

Projeto de Software

**Alunos: João da Costa, Lucas de
Lima, Luiz Fernando, Ignacio
Sander, Matheus Janibelli**

Sumário

- Ideia do projeto
- O escopo
- Os requisitos arquiteturais
- Restrições da arquitetura
- Padrões arq. adotados
- Diagramas

Ideia do projeto

Para realizar o trabalho, nosso grupo decidiu criar um sistema de delivery de comida similar ao “iFood”.



O escopo

Nosso projeto consiste em desenvolver uma aplicação que simula as funcionalidades básicas de um sistema de pedidos de comida por delivery, semelhante ao “iFood”. Esse sistema será composto por *três* perfis principais de usuários: **Clientes, Donos de Restaurante e Entregadores.**

O objetivo é permitir que clientes naveguem por restaurantes, escolham produtos, façam pedidos e acompanhem seu status. Já os Donos de Restaurantes vão poder cadastrar um restaurante e administrá-lo, alterando certas informações e o cardápio. Por fim, os Entregadores serão responsáveis por cadastrar seus veículos e aceitar/recusar entregas.



Os requisitos arquiteturais

- **Modularidade:** Separar claramente as responsabilidades em camadas (ex:apresentação, lógica de negócios, persistência de dados).

→ *Objetivo:* Facilitar manutenção e evolução do sistema.

- **Escalabilidade:** Garantir que a aplicação possa ser expandida futuramente, incluindo novos recursos como pagamentos reais ou geolocalização.

→ *Objetivo:* Permitir que novos módulos sejam incluídos sem problemas futuramente.

- **Reutilização de componentes:** Reaproveitar partes do sistema (ex: componentes de login, listagens de itens) em diferentes contextos.

→ *Objetivo:* Manter qualidade do código.

- **Facilidade de manutenção:** A arquitetura deve permitir correções rápidas e fáceis sem causar efeitos colaterais indesejados.

→ *Objetivo:* Corrigir eventuais problemas de forma rápida e segura.

- **Desempenho aceitável:** O sistema deve responder de forma fluida a interações comuns dos usuários, como adicionar itens ao carrinho ou confirmar pedidos.

→ *Objetivo:* Manter uma boa experiência de usuário.

- **Compatibilidade:** O sistema deve ser compatível com Windows 10 e Windows 11.

→ *Objetivo:* Permitir que usuários com os principais sistemas operacionais sejam capazes de utilizar o sistema.

- **Manutenibilidade:** O código deve ser bem documentado e devem ser utilizadas bibliotecas com boa documentação.

→ *Objetivo:* Facilitar eventuais mudanças no código.

Restrições da Arquitetura

- **Recursos Limitados:**

- Projeto inteiro vai rodar apenas localmente.

- Sem integração com meios de pagamento reais (simulação apenas).

- **Prazo de Entrega:**

- Para respeitar o prazo de entrega, a arquitetura deve dar prioridade à simplicidade.

- **Segurança Básica:**

- Implementar segurança mínima com criptografia simples de senhas.

- Não haverá criptografia ponta-a-ponta nem verificação em duas etapas.

- **Respeitar as especificações da professora:**

- É necessário que a arquitetura esteja de acordo com as especificações dadas pela professora

Padrões arq. adotados

Inicialmente o grupo analisou as alternativas dentre os modelos arquiteturais vistos em

sala de aula e chegamos a *três* os quais consideramos os melhores:

→ **Microserviços** (Microservices)

Motivo:

- Permite que cada tipo de usuário (cliente, restaurante, entregador) tenha seu próprio serviço dedicado.
- Facilita a escalabilidade e manutenção, já que cada funcionalidade (ex: gerenciamento de pedidos, autenticação, pagamentos, etc.) pode ser desenvolvida e implantada separadamente.
- É ideal para sistemas com grande complexidade e com múltiplos fluxos simultâneos.

→ **MVC** (Model-View-Controller)

Motivo:

- Útil para organizar o código da aplicação front-end e back-end, separando interface, lógica e dados.
- Ajuda a manter a clareza e a separação de responsabilidades, especialmente no desenvolvimento do app cliente.

→ **Camadas** (Layered Architecture)

Motivo:

- Ajuda a organizar a aplicação em diferentes camadas: interface (app), lógica de negócio, e persistência (banco de dados).
- Isso torna o sistema mais modular e facilita a manutenção.

