Descrição Arquitetural do Sistema EasyStop

Controle de Versões da Documentação	
Nome	**Data da Atualização**
Thâmara Cordeiro de Castro	30 de novembro, 10:39
Thâmara Cordeiro de Castro	13 de dezembro, 13:09
Thâmara Cordeiro de Castro	15 de dezembro, 11:52

Trabalho da matéria de Projeto de Software
Grupo 6

1. Introdução

O sistema EasyStop foi desenvolvido com uma arquitetura modular e organizada, fundamentada no padrão MVC (Model–View–Controller). A divisão clara das responsabilidades, Modelos de Domínio, Lógica de Controle e Interface Gráfica, proporciona alto grau de coesão, baixo acoplamento, facilidade de manutenção e escalabilidade natural da aplicação. Além disso, o projeto incorpora boas práticas de persistência com ORMLite, interface gráfica com JavaFX/FXML e armazenamento local utilizando SQLite.

2. Contexto do Sistema

No coração de qualquer estacionamento, existe um fluxo constante de veículos que precisa ser gerenciado com precisão e eficiência. O EasyStop nasce para enfrentar os desafios cotidianos enfrentados por gestores e funcionários de estacionamentos, que muitas vezes dependem de métodos manuais e desorganizados para controlar o movimento de entrada e saída de carros.

Imagine um estacionamento movimentado em um dia típico. Os funcionários precisam anotar manualmente as placas dos veículos, os horários de entrada, as vagas ocupadas e calcular manualmente o tempo de permanência quando o cliente retorna. Esse processo, além de consumir tempo valioso, está sujeito a erros humanos — uma placa registrada incorretamente, um horário anotado com imprecisão, um cálculo de valor feito de maneira equivocada. Cada erro pode significar prejuízo financeiro, insatisfação do cliente ou até mesmo conflitos na hora do pagamento.

Os clientes, por sua vez, enfrentam suas próprias frustrações. Esperam agilidade no momento da entrada, mas se deparam com filas enquanto o funcionário preenche fichas de papel. Na saída, a espera pode ser ainda maior, enquanto são feitos cálculos manuais e é gerado um comprovante de pagamento. Essa demora não só irrita o cliente, que muitas vezes está com pressa, mas também reduz a capacidade do estacionamento de atender a um maior número de veículos ao longo do dia.

Para o gestor do estabelecimento, a falta de controle é uma dor constante. Sem um sistema organizado, é difícil acompanhar quais vagas estão ocupadas, quais estão livres, quais veículos permanecem no local além do horário previsto, e quanto faturamento está sendo gerado em tempo real. A reconciliação de caixa ao final do dia torna-se uma tarefa árdua, exigindo a conferência de papéis, fichas e anotações soltas, com grande margem para divergências e falhas.

Além disso, a experiência do cliente fica comprometida. Não há histórico de visitas, não é possível oferecer facilidades como pagamentos recorrentes ou descontos para clientes frequentes, e qualquer questionamento sobre o valor cobrado depende da busca por uma

ficha de papel que pode ter sido perdida ou arquivada de forma inadequada. A confiança no estabelecimento pode ser abalada por esses pequenos atritos.

O EasyStop surge como resposta a essas dores, oferecendo uma solução que organiza, simplifica e traz transparência para a gestão de estacionamentos. Ele transforma o caos de papéis e anotações em um fluxo digital claro, onde cada entrada, cada vaga ocupada, cada cálculo e cada pagamento são registrados com precisão, beneficiando tanto quem gerencia quanto quem utiliza o espaço.

3. Decisões Arquiteturais

O EasyStop adota uma arquitetura em camadas baseada no padrão MVC, estruturada para suportar os principais fluxos do sistema, como check-in, check-out e processamento de pagamentos.

A arquitetura foi projetada para:
- Isolar regras de negócio da interface gráfica
- Permitir a inclusão de novos meios de pagamento
- Facilitar manutenção e evolução do sistema

Decisão 1 – Uso do padrão MVC
O padrão MVC foi adotado para separar interface (JavaFX/FXML), lógica de controle (Controllers) e domínio. Essa decisão permite evolução da interface sem impacto direto nas regras de negócio.

Decisão 2 – Persistência com ORMLite e SQLite
Foi adotado ORMLite para reduzir código de acesso a dados e SQLite por sua leveza e adequação a aplicações locais.

Decisão 3 – Processamento de Pagamentos no Domínio
As regras de pagamento foram concentradas no domínio para evitar lógica de negócio na interface ou nos serviços.

