Projeto Cidade Inteligente: Iluminação Pública

APLICAÇÃO DE PROGRAMAÇÃO ORIENTADA A OBJETOS E PADRÕES DE PROJETO