Padrões arq. adotados – Nossa Escolha

Após análise, decidimos utilizar o padrão **MVC (Model-View-Controller)**. Abaixo estão listadas as justificativas para a escolha:

1. Modularidade

- O padrão MVC separa o sistema em três camadas bem definidas:
 - *Model*: Regras de negócio e dados.
 - *View*: Interface com o usuário.
 - *Controller*: Lógica de controle e fluxo entre Model e View.
- Isso facilita o desenvolvimento paralelo por diferentes membros da equipe e permite alterar uma parte sem afetar as outras.

2. Manutenibilidade

- Com o código separado por responsabilidades, é mais fácil localizar, entender e corrigir erros.

3. Reusabilidade

- Os *Models* (como Pedido, Restaurante, Produto) podem ser reutilizados entre diferentes interfaces (ex: app do cliente e painel do dono do restaurante).

4. Escalabilidade

- Embora o MVC por si só não seja distribuído, ele organiza o sistema de forma que partes específicas possam ser isoladas e escaladas posteriormente.
- Por exemplo: se o módulo de pedidos crescer, pode ser facilmente extraído para um serviço separado no futuro.

Padrões arq. adotados – Nossa Escolha

Após análise, decidimos utilizar o padrão **MVC (Model-View-Controller)**. Abaixo estão listadas as justificativas para a escolha:

5. Facilidade de Testes

- A separação entre lógica de negócio, controle e interface facilita testes unitários e de integração.
- Pode-se testar a lógica dos pedidos ou cálculo de taxas sem depender da interface gráfica.

6. Usabilidade

- Como a View está separada da lógica, a equipe pode focar em criar uma experiência de usuário fluida sem interferir no backend.
- Permite criação de interfaces diferentes para cada perfil de usuário (Cliente, Dono, Entregador), com base nas mesmas regras de negócio.

7. Flexibilidade

- Mudanças na interface, como design ou layout, não afetam a lógica de negócios nem os dados.
- É possível adaptar o mesmo backend para diferentes interfaces (ex: app Android, iOS e painel web).

Em resumo:

A adoção do padrão **MVC** traz uma organização clara que melhora a **modularidade**, **manutenibilidade** e **testabilidade** do sistema, além de permitir **reuso** de componentes e **facilitar** o desenvolvimento de interfaces para diferentes tipos de usuários. Essa estrutura é especialmente valiosa para um sistema com múltiplos perfis e funcionalidades como o app do *iFood* proposto.

Diagramas

Diagrama com Visão Geral do Sistema - Arquitetura MVC

Camada Model (Negócio + Dados)

- Cliente
- Restaurante/Loja
- Produto
- Pedido
- Pagamento
- Entrega
- Entregador

Funções:

- Contêm regras de negócio
- Lógica de persistência
- Integração com banco de dados

Camada View (Interface do Usuário)

- App Web/Mobile Cliente:
 - Tela de login
 - Tela de busca de restaurantes
 - Cupom
 - Cartão e pagamento
 - Tela de carrinho
 - Tela de pedido em andamento
- App Restaurante:
 - Tela de recebimento de pedidos
 - Tela de status do pedido
 - Faturamento
 - Controle de Lojas
 - Produtos
 - Preço
- App Entregador:
 - Tela de pedidos disponíveis
 - Tela de entrega ativa
 - Rotas de entregas

Camada Controller (Orquestrador)

- PedidoController
- PagamentoController
- EntregaController
- ClienteController
- RestauranteController

Funções:

- Recebe requisições da view
- Valida dados
- Chama os métodos do model
- Retorna resposta para a view