4. Padrões Arquiteturais e Projeto

O sistema adota como base o **padrão arquitetural MVC** (Model–View–Controller), promovendo uma separação clara de responsabilidades entre a interface de usuário, a lógica de controle e o domínio da aplicação. Essa organização contribui para a manutenibilidade do código, facilita a evolução independente das camadas e reduz o acoplamento entre os componentes do sistema.

No domínio da aplicação, é utilizado o **Template Method Pattern** para estruturar o fluxo de processamento de pagamentos, definindo um algoritmo padrão e permitindo variações de comportamento conforme o meio de pagamento adotado. Essa abordagem garante consistência no processo e facilita a extensão do sistema para novos tipos de pagamento.

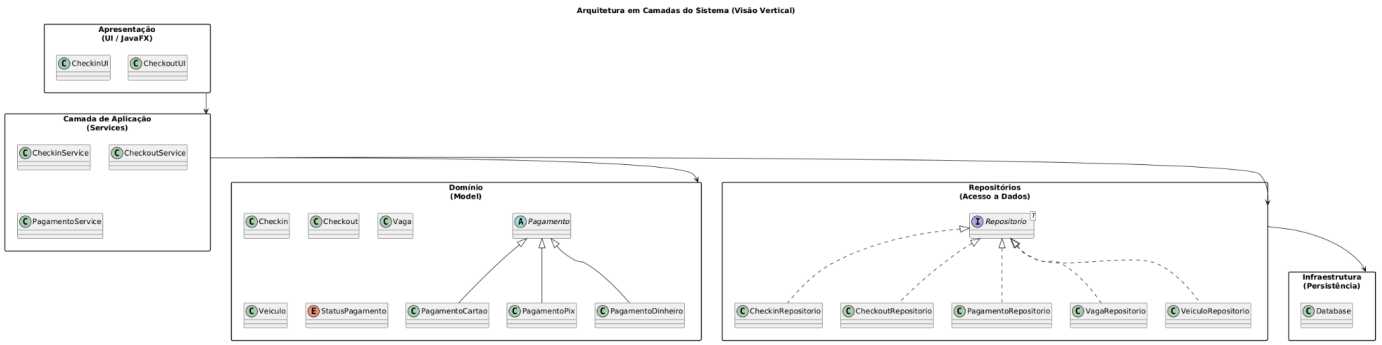
De forma complementar, o sistema aplica o padrão **Strategy** de maneira implícita, possibilitando a seleção dinâmica do algoritmo de pagamento em tempo de execução. Além disso, o Repository Pattern é empregado para abstrair o acesso aos dados, isolando a persistência do restante da aplicação, enquanto o **Facade Pattern** é utilizado na camada de serviços para simplificar a execução dos casos de uso. Por fim, o padrão Singleton é adotado na infraestrutura para controlar o acesso ao banco de dados.

A combinação desses padrões resulta em uma arquitetura coesa, extensível e alinhada aos princípios SOLID, favorecendo a organização do código e a manutenção do sistema ao longo do tempo.

5. Arquitetura Geral

5.1 Visão de Camadas

O diagrama abaixo representa a visão arquitetural em camadas, abstraindo as classes concretas para não poluir o diagrama, mas mantendo fidelidade à sua organização de pastas e responsabilidades.



6. Fluxo Arquitetural de um Caso de Uso

O fluxo arquitetural do processamento de pagamento inicia-se na camada de apresentação, onde o usuário seleciona o meio de pagamento e informa os dados necessários por meio da interface JavaFX.

O Controller recebe a ação do usuário e encaminha a solicitação para a camada de serviços, que atua como uma fachada para o caso de uso. O serviço é responsável por instanciar o tipo de pagamento adequado (Cartão, PIX ou Dinheiro), com base na opção selecionada.

A partir desse ponto, o processamento é delegado ao domínio. O método processarPagamento é executado conforme o algoritmo definido na classe abstrata Pagamento, seguindo o Template Method Pattern. As regras específicas são aplicadas pela subclasse correspondente ao meio de pagamento escolhido.

Após a validação e definição do status do pagamento, o serviço solicita ao repositório a persistência das informações no banco de dados, utilizando o ORMLite. Por fim, o resultado do processamento é retornado ao Controller, que atualiza a interface do usuário.

7. Princípios de Design Aplicados

O projeto foi desenvolvido com base em princípios de design de software que visam promover organização, manutenibilidade, extensibilidade e baixo acoplamento. A seguir são apresentados os principais princípios aplicados no sistema.

