

# Documento de Decisões Arquiteturais - Grupo 1

Alunos: Gabriel Freitas, Gabriel Rodrigues, Laura Martins, Léia Santos e Tallya Barbosa  
Documento desenvolvido com contribuição de todos os integrantes, em especial Tallya e Léia.

Em discussão, o grupo optou por detalhar o processo de remodelagem do sistema do padrão MVC para o padrão DDD, devido à escolha de tópico de inovação realizada em sala. A seguir, está o passo a passo a ser aplicado em cada camada para realizar essa remodelação, antes de seguir com a implementação das funcionalidades do projeto.

## Modificações Necessárias para Remodelar o Projeto de MVC para DDD

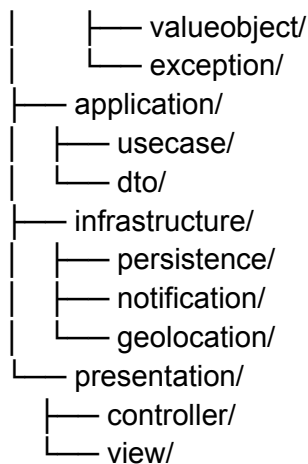
### 1. Reestruturação de Pacotes e Camadas

#### Estrutura Atual (MVC):

```
src/main/java/br/com/appvagasvan/  
├── controller/  
├── model/  
├── services/  
└── view/
```

#### Nova Estrutura (DDD):

```
src/main/java/br/com/appvagasvan/  
├── domain/  
│   ├── passageiro/  
│   │   ├── entity/  
│   │   ├── valueobject/  
│   │   ├── repository/  
│   │   └── service/  
│   ├── turno/  
│   │   ├── entity/  
│   │   ├── valueobject/  
│   │   ├── repository/  
│   │   └── service/  
│   ├── viagem/  
│   │   ├── entity/  
│   │   ├── valueobject/  
│   │   ├── repository/  
│   │   └── service/  
│   └── shared/  
└──
```



## 2. Transformações nas Classes de Modelo

### 2.1. Passageiro - De Model para Entidade de Domínio

#### Modificações necessárias:

- Remover dependências do JavaFX (Properties)
- Adicionar método construtor privado e factory method
- Implementar validações de negócio no construtor
- Criar Value Object: Endereco
- Adicionar métodos de domínio: confirmarPresenca(), cancelarPresenca()
- Implementar lógica de validação: nome não pode ser vazio, endereço válido
- Adicionar eventos de domínio: PassageiroConfirmouPresencaEvent

#### Exemplo de transformação:

Atual: Passageiro com Properties do JavaFX

Novo: Passageiro como Aggregate Root com:

- PassageiroId (integer)
- Nome (string)
- Endereco (Value Object)
- StatusConfirmacao (boolean)
- Métodos: confirmar(), cancelar(), validarEndereco()

### 2.2. Turno - De Model para Aggregate Root

#### Modificações necessárias:

- Remover Properties do JavaFX
- Criar atributos: TurnoId, NomeTurno, Horario, CapacidadeVagas, IdMotorista
- Implementar invariantes: capacidade > 0, não permitir mais confirmações que vagas

- Adicionar métodos de negócio: adicionarPassageiro(), removerPassageiro(), confirmarParticipacao()
- Validar regras: RN1 (só confirmar se associado), RN2 (respeitar horário limite)
- Criar eventos: PassageiroAdicionadoAoTurnoEvent, CapacidadeEsgotadaEvent
- Encapsular coleções: expor apenas listas imutáveis

## 2.3. Motorista - De Model para Entidade

### Modificações necessárias:

- Transformar em entidade dentro do contexto de Viagem
- Criar Value Objects: EnderecoOrigem
- Remover referência direta a turnos (usar repositório)
- Adicionar validações específicas

## 2.4. SimulacaoCorrida - De Model para Entidade de Domínio

### Modificações necessárias:

- Criar atributos: Distancia, TempoEstimado, OrdemColeta
- Adicionar lógica de otimização no domínio (ou service)
- Implementar validações: distância  $\geq 0$ , tempo  $\geq 0$

# 3. Criação de Value Objects

## 3.1. Atributos e Value Objects a Criar:

### Passageiro:

- PassageiroId - identificador único
- Nome - validação: não vazio, tamanho entre 3-100 caracteres
- Endereco - validação: formato válido, não vazio
- Telefone - validação: formato brasileiro
- DeviceToken - para notificações

### Turno:

- TurnoId - identificador único
- NomeTurno - validação: não vazio
- Horario - validação: formato HH:mm válido
- CapacidadeVagas - validação: valor positivo
- HorarioLembrete - validação: anterior ao horário do turno
- IdMotorista - validação: motorista cadastrado no sistema

### **Viagem/Simulação:**

- Distancia - em km, valor não negativo
- TempoEstimado - em minutos, valor não negativo
- OrdemColeta - lista ordenada de PassageiroId