Diagramas

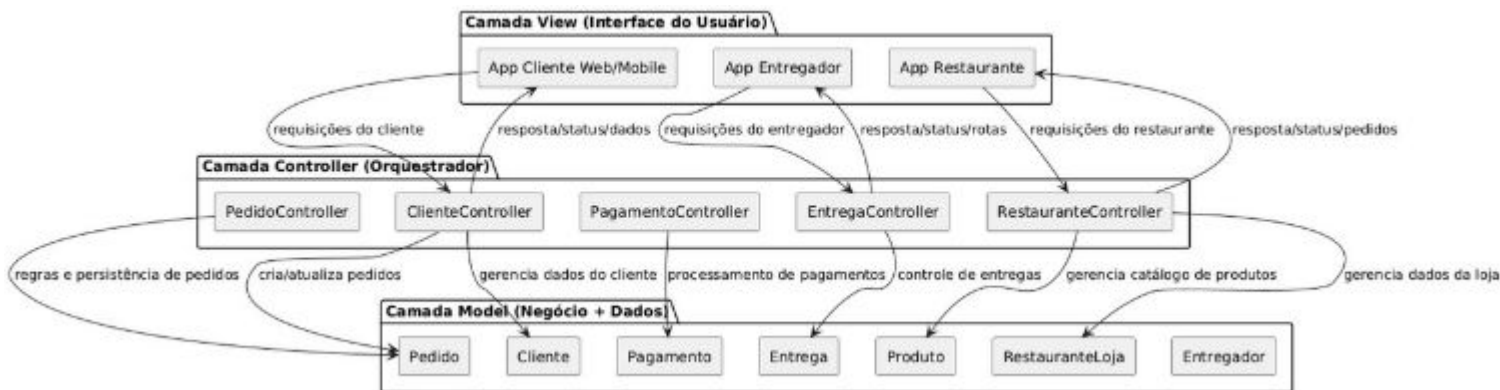
Diagrama com Visão Geral do Sistema - Arquitetura MVC

Fluxo resumido (MVC aplicado ao fluxo "Fazer Pedido")

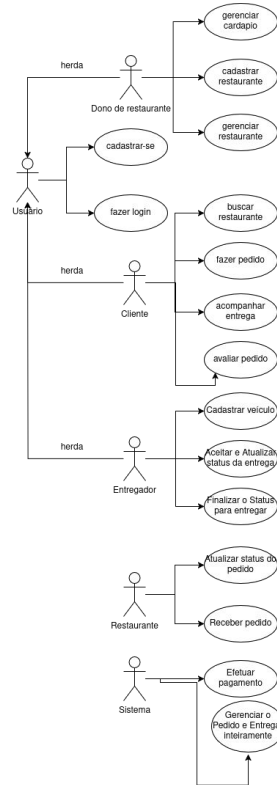
1. **View: Cliente** seleciona produtos e clica em "Confirmar Pedido"
2. **Controller:** PedidoController recebe os dados, valida e cria o pedido
3. **Model:** **Pedido** é salvo no banco, status inicial é "Aguardando confirmação"
4. **View (Restaurante):** recebe notificação em tempo real do novo pedido
5. **View (Cliente):** mostra "Pedido enviado"

Justificativas:

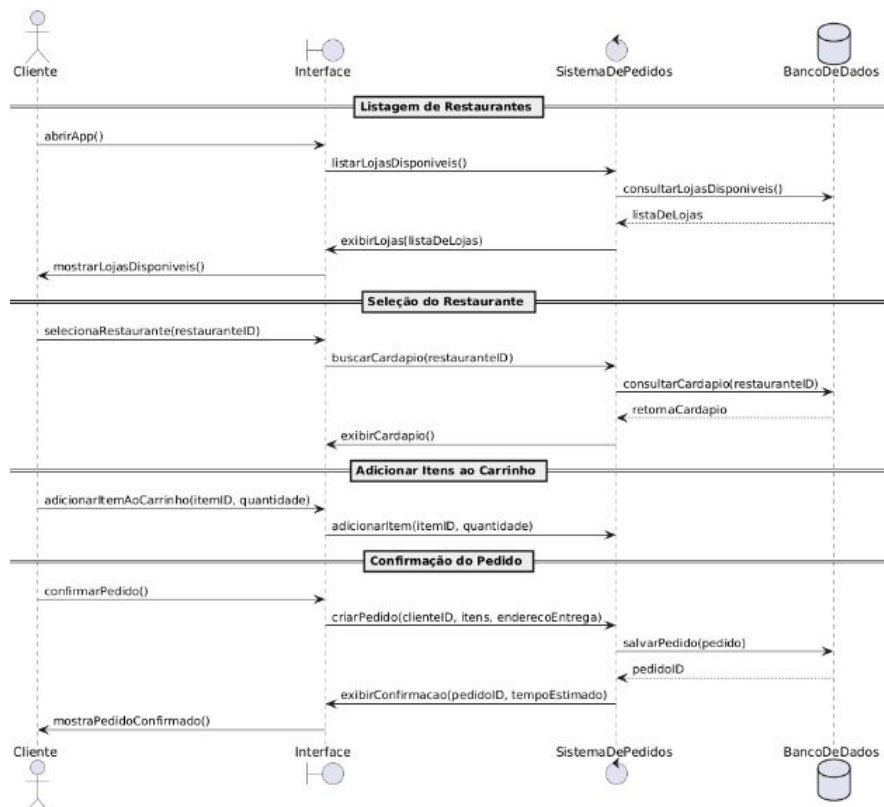
1. Reforça a separação de responsabilidades.
2. Garante organização e facilita testes unitários por camada.
3. Aumenta a manutenibilidade e a escalabilidade.