Single Responsibility Principle (SRP)

O princípio da **Responsabilidade Única** é amplamente aplicado no sistema, garantindo que cada classe possua um único motivo para mudança.

As entidades de domínio, como `Pagamento`, `Checkout`, `Checkin`, `Vaga` e `Veiculo`, concentram apenas regras de negócio. As classes específicas de pagamento (`PagamentoCartao`, `PagamentoPix` e `PagamentoDinheiro`) encapsulam exclusivamente as regras relacionadas ao respectivo meio de pagamento. Já os repositórios são responsáveis apenas pelo acesso e persistência dos dados, enquanto a camada de serviços coordena os fluxos dos casos de uso.

Open/Closed Principle (OCP)

O sistema segue o princípio de que entidades de software devem estar **abertas para extensão, mas fechadas para modificação**.

Esse princípio é claramente observado no módulo de pagamentos. A classe abstrata `Pagamento` define um comportamento genérico, permitindo que novos meios de pagamento sejam adicionados por meio de subclasses, sem a necessidade de alterar o código existente. Dessa forma, o sistema pode evoluir com menor risco de introdução de erros.

Liskov Substitution Principle (LSP)

O princípio da **Substituição de Liskov** é respeitado ao permitir que qualquer instância de `PagamentoCartao`, `PagamentoPix` ou `PagamentoDinheiro` seja utilizada no lugar da classe base `Pagamento`, sem comprometer o funcionamento do sistema.

Todas as subclasses mantêm o contrato definido pela classe abstrata, especialmente no que se refere ao método `processarPagamento`, garantindo consistência no comportamento esperado.

Dependency Inversion Principle (DIP)

O sistema aplica o princípio da **Inversão de Dependência** ao fazer com que as camadas superiores dependam de abstrações, e não de implementações concretas.

A utilização da interface genérica `Repositorio<T>` permite que a camada de serviços opere sobre abstrações, sem conhecer detalhes da persistência ou da tecnologia utilizada. Isso reduz o acoplamento e facilita a substituição de implementações, bem como a criação de testes automatizados.

Separation of Concerns (SoC)

O princípio de **Separação de Interesses** é atendido por meio da arquitetura em camadas e do padrão MVC.

Cada camada possui responsabilidades bem definidas: a apresentação lida com a interface do usuário, a camada de serviços coordena os casos de uso, o domínio concentra as regras de negócio e a persistência é isolada nos repositórios e na infraestrutura. Essa separação contribui para a clareza da arquitetura e para a evolução controlada do sistema.

High Cohesion and Low Coupling

O sistema apresenta **alta coesão**, uma vez que classes e pacotes possuem responsabilidades bem delimitadas, e **baixo acoplamento**, pois as dependências entre os componentes são minimizadas e mediadas por abstrações.

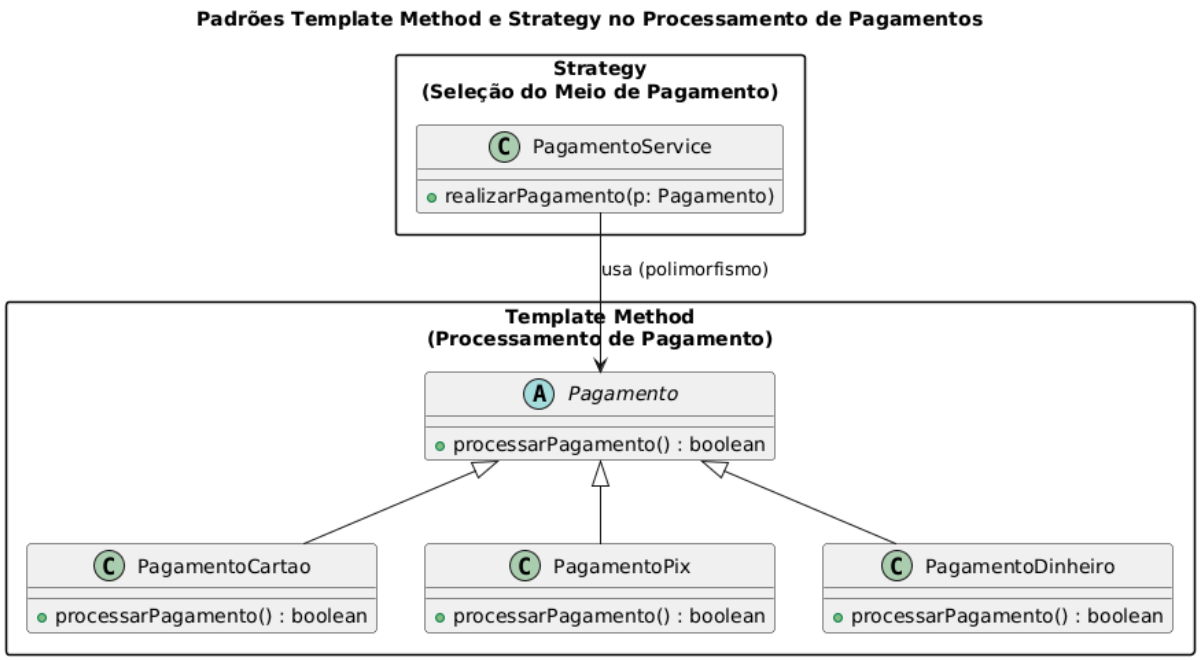
Esse equilíbrio favorece a reutilização de código, reduz impactos de mudanças e melhora a qualidade geral da arquitetura.

