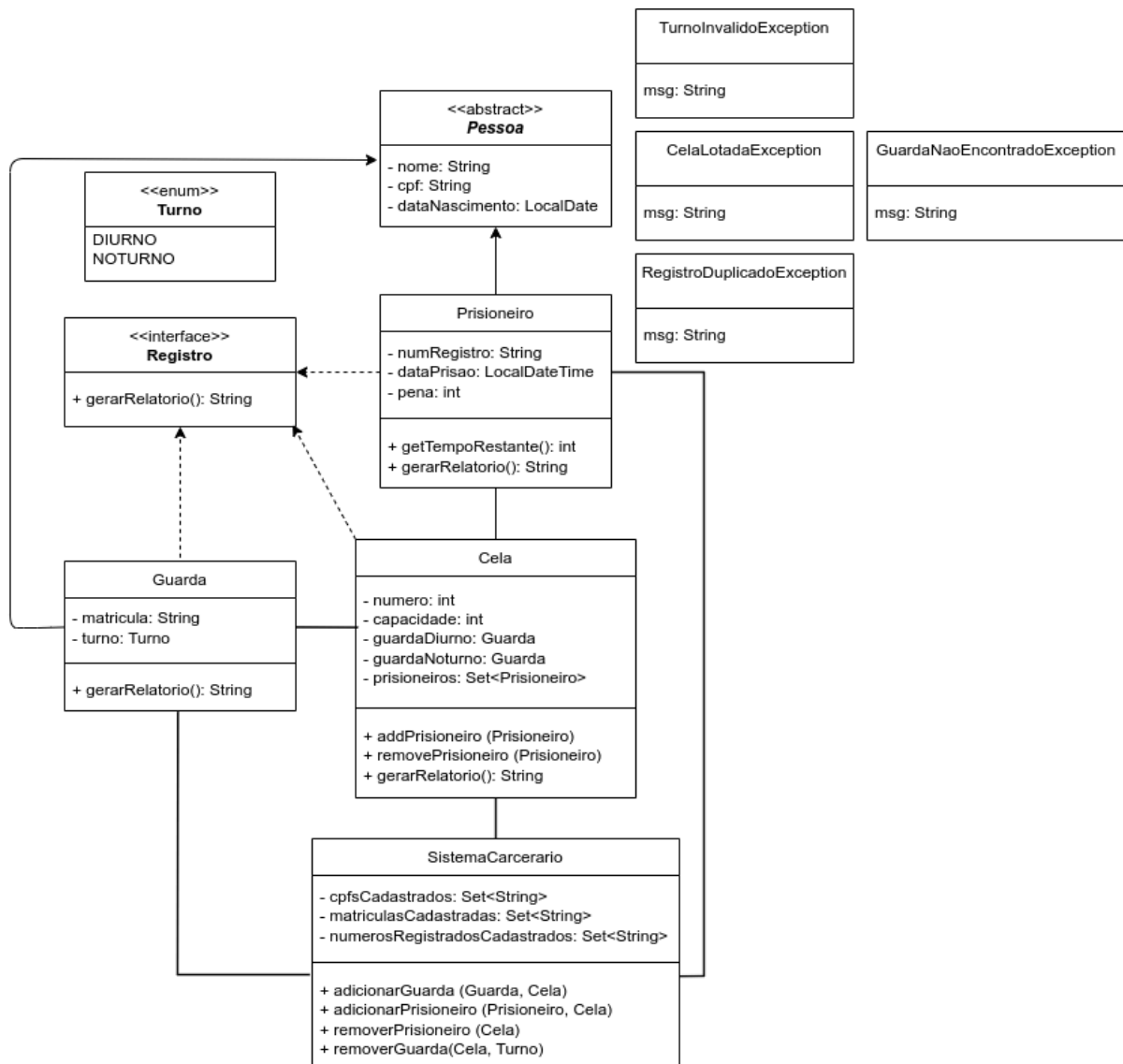


Aluna: Lais dos Anjos Varela.

