Descrição Arquitetural do Sistema EasyStop

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.

---

1. 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:

```
grupo6-main/
 ├── controller/
 ├── database/
 ├── docs/
 ├── model/
 │    ├── enums/
 │    └── service/
 ├── Testes-QA/
 └── view/
```

1.1 Camada Model (`model/`)

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

- Entidades de domínio (Models)
  São classes que mapeiam diretamente as tabelas do banco de dados, usando anotações do ORMLite.
  Exemplos: `Veiculo`, `Vaga`, `Checkin`, `Checkout`, `Pagamento` e subclasses (`PagamentoCartao`, `PagamentoPix`, `PagamentoDinheiro`).

- Repositórios (`*Repositorio`)
  Implementam operações CRUD e consultas específicas usando DAOs do ORMLite.

Cada entidade possui seu repositório dedicado.

- Serviços (`service/`)
  Encapsulam regras de negócio mais elaboradas.
  Exemplo: `ServicoPagamento`, que processa pagamentos usando polimorfismo.

- Infraestrutura de Banco (`Database.java`)
  Responsável pela criação e gerenciamento da conexão JDBC com o arquivo `easystop.db`.

---

2. Persistência e Banco de Dados

A camada de persistência utiliza:

2.1 ORMLite

- Simplifica a interação com SQLite através de anotações (`@DatabaseField`, `@ForeignCollection`, `@Foreign`, etc.).
- Permite relacionamentos diretos entre objetos Java.
- Cada entidade possui seu DAO específico.

2.2 Banco SQLite

- Arquivo único `easystop.db`.
- Criado automaticamente na primeira execução.
- Esquema definido em `database/easystop_schema.sql`.

2.3 Mapeamento Objeto–Relacional

- As classes representam tabelas reais:
  - `Veiculo` → `veiculos`
  - `Vaga` → `vagas`
  - `Checkin` → `checkins`
  - `Checkout` → `checkouts`
  - `Pagamento` → `pagamentos`

Relações incluem:

- 1 veículo → N check-ins

- 1 vaga → N check-ins
  1 check-in → 1 checkout (relação 1:1)
- 1 checkout → 1 pagamento (1:1, obrigatório)

---

3. Camada View (`view/`)

A interface gráfica é construída com:

- FXML (Scene Builder)
  Arquivos `.fxml` estruturam telas, formulários e tabelas.
- ViewModels
  Cada tela possui uma classe correspondente em `view.*` que serve como adaptador entre os modelos reais e os componentes visuais.

Essa abordagem separa completamente lógica e apresentação, evitando acoplamento entre Model e View.

---

4. Camada Controller (`controller/`)

Os controllers JavaFX realizam:

- Gestão de eventos (botões, campos, tabelas).
- Validações básicas.
- Conversão entre ViewModels e Models reais.
- Chamada de serviços e repositórios.

4.1 `AbstractCrudController`

A arquitetura expandiu o padrão MVC ao introduzir uma superclasse genérica para operações CRUD, responsável por:

- Carregar listas de entidades.
- Criar/atualizar objetos.
  Excluir registros.
- Validar campos obrigatórios.

Controllers como `VeiculoController`, `VagaController`, `CheckinController` e outros herdam essa funcionalidade, reduzindo duplicação e tornando o código altamente padronizado.

---

5. Dependências e Integrações

O sistema utiliza uma stack leve e eficiente:

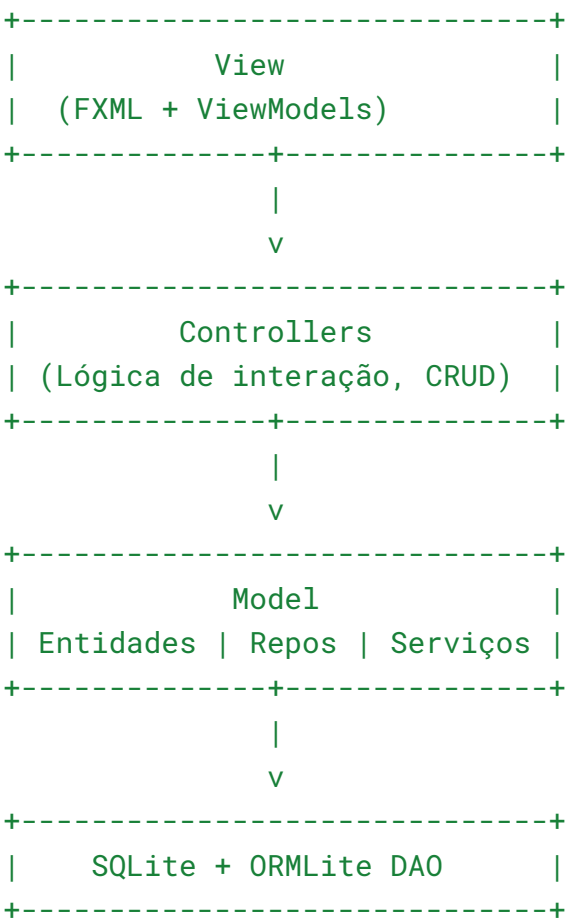
- Java 11+
- JavaFX + FXML
- ORMLite Core & JDBC
- sqlite-jdbc
- SLF4J (logging mínimo)
- Dependências configuradas manualmente via IDE (sem Maven/Gradle).