Diagramas – Casos de Uso



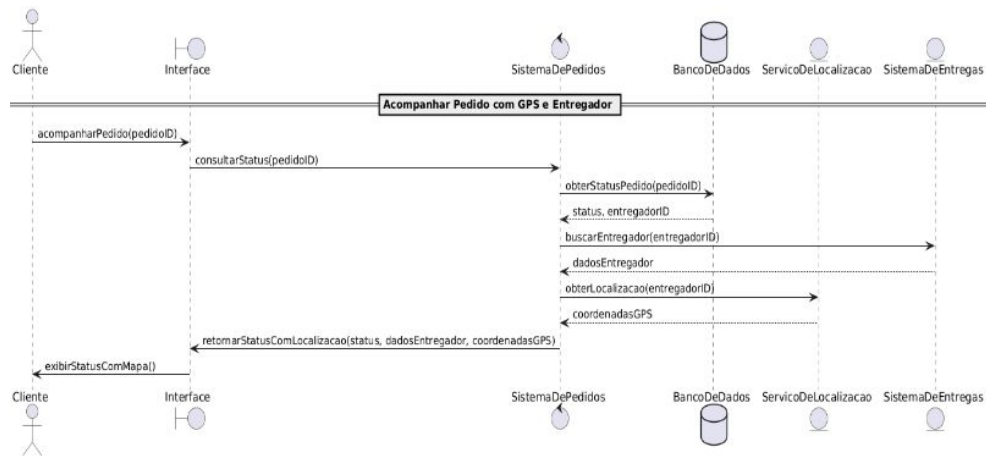
Diagramas - DSS



Contrato da Operação: criar Pedido()

- **Nome:** criarPedido
- **Responsável:** Sistema De Pedidos
- **Descrição:** Cria um novo pedido com base nos itens do carrinho, cliente e endereço.
- **Parâmetros de entrada:**
 - *clienteID*: String,
 - *itens*: List<Item>,
 - *enderecoEntrega*: String
- **Pré-condições:** O cliente está autenticado e há itens no carrinho.
- **Pós-condições:** Pedido é salvo no banco e um ID é retornado junto com o tempo estimado.
- **Saída:**
 - *pedidoID*: String,
 - *tempoEstimado*: int
- **Exceções:** Cliente não encontrado, erro de conexão com banco, carrinho vazio.