Matrícula: 202210507943

ATIVIDADE AVALIATIVA I - Desenvolvimento de Sistema com Aplicação de Conceitos de POO

Para esta atividade, decidi fazer um sistema carcerário, tentando aplicar da melhor forma os requisitos da avaliação. Utilizei o Visual Studio Code (VSCode) como ferramenta para realização do código e do diagrama UML. Aproveitei as extensões de suporte tanto para a codificação em Java quanto para a criação do diagrama UML, por meio da ferramenta Draw.io integrada. Segue o diagrama UML:



Exceções Personalizadas

- `TurnoInvalidoException`: Lançada quando há uma tentativa de cadastrar um guarda em um turno já ocupado na cela.
- `CelaLotadaException`: Indicada quando se tenta adicionar um prisioneiro a uma cela que já atingiu sua capacidade máxima.
- `RegistroInvalidoException`: Utilizada para sinalizar registros duplicados de valores como CPF, matrícula e número de registro.
- `GuardaNaoEncontradoException`: Lançada quando se tenta remover um guarda de um turno que está vazio.

Classes

Como forma de atender ao requisito de classe abstrata, criei a classe `Pessoa`, que representa características comuns a qualquer indivíduo. As classes `Guarda` e `Prisioneiro` herdam dessa abstração, reutilizando atributos essenciais como nome, CPF e data de nascimento, informações básicas que toda pessoa deve possuir. A classe `Cela` contém os métodos para adicionar e remover prisioneiros, mas a lógica de validação e controle está centralizada na classe `SistemaCarcerario`. É nessa classe que a lógica é implementada, incluindo o lançamento das exceções personalizadas. Para garantir que identificadores como CPF, matrícula e número de registro não sejam duplicados, a classe utiliza conjuntos `Set` do tipo `String`.

```
public class SistemaCarcerario {  
    private Set<String> cpfsCadastrados = new HashSet<>();  
    private Set<String> matriculasCadastradas = new HashSet<>();  
    private Set<String> numerosRegistroCadastrados = new HashSet<>();  
}
```

Nessa classe há quatro métodos:

- **adicionarGuarda**: Este método adiciona um guarda a uma cela, verificando antes se ele já está registrado e se o turno é válido.
 - Estruturas condicionais para `RegistroInvalidoException`:

```
if (matriculasCadastradas.contains(guarda.getMatricula())) {  
    throw new RegistroDuplicadoException("Matrícula já cadastrada: " + guarda.getMatricula());  
}  
if (cpfsCadastrados.contains(guarda.getCpf())) {  
    throw new RegistroDuplicadoException("CPF já cadastrado: " + guarda.getCpf());  
}
```

Se o objeto *guarda* passado por parâmetro for igual a algum item contido no conjunto *matriculasCadastradas* ou *cpfsCadastrados*, a exceção é lançada. Caso contrário, o objeto é adicionado no conjunto.

```
matriculasCadastradas.add(guarda.getMatricula());  
cpfsCadastrados.add(guarda.getCpf());
```

- Estruturas condicionais para *TurnoInvalidoException*:

```
if (cela.getGuardaDiurno() == null && guarda.getTurno() == Turno.DIURNO) {  
    cela.setGuardaDiurno(guarda);  
}  
else if (cela.getGuardaNoturno() == null && guarda.getTurno() == Turno.NOTURNO) {  
    cela.setGuardaNoturno(guarda);  
}  
else if (cela.getGuardaDiurno() != null && cela.getGuardaDiurno().getTurno() == guarda.getTurno()  
|| cela.getGuardaNoturno() != null && cela.getGuardaNoturno().getTurno() == guarda.getTurno()) {  
    throw new TurnoInvalidoException("Já existe guarda cadastrada para o turno " + guarda.getTurno());  
}
```

Primeiro, é atribuído o objeto *guarda* ao turno equivalente, caso esteja null.

Se ambos os turnos estiverem preenchidos, cai no else if que lança a exceção indicando que a atribuição é inválida, pois o turno já está preenchido.