### **Compartilhados:**

- Email - validação de formato
- Coordenadas - latitude/longitude para geolocalização

## **4. Criação de Repositórios (Interfaces)**

### **4.1. Repositórios no Domínio (Interfaces):**

domain/passageiro/repository/PassageiroRepository

- save(Passageiro)
- findById(PassageiroId)
- findAll()
- delete(PassageiroId)
- findByTurno(TurnoId)

domain/turno/repository/TurnoRepository

- save(Turno)
- findById(TurnoId)
- findByMotorista(MotoristaId)
- delete(TurnoId)
- findAll()

domain/viagem/repository/SimulacaoCorridaRepository

- save(Viagem)
- findByTurno(TurnoId)
- findSimulacoesRecentes()

### **4.2. Implementações na Infraestrutura:**

infrastructure/persistence/

- PassageiroRepository (inicialmente)
- TurnoRepository
- SimulacaoCorridaRepository

## **5. Transformação de Services**

### **5.1. GerenciadorVans - Dividir em Domain Services**

### **Criar Domain Services:**

domain/turno/service/GerenciadorTurno

- criarTurno()
- validarCapacidade()
- verificarDisponibilidade()

domain/passageiro/service/GerenciadorConfirmacao

- confirmarParticipacao()
- cancelarConfirmacao()
- validarAssociacao()

domain/viagem/service/OtimizadorRota

- calcularRotaOtimizada()
- simularCorrida()
- estimarTempoDistancia()

domain/shared/service/ValidadorRegrasTurno

- validarHorarioLimite()
- validarCapacidadeMaxima()

## **5.2. Remover CRUD Genérico**

- Eliminar interface GerenciadorCRUD
- Cada repositório terá métodos específicos ao seu contexto
- Operações complexas ficam em Application Services

# **6. Criação da Camada de Aplicação (Use Cases)**

## **6.1. Use Cases a Implementar:**

application/usecase/passageiro/

- ConfirmarParticipacaoUseCase
- CancelarParticipacaoUseCase

application/usecase/turno/

- CriarTurnoUseCase
- EditarTurnoUseCase
- DeletarTurnoUseCase
- VisualizarPassageirosConfirmadosUseCase

application/usecase/viagem/

- SimularCorridaUseCase
- VisualizarOrdemOtimizadaUseCase
- AdicionarPassageiroAoTurnoUseCase
- RemoverPassageiroDoTurnoUseCase

application/usecase/notificacao/  
- EnviarLembreteConfirmacaoUseCase

## 6.2. Estrutura de um Use Case:

Cada Use Case deve:

- Receber um DTO de entrada
- Coordenar repositórios e domain services
- Retornar um DTO de saída
- Tratar exceções de domínio
- Publicar eventos de domínio se necessário

## 7. Criação de DTOs

### 7.1. DTOs de Input:

application/dto/  
- ConfirmarParticipacaoInput  
- CriarTurnoInput  
- SimularCorridaInput  
- AdicionarPassageiroInput

### 7.2. DTOs de Output:

application/dto/  
- PassageiroOutput  
- TurnoOutput  
- SimulacaoCorridaOutput  
- ListaPassageirosConfirmadosOutput

## 8. Camada de Infraestrutura

### 8.1. Persistência:

- Implementar repositórios in-memory inicialmente
- Preparar para migração futura para JPA/Hibernate
- Implementar conversores: Entidade → Modelo de Persistência

### 8.2. Serviços Externos:

infrastructure/notification/NotificationService  
- Implementação do envio de notificações push  
- Integração com Firebase ou similar

infrastructure/geolocation/GeolocationService

- Integração com Google Maps API
- Cálculo de rotas e distâncias

## 9. Adaptação dos Controllers

### 9.1. Modificações no DriverDashboardController:

- Remover acesso direto ao GerenciadorVans
- Injetar Use Cases via construtor
- Converter dados da view para DTOs
- Chamar Use Cases e processar DTOs de saída
- Tratar exceções de domínio e apresentar feedback

#### Exemplo:

Antes:

```
GerenciadorVans.getInstance().confirmarParticipacao(...)
```

Depois:

```
ConfirmarParticipacaoInput input = new ConfirmarParticipacaoInput(...)
ConfirmarParticipacaoOutput output = confirmarParticipacaoUseCase.execute(input)
atualizarView(output)
```

## 10. Implementação de Eventos de Domínio

### 10.1. Eventos a Criar:

domain/passageiro/event/

- PassageiroConfirmouPresencaEvent
- PassageiroCancelouPresencaEvent

domain/turno/event/

- TurnoCriadoEvent
- PassageiroAdicionadoAoTurnoEvent
- CapacidadeEsgotadaEvent
- TurnoRemovidoEvent

domain/viagem/event/

- RotaOtimizadaCalculadaEvent
- SimulacaoConcluidaEvent

### 10.2. Event Handler:

application/event/

- EventPublisher
- EventHandler (para cada evento)