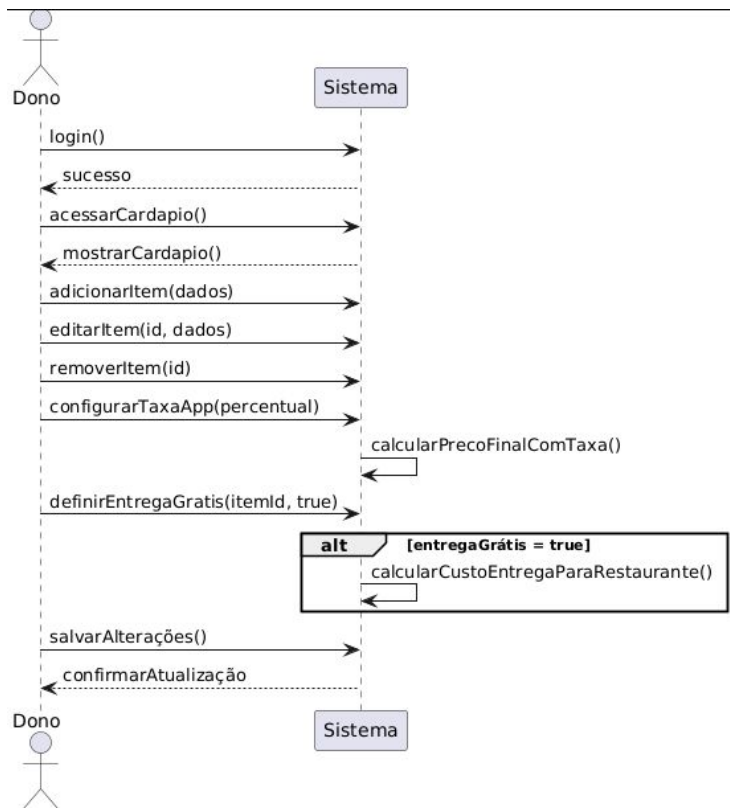
Diagramas - DSS



Contrato da Operação: acompanharPedidoComLocalizacao(pedidoID)

- **Nome:** acompanharPedidoComLocalizacao(pedidoID)
- **Descrição:** Essa operação permite ao cliente acompanhar em tempo real o status do pedido, ver o nome do entregador e a localização GPS atual dele através do sistema.
- **Parâmetros de Entrada:**
 - pedidoID (String): Identificador único do pedido
- **Retorno (Saída):**
 - Situação atual do pedido (ex: "a caminho", "entregue").
 - dadosEntregador
 - coordenadasGPS
- **Pré-condições:**
 - O cliente deve estar autenticado.
 - O pedido já deve ter sido confirmado e estar em fase de entrega.
 - O sistema deve ter dados de localização do entregador atualizados.
- **Pós-condições:**
 - O cliente verá no app/site o status atualizado do pedido e um mapa com o entregador em tempo real.
- **Exceções:**
 - PedidoNaoEncontradoException: O ID do pedido informado não existe.
 - EntregadorIndefinidoException: O pedido ainda não foi atribuído a um entregador.
 - ServicoGPSIndisponivelException: Falha ao recuperar localização do entregador no sistema externo.

Diagramas - DSS



Contrato da Operação: gerenciarCardapio()

- **Nome:** gerenciarCardapio()
- **Descrição:** Essa operação permite ao dono do restaurante gerenciar o cardápio, incluindo adicionar, editar ou remover itens, definir a taxa do aplicativo e configurar a entrega grátis opcional para determinados produtos, assumindo o custo da entrega quando necessário.
- **Parâmetros de Entrada:**
 - itemId (opcional) (String): Identificador do item (em caso de edição ou remoção).
 - nomeItem (String): Nome do item do cardápio.
 - descricaoItem (String): Descrição do item.
 - precoBase (Float): Preço inicial definido pelo restaurante.
 - entregaGratis (Boolean): Indica se o item terá entrega grátis
 - taxaAplicativo (opcional) (Float): Percentual ou valor fixo da taxa do app aplicada ao preço final.
- **Retorno (Saída):**
 - statusOperacao (String): Mensagem indicando sucesso ou falha na atualização do cardápio.
 - precoFinalCliente (Float): Valor final do item com a taxa do aplicativo aplicada.
- **Pré-condições:**
 - O dono do restaurante deve estar autenticado no sistema.
 - O restaurante deve estar previamente cadastrado e ativo.
- **Pós-condições:**
 - O cardápio é atualizado conforme as ações realizadas.
 - Os itens com entrega grátis terão o custo adicionado ao restaurante.