8. Template Method Pattern Aplicado

8.1Camada Model - Domínio e Regras de Negócios

O sistema precisa processar diferentes tipos de pagamento (Cartão, PIX, Dinheiro), cada um com regras de validação e execução específicas, mas seguindo um fluxo comum de processamento.

```
Pagamento (abstract)
├── PagamentoCartao (concrete)
├── PagamentoPix (concrete)
└── PagamentoDinheiro (concrete)
```



8.2 Regras de Negócio por Tipo de Pagamento

8.2.1 PagamentoCartao

Ao instanciar um pagamento com cartão, o sistema identifica automaticamente a bandeira do cartão com base nos padrões do número informado:

- Cartões iniciados com 4 → VISA
- Cartões iniciados com 51–55 → MASTERCARD
- Cartões iniciados com 34 ou 37 → AMEX
- Qualquer outro padrão → DESCONHECIDA

Essa regra é encapsulada no método privado detectarBandeira, garantindo que a lógica de identificação fique centralizada no domínio e não espalhada pela aplicação.

9.2.2 PagamentoPix

A classe PagamentoPix implementa apenas as regras de negócio específicas do pagamento via PIX, respeitando o contrato estabelecido pela classe base e mantendo a consistência do fluxo de pagamento no sistema.

Todo pagamento via PIX deve estar obrigatoriamente associado a um objeto Checkout, que representa o contexto da compra, bem como a um valor monetário. Essas informações são

inicializadas no construtor da classe por meio da chamada ao construtor da superclasse Pagamento.

Essa regra garante que nenhum pagamento exista de forma isolada, sempre estando vinculado a uma transação válida.

O tipo do pagamento é definido como "PIX" no momento da criação do objeto. Essa identificação é utilizada pelo sistema para:

- diferenciar os meios de pagamento
- aplicar regras específicas
- facilitar auditoria e relatórios

Durante a execução do método processarPagamento, o sistema valida a existência do código PIX informado pelo usuário. A regra de negócio estabelece que:

- O código PIX **não** pode ser nulo
- O código PIX **não** pode estar vazio ou em branco

Caso essa condição **não** seja atendida:

- O pagamento é automaticamente **recusado**

O status é atualizado para StatusPagamento.RECUSADO, o

- método retorna false

Se o código PIX estiver presente e válido:

- O pagamento é considerado aprovado

O status é atualizado para StatusPagamento.APROVADO

- O método retorna true

Diferentemente do pagamento com cartão, o pagamento via PIX não exige validações adicionais de formato ou segurança neste contexto, refletindo a simplicidade e rapidez características desse meio de pagamento.

8.2.3 - Fluxo Arquitetural do Caso de Uso: Processar Pagamento

O fluxo arquitetural do processamento de pagamento inicia-se na camada de apresentação, onde o usuário seleciona o meio de pagamento e informa os dados necessários por meio da interface JavaFX.

O Controller recebe a ação do usuário e encaminha a solicitação para a camada de serviços, que atua como uma fachada para o caso de uso. O serviço é responsável por instanciar o tipo de pagamento adequado (Cartão, PIX ou Dinheiro), com base na opção selecionada.

A partir desse ponto, o processamento é delegado ao domínio. O método processarPagamento é executado conforme o algoritmo definido na classe abstrata Pagamento, seguindo o Template Method Pattern. As regras específicas são aplicadas pela subclasse correspondente ao meio de pagamento escolhido.

