### 1. Visão Geral do Sistema

O sistema de Ordem de Serviço (OS) foi projetado para gerenciar o fluxo completo de reparo de equipamentos eletrônicos, desde a abertura do chamado até a sua finalização. A arquitetura modular separa as responsabilidades entre diferentes atores (Atendente, Administrador, Técnico e Cliente) e utiliza padrões de projeto para garantir flexibilidade, manutenibilidade e robustez.

### 2. Arquitetura e Padrões de Projeto

A arquitetura foi construída sobre três padrões de projeto fundamentais: **State**, **Template Method** e **Singleton**.

#### 2.1. Padrão State: Gerenciando o Fluxo da OS (Kanban)

**Problema:** Uma OrdemServico passa por diversos estágios (Aberta, Aguardando Aprovação, Em Reparo, Finalizada, etc.). O comportamento esperado da OS (quais ações são permitidas) muda drasticamente em cada estágio. Implementar essa lógica com condicionais (if/else) dentro da classe OrdemServico a tornaria complexa e difícil de manter.

**Solução:** O padrão **State** foi aplicado para encapsular o comportamento de cada estágio em sua própria classe.

**EstadoOS (Interface):** Define um contrato comum para todos os estados, declarando métodos como abrir(), aprovar(), finalizar(), etc.

**Classes de Estado Concretas (EstadoAberta, EstadoAguardandoAprovacao, etc.):** Cada classe implementa a interface EstadoOS. Ela contém a lógica específica para aquele estado. Por exemplo, na classe EstadoAguardandoAprovacao, o método aprovar() fará a transição para o estado EstadoAprovada, enquanto o método finalizar() lançará uma exceção, pois essa ação é inválida nesse estágio.

**OrdemServico (Contexto):** A classe OrdemServico mantém uma referência ao seu objeto de estado atual (estadoAtual). Quando um método como ordemDeServico.aprovar() é chamado, ele delega a chamada para o objeto de estado atual: estadoAtual.aprovar(this). Isso desacopla a lógica de negócio do fluxo de estados.

#### 2.2. Padrão Template Method: Padronizando o Processo de Reparo

**Problema:** O processo de reparo de diferentes tipos de hardware (Notebook, Celular, Computador) segue um fluxo de passos semelhante (diagnóstico, reparo, testes, limpeza), mas a execução de cada passo é diferente. Queremos evitar a duplicação de código da estrutura do algoritmo e forçar um passo a passo consistente para o técnico.

**Solução:** O padrão **Template Method** define o "esqueleto" de um algoritmo em uma superclasse, permitindo que subclasses redefinam certos passos sem alterar a estrutura do algoritmo.

**TemplateReparo (Classe Abstrata):** Contém o método executarProcessoDeReparo(), que é final para que não possa ser sobrescrito. Este método chama uma sequência de outros métodos protected que representam os passos do algoritmo (ex: diagnosticarProblema(), realizarReparoPrincipal()).

**Métodos Abstratos e Ganchos:** Alguns passos são declarados como abstract (ex: realizarReparoPrincipal), forçando as subclasses a implementá-los. Outros (chamados de "hooks" ou ganchos) podem ter uma implementação padrão (ex: limpezaFinal()), que as subclasses podem ou não sobrescrever.

**Classes Concretas (ReparoNotebook, ReparoCelular):** Herdam de TemplateReparo e implementam apenas os passos abstratos, fornecendo os detalhes específicos para cada tipo de hardware. O Técnico então utiliza a implementação concreta apropriada para a OrdemServico em questão.

#### 2.3. Padrão Singleton: Garantindo uma Única Conexão com o Banco de Dados

**Problema:** Abrir e fechar conexões com o banco de dados são operações custosas. Em toda a aplicação, precisamos de um ponto de acesso central e único para o banco, garantindo que não existam múltiplas conexões desnecessárias sendo criadas.

**Solução:** O padrão **Singleton** garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global a ela.

**DatabaseConnector (Classe Singleton):**

* Possui um construtor private para impedir que seja instanciada diretamente.
* Tem um atributo estático e privado que armazena a única instância (private static DatabaseConnector instance).
* Fornece um método estático público getInstance() que retorna a instância única. Este método cria a instância na primeira vez que é chamado e a retorna em todas as chamadas subsequentes.
* A implementação utiliza "double-checked locking" para garantir a segurança em ambientes com múltiplas threads.