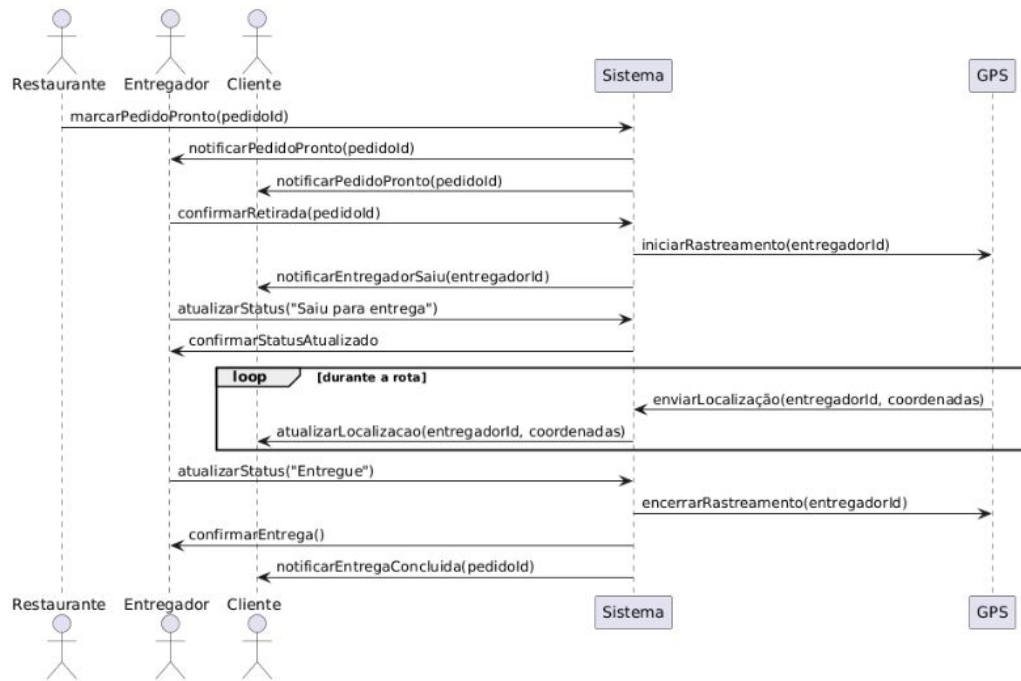
Diagramas - DSS



Contrato da Operação: processarEntrega()

- **Nome:** processarEntrega()
- **Descrição:** Essa operação permite ao entregador aceitar um pedido, buscar no restaurante e entregar ao cliente, atualizando o status da entrega em cada etapa do processo.
- **Parâmetros de Entrada:**
 - Não possui parâmetros diretos, pois todas as ações são feitas com base no entregador autenticado e pedidos disponíveis no sistema.
- **Retorno (Saída):**
 - Pedido aceito: "Aceito"
 - Em rota para restaurante: "Em rota para restaurante"
 - Pedido retirado: "Pedido retirado"
 - Em rota para entrega: "Em rota para entrega"
 - Pedido entregue: "Entregue"
- **Pré-condições:**
 - O entregador deve estar autenticado no sistema.
 - Deve haver pelo menos um pedido disponível para entrega.
- **Pós-condições:**
 - O pedido é marcado como entregue no sistema.
 - O status do pedido é atualizado ao longo das etapas.
 - O sistema finaliza a entrega com sucesso.

Diagramas - DSS



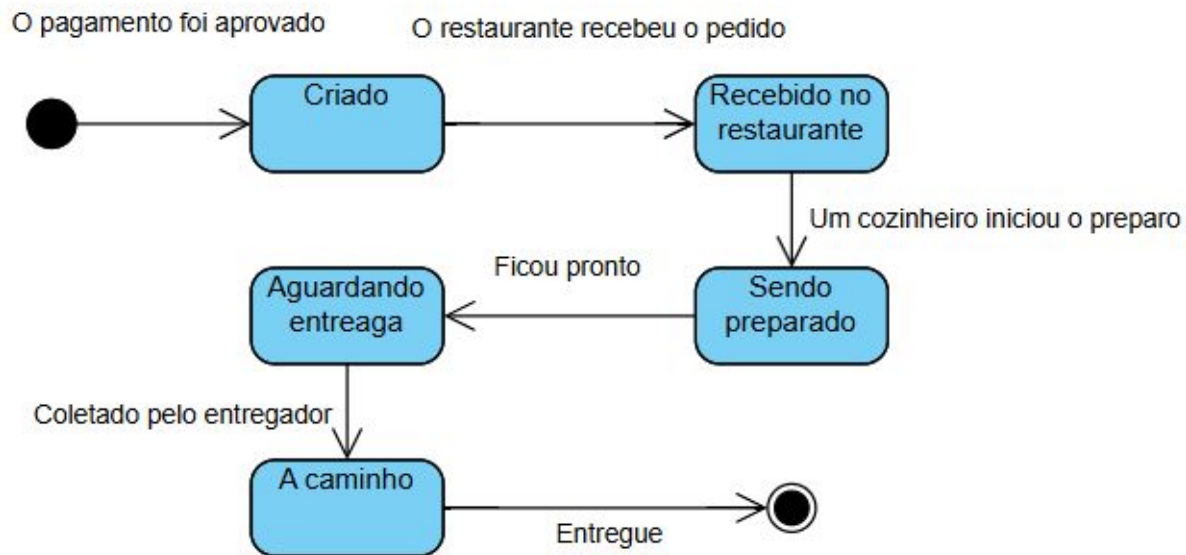
Contrato da Operação:

gerenciarEntregaComRastreamento(pedidoId)