Após a validação e definição do status do pagamento, o serviço solicita ao repositório a persistência das informações no banco de dados, utilizando o ORMLite. Por fim, o resultado do processamento é retornado ao Controller, que atualiza a interface do usuário.

A classe `PagamentoDinheiro` implementa apenas as regras necessárias para validar e concluir pagamentos realizados em dinheiro, respeitando o contrato estabelecido pela superclasse e mantendo a uniformidade do fluxo de pagamento no sistema.

Todo pagamento em dinheiro deve estar obrigatoriamente associado a um objeto `Checkout` e a um valor total da compra. Essa associação é realizada no construtor por meio da chamada ao construtor da classe `Pagamento`, garantindo que o pagamento esteja sempre vinculado a uma transação válida no sistema.

O pagamento é identificado internamente como do tipo `"DINHEIRO"`. Essa informação é utilizada pelo sistema para:

- diferenciação entre meios de pagamento
- geração de relatórios
- aplicação de regras específicas conforme o tipo de pagamento

Durante a execução do método `processarPagamento`, o sistema aplica a principal regra de negócio do pagamento em dinheiro:

- O **valor recebido do cliente** deve ser **maior ou igual** ao valor total da compra.

Caso o valor recebido seja **inferior** ao valor devido:

- O pagamento é **recusado**

- O status é atualizado para `StatusPagamento.RECUSADO`
- O método retorna `false`

Se o valor recebido for suficiente para cobrir o valor da compra:

- O pagamento é considerado **aprovado**
- O status é atualizado para `StatusPagamento.APROVADO`
- O método retorna `true`

Nesse contexto, não há necessidade de validações adicionais ou comunicação externa, uma vez que o pagamento em dinheiro ocorre de forma imediata.

A classe mantém o controle do **troco** por meio do atributo `troco`, permitindo que o sistema registre:

- o valor excedente entregue pelo cliente
- a diferença entre o valor recebido e o valor da compra

Embora o cálculo do troco não esteja explícito no método `processarPagamento`, sua presença no domínio permite que essa regra seja aplicada previamente ou por outro componente do sistema, mantendo a responsabilidade de validação focada no pagamento.

A classe PagamentoDinheiro sobrescreve o método processarPagamento, que representa um passo variável do algoritmo definido na classe abstrata Pagamento.

Enquanto a classe base define a estrutura geral do processo de pagamento, a classe PagamentoDinheiro define as regras específicas para validação e aprovação de pagamentos realizados em dinheiro.

Essa abordagem permite:

- reutilização do fluxo de pagamento
- baixo acoplamento entre meios de pagamento
- fácil inclusão de novos tipos de pagamento no futuro