- **adicionarPrisoneiro**: Esse método serve para adicionar um prisioneiro a uma cela, verificando antes se ele já está registrado e se a cela tem espaço disponível. Se alguma das condições não for respeitada, lança uma exceção personalizada. *RegistroDuplicadoException*, se o prisioneiro já estiver cadastrado (pelo número de registro ou CPF). *CelaLotadaException*, se a cela já estiver cheia. Caso contrário, os prisioneiros são adicionados à lista de prisioneiros da classe *Cela*, através do método *addPrisoneiro*. Também é adicionado aos conjuntos.

```
public void adicionarPrisoneiro(Prisoneiro prisoneiro, Cela cela) throws RegistroDuplicadoException, CelaLotadaException {  
    if (numerosRegistroCadastrados.contains(prisoneiro.getNumRegistro())) {  
        throw new RegistroDuplicadoException("Número de registro já cadastrado: " + prisoneiro.getNumRegistro());  
    }  
    if (cpfsCadastrados.contains(prisoneiro.getCpf())) {  
        throw new RegistroDuplicadoException("CPF já cadastrado: " + prisoneiro.getCpf());  
    }  
    if (cela.getPrisoneiros().size() >= cela.getCapacidade()) {  
        throw new CelaLotadaException(message:"Cela lotada, não é possível adicionar mais prisioneiros");  
    } else {  
        cela.addPrisoneiro(prisoneiro);  
    }  
    numerosRegistroCadastrados.add(prisoneiro.getNumRegistro());  
    cpfsCadastrados.add(prisoneiro.getCpf());  
}
```

- **removerGuarda**: Método responsável por lançar a exceção *GuardaNaoEncontradoException*. Verifica se existe um guarda naquele turno, se sim, lança a exceção, caso contrário, remove o guarda atual.

```

public void removerGuarda(Cela cela, Turno turno) throws GuardaNaoEncontradoException{
    if (turno == Turno.DIURNO && cela.getGuardaDiurno() == null) {
        throw new GuardaNaoEncontradoException("Não há guarda diurno cadastrado para ser removido na cela " + cela.getNumero());
    } else if (turno == Turno.NOTURNO && cela.getGuardaNoturno() == null) {
        throw new GuardaNaoEncontradoException("Não há guarda noturno cadastrado para ser removido na cela " + cela.getNumero());
    }
    cela.setGuardaDiurno(guardaDiurno:null);
    cela.setGuardaNoturno(guardaNoturno:null);
}

```

- **removerPrisioneiro:** Esse método serve para remover prisioneiros de uma cela quando o tempo restante da pena deles for menor ou igual a zero (ou seja, quando eles já cumprirem a pena). Por ser um conjunto Set, utilizei um Iterator para percorrer e remover de forma segura. Enquanto a lista tiver o próximo item, verifica o tempo restante.

```

public void removerPrisioneiro(Cela cela) {
    Iterator<Prisioneiro> iterator = cela.getPrisioneiros().iterator();
    while (iterator.hasNext()) {
        Prisioneiro prisioneiro = iterator.next();
        if (prisioneiro.getTempoRestante() <= 0) {
            iterator.remove();
        }
    }
}

```

Interface

Como interface, desenvolvi a interface Registro, que define o método gerarRelatorio. Esse método não recebe parâmetros e retorna uma String com informações relevantes, variando conforme a classe que o implementa. As classes Guarda, Prisioneiro e Cela implementam essa interface. No caso da classe Cela, o relatório gerado também inclui as informações dos prisioneiros e guardas associados, utilizando as implementações de gerarRelatorio dessas classes.

```

@Override
public String gerarRelatorio() {
    String relatorio = "\nCela nº " + this.numero + "\nCapacidade: " +
        this.capacidade + "\n-----\nGuarda diurno: \n" + getGuardaDiurno().gerarRelatorio()
        + "\n-----\nGuarda noturno: \n" + getGuardaNoturno().gerarRelatorio() + "\n-----\nPrisioneiros: \n";
    if (prisioneiros.isEmpty()) {
        relatorio = "Nenhum prisioneiro na cela.";
    }
    for (Prisioneiro prisioneiro : prisioneiros) {
        relatorio += prisioneiro.gerarRelatorio() + "Tempo Restante: " + prisioneiro.getTempoRestante() + "\n";
    }
    return relatorio;
}

```

Conclusão

O desenvolvimento do sistema carcerário permitiu aplicar de forma prática os principais conceitos da Programação Orientada a Objetos e os demais requisitos da avaliação, melhorando meu entendimento do conteúdo.