Exemplo:

PassageiroConfirmouPresencaEventHandler

- Atualizar estatísticas
- Notificar motorista
- Registrar log

## 11. Tratamento de Exceções

### 11.1. Exceções de Domínio:

domain/shared/exception/

- DomainException (base)
- PassageiroNaoAssociadoException
- CapacidadeEsgotadaException
- HorarioLimiteExcedidoException
- TurnoComPassageirosException
- EnderecoInvalidoException

### 11.2. Uso:

- Lançar exceções específicas nas entidades e value objects
- Capturar nos Use Cases
- Converter para mensagens amigáveis nos controllers

## 12. Regras de Negócio no Domínio

### 12.1. Mover Validações para Entidades:

No Turno:

- Capacidade deve ser > 0 (no construtor/setter)
- Não permitir confirmações além da capacidade
- Não deletar turno com passageiros associados
- Validar horário de lembrete < horário do turno

No Passageiro:

- Nome não vazio, tamanho válido
- Endereço não vazio, formato válido
- Só pode confirmar em turno associado
- Não pode confirmar duas vezes o mesmo turno

### **Na Viagem/Simulação:**

- Só simular com passageiros confirmados
- Distância e tempo não negativos
- Ordem de coleta respeitando passageiros confirmados

## **13. Separação de Responsabilidades**

### **13.1. Domain Layer:**

- Apenas lógica de negócio pura
- Sem dependências de frameworks (exceto Java puro)
- Entidades, Value Objects, Domain Services
- Interfaces de repositórios

### **13.2. Application Layer:**

- Orquestração de Use Cases
- Coordenação entre repositórios e services
- Transações (futuramente)
- Conversão entre domínio e DTOs

### **13.3. Infrastructure Layer:**

- Implementação de repositórios
- Integração com serviços externos
- Detalhes técnicos de persistência
- Configurações de frameworks

### **13.4. Presentation Layer:**

- Controllers JavaFX
- Conversão entre DTOs e modelos de view
- Validações de UI
- Binding com componentes visuais

## **14. Eliminação do Singleton**

### **14.1. Problema Atual:**

- `GerenciadorVans.getInstance()` é um singleton
- Dificulta testes e viola princípios DDD

### **14.2. Solução:**

- Usar injeção de dependência

- Criar factory ou container simples
- Controllers recebem Use Cases via construtor
- Use Cases recebem repositórios via construtor

## 15. Agregados e Consistência

### 15.1. Definir Agregados:

#### Agregado Turno (root):

- Turno (root)
- Lista de associações Passageiro-Turno
- Confirmações

#### Agregado Passageiro (root):

- Passageiro (root)
- Dados pessoais

#### Agregado Viagem (root):

- Viagem/Simulação (root)
- Rota otimizada
- Referências para Turnold e Passageirolds

### 15.2. Regras de Acesso:

- Sempre acessar agregado pela raiz
- Não manter referências entre agregados, usar IDs
- Garantir consistência dentro do agregado

## 16. Testes

### 16.1. Reorganizar Testes:

#### Domain Layer:

test/domain/passageiro/PassageiroTest  
test/domain/turno/TurnoTest  
test/domain/valueobject/NomeTest  
test/domain/valueobject/EnderecoTest

#### Application Layer:

test/application/usecase/ConfirmarParticipacaoUseCaseTest  
test/application/usecase/SimularCorridaUseCaseTest

## Infrastructure Layer:

test/infrastructure/persistence/PassageiroRepositoryInMemoryTest

### 16.2. Adaptar GerenciadorVansTest:

- Dividir em testes unitários de domínio
- Criar testes de Use Cases separados
- Mockar repositórios nos testes de aplicação

## Resumo das Principais Transformações

1. **Pacotes:** Reorganizar de MVC (model/view/controller) para DDD (domain/application/infrastructure/presentation)
2. **Models** → **Entidades:** Remover JavaFX Properties, adicionar validações e comportamentos de negócio
3. **Services** → **Domain Services + Use Cases:** Separar lógica de domínio (domain services) de orquestração (use cases)
4. **Criar Value Objects:** Extrair conceitos como Nome, Endereco, Horário para objetos imutáveis
5. **Repositórios:** Criar interfaces no domínio e implementações na infraestrutura
6. **DTOs:** Criar objetos de transferência para comunicação entre camadas
7. **Eventos:** Implementar eventos de domínio para ações importantes
8. **Exceções:** Criar hierarquia de exceções de domínio
9. **Controllers:** Adaptar para usar Use Cases em vez de acessar serviços diretamente
10. **Eliminação de Singleton:** Usar injeção de dependência

Esta remodelação permitirá um código mais manutenível, testável e alinhado com os princípios de Domain-Driven Design, mantendo a lógica de negócio isolada e protegida de detalhes técnicos.