10 .Estrutura de Pastas e Organização em Camadas

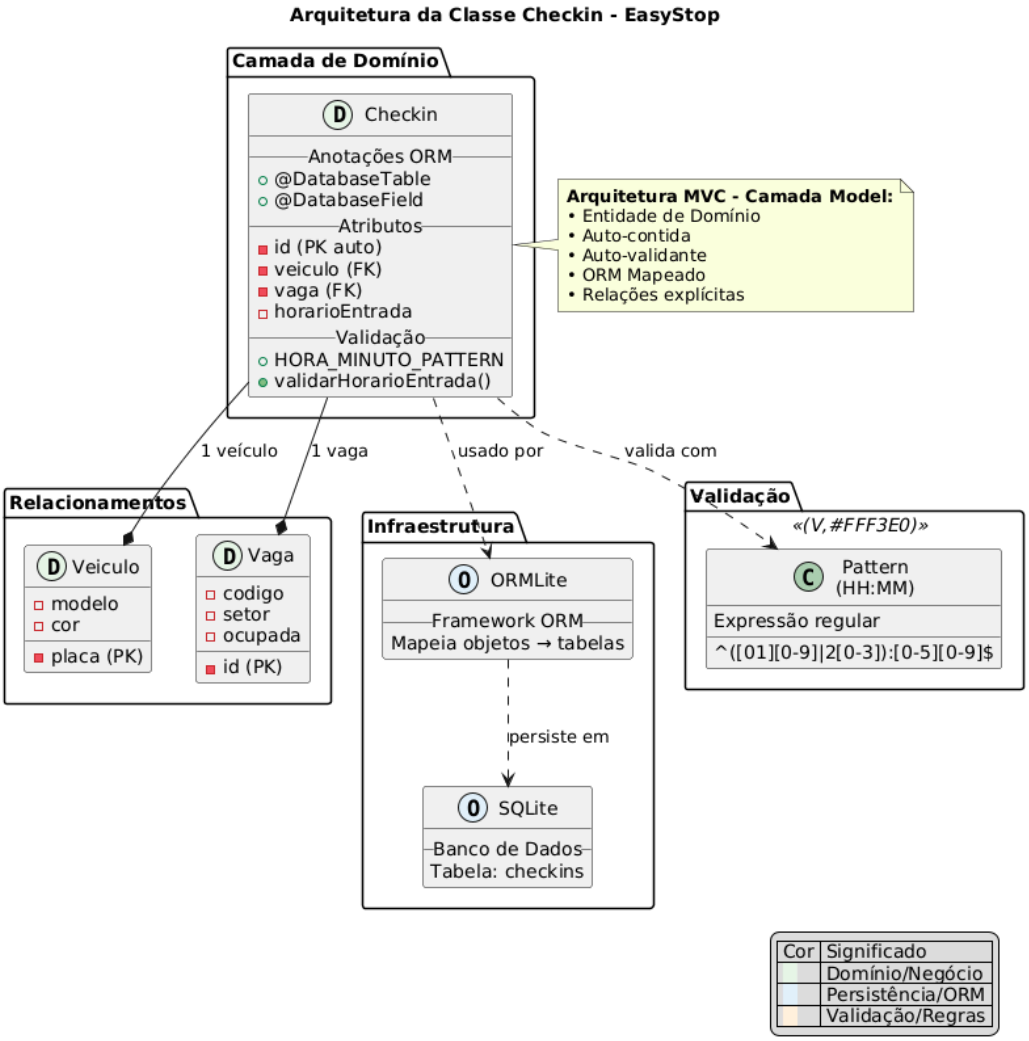
A estrutura do projeto é segmentada em diretórios que refletem diretamente as camadas arquiteturais, facilitando a navegação e promovendo clareza no fluxo da aplicação:

1.1 Camada Model

Representa a camada de domínio e concentra toda a lógica central do sistema:

- **Entidade de Domínio**
    - São classes que mapeiam diretamente as tabelas do banco de dados, usando anotações do ORMlite.

Exemplo 1. model/Checkin.Java



img 01 - Arquitetura de Classe Checkin

A primeira camada de funcionalidade que encontramos é o mapeamento objeto-relacional. A anotação `@DatabaseTable(tableName = "checkins")` atua como um contrato formal entre o mundo dos objetos Java e o universo relacional do banco de dados. Esta simples declaração estabelece que cada instância da classe Checkin corresponde inequivocamente a um registro na tabela "checkins" do banco SQLite. O ORMLite, nosso mediador tecnológico, interpreta este contrato e executa a tradução silenciosa entre paradigmas – transforma propriedades de objetos em colunas de tabelas, métodos em constraints, e herança em estratégias de mapeamento.

@DatabaseTable(tableName = "checkins") // ← MAPEAMENTO TABELA

```
public class Checkin {

    @DatabaseField(generatedId = true)  // ← CHAVE PRIMÁRIA AUTOMÁTICA
    private int id;

    @DatabaseField(foreign = true, foreignAutoRefresh = true,  // ← RELACIONAMENTO
            canBeNull = false, columnName = "veiculo_placa")
    private Veiculo veiculo;
```

Dentro deste contrato, cada atributo da classe recebe seu papel específico através das anotações `@DatabaseField`. O campo `id` com `generatedId = true` não é apenas um número sequencial; é a identidade única do checkin no sistema, gerada automaticamente pelo banco de dados para garantir unicidade e rastreabilidade. Já os campos `veiculo` e `vaga` com `foreign = true` representam algo mais profundo: são relacionamentos vivos. Eles não armazenam meros identificadores, mas mantêm referências a objetos completos. Quando dizemos que um Checkin possui um Veiculo, estamos estabelecendo uma conexão semântica que o sistema preserva através do ciclo de vida completo da transação.

- Repositórios (*Repositorio) Implementam operações CRUD e consultas específicas usando DAOs do ORMLite
  Cada entidade possui seu repositório dedicado

- Serviços
  - As regras de negócios ficam nas classes concretas e o serviço apenas orquestra

Componente	**Responsabilidade**	**Complexidade**
ServicoPagamento	Orquestração simples	Baixa
Pagamento (abstrata)	Template do algoritmo	Média
PagamentoCartao/Pix/Dinheiro	Regras específicas	Alta
PagamentoRepositorio	Persistência	Baixa