Classe de Associação:** A classe `Agendamento` é o exemplo mais forte de associação. Sua função é conectar um objeto `Usuario` a um objeto `EspacoFisico` em um determinado período de tempo (`LocalDateTime`). Ela representa a relação "reserva" entre as duas entidades principais.
- **Composição em `EspacoFisico`:** A classe `EspacoFisico` possui uma lista de `equipamentos`. Isso representa uma relação de composição, onde um espaço físico "tem" equipamentos.
- **Injeção de Dependência nos Serviços:** As classes de serviço (como `GerenciadorUsuarios` e `GeradorRelatorios`) recebem as listas (`List<Usuario>`, `List<Agendamento>`) em seus construtores. Isso é uma forma de associação conhecida como injeção de dependência. Em vez de o serviço criar sua própria lista, ele depende de uma lista externa, permitindo que múltiplos serviços compartilhem e operem sobre o mesmo conjunto de dados.

2. Justificativa para Exceções Customizadas

- **`CampoInvalidoException` (e suas subclasses `EmailInvalidoException`, `SenhaInvalidaException`):**
    - **Justificativa:** Lançada durante a validação de dados de entrada. Permite que o serviço de cadastro informe à interface exatamente qual campo está com problema (ex: nome vazio, formato de email incorreto) e por quê. Isso possibilita a exibição de mensagens de erro específicas e úteis para o usuário final.
- **`UsuarioNaoEncontradoException`:**
    - **Justificativa:** Essencial para operações de busca e deleção. Quando um usuário tenta fazer um agendamento ou deletar um perfil com uma matrícula ou email que não existe, esta exceção é lançada. Ela separa claramente a falha "não encontrado" de outros possíveis erros (como problemas de conexão, que não se aplicam aqui).
- **`UsuarioJaCadastradoException`:**
    - **Justificativa:** Garante a integridade dos dados. Impede que dois usuários sejam cadastrados com o mesmo email, que é um identificador único.
- **`HorarioIndisponivelException`:**
    - **Justificativa:** Implementa uma das principais regras de negócio do sistema. É lançada pelo `GerenciadorAgendamento` quando há um conflito de horários. O nome da exceção é autoexplicativo, informando imediatamente à camada de interface que a falha ocorreu por uma sobreposição de reservas.
- **`DiasExcedidosException`:**
    - **Justificativa:** Implementa a segunda regra de negócio obrigatória. Isola a lógica de verificação de que alunos não podem reservar por mais de 24 horas. Se a regra for violada, esta exceção é lançada, permitindo que a `Principal` mostre uma mensagem específica sobre esta política do sistema.