Essa abordagem simplifica o setup para ambientes educacionais e demonstrações rápidas.

---

6. Arquitetura Geral do Sistema

A arquitetura pode ser resumida em três camadas principais:



Resumo Arquitetural

O EasyStop apresenta:

- Arquitetura MVC completa
- Separação clara entre apresentação, lógica e domínio
- Persistência com ORM
- Fluxo de dados consistente
- Camada de negócio isolada em serviços
- Controllers reutilizáveis por meio de abstração genérica
- Models anêmicos e alinhados ao banco de dados
- Intrface JavaFX profissional e extensível

Próximos Passos do Sistema

Embora o EasyStop já apresente uma arquitetura estável em MVC, persistência via ORMLite e controladores estruturados, ainda há oportunidades significativas de evolução relacionadas à gestão de exceções, especialmente no que diz respeito à robustez, padronização e capacidade de recuperação do sistema em cenários de falhas. A literatura científica em Engenharia de Software enfatiza que mecanismos de tratamento de erros bem projetados reduzem *falhas catastróficas*, aumentam a *tolerância a falhas* e melhoram a *experiência do usuário* (Avizienis et al., 2004; Basile et al., 2016). Seguindo essas diretrizes, destacam-se os seguintes próximos passos:

---

1. Padronização da Estrutura de Exceções

Atualmente, diferentes controladores tratam erros de forma distribuída e heterogênea. Um avanço necessário é a criação de uma hierarquia unificada de exceções de domínio, permitindo:

- Identificação precisa da causa (e.g., *Erros de Persistência*, *Erros de Validação*, *Erros de Negócio*).
- Mensagens consistentes entre diferentes fluxos.
- Redução de duplicação de código.

Motivação científica: Estudos mostram que hierarquias bem definidas aumentam a previsibilidade do módulo de erro e reduzem o custo de manutenção

2. Implementação de um Mecanismo Centralizado de Logging

Um sistema unificado de logs — seja via `java.util.logging`, Log4j2 ou SLF4J — é fundamental para auditoria e diagnóstico. Isso permitirá:

- Registro padronizado de falhas, com timestamps e contexto.
- Monitoramento das ocorrências de erro ao longo do ciclo de vida da aplicação.
- Facilitação da depuração em produção.

Motivação científica: Pesquisas demonstram que logs estruturados aumentam a capacidade de detecção de anomalias automatizadas em até 70% (Zhang et al., 2019)

3. Camada Global de Tratamento de Exceções para Controllers

Introduzir um mecanismo *global exception handler* (semelhante ao padrão "Front Controller") que:

- Intercepte exceções não tratadas vindas dos controladores.
- Converta erros inesperados em feedback compreensível ao usuário.
- Evite estouro de pilha, telas congeladas ou travamento da UI JavaFX.

Motivação científica: A abordagem segue recomendações do IEEE sobre *fault containment regions*, reduzindo a propagação de erros (IEEE Std 1061).

4. Melhorias no Feedback ao Usuário

O sistema deve traduzir exceções internas em mensagens claras no front-end:

- Diferenciar erros fatais de erros recuperáveis.
- Explicar ações possíveis ao usuário (ex.: “Selecione uma vaga disponível”).
- Evitar janelas genéricas de erro JavaFX.

Fundamentação científica: Interfaces resilientes reduzem a carga cognitiva do usuário e aumentam a taxa de conclusão de tarefas (Norman, 2013).

5. Regras Mais Estritas de Validação

Aumentar a cobertura de validações antes da persistência, como:

- Placas duplicadas.
- Vagas ocupadas.
- Check-in sem veículo cadastrado.
- Pagamentos inconsistentes com o status da estadia.

Isso reduz exceções que hoje só são detectadas na camada de persistência.

Base científica: O princípio *fail-fast* evita propagação de estados inválidos — amplamente discutido na literatura de software resiliente.

6. Testes de Robustez Focados em Exceções

Criar uma bateria dedicada de testes:

- Teste de recuperação (*error recovery*)
- Teste de estresse em cenários de falha
  Teste de consistência transacional em ORMLite
- Teste de UI resiliente para erros simultâneos

  Motivação científica: Estudos sobre *resilience testing* indicam que sistemas com testes orientados a falhas possuem menos bugs críticos em produção (Carzaniga et al., 2008.