- **Nome:**
gerenciarEntregaComRastreamento(pedidoId)
- **Descrição:** O sistema gerencia o processo completo de entrega, desde a notificação do restaurante que o pedido está pronto, passando pela aceitação e retirada pelo entregador, rastreamento em tempo real via GPS, até a confirmação da entrega ao cliente.
- **Parâmetros de Entrada:**
 - pedidoId (String): Identificador único do pedido
- **Retorno (Saída):**
 - statusEntrega (String): Status atual do pedido (ex: "Pronto", "Saiu para entrega", "Entregue")
 - localizacaoEntregador (Objeto): Dados GPS atuais do entregador (latitude, longitude)
 - notificacoesCliente (String): Mensagens enviadas ao cliente sobre o status da entrega

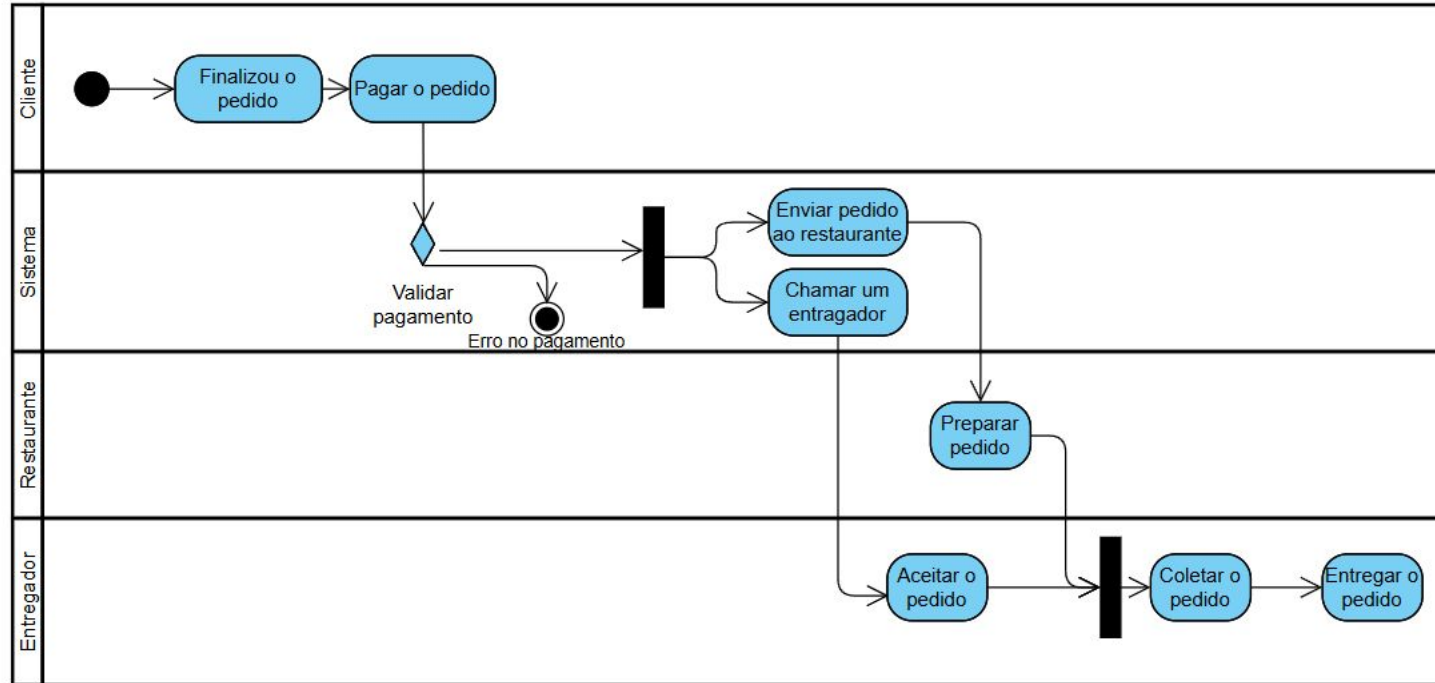
Diagrama - Estados

Pedido



Diagramas - Atividades

Pedido



Diagramas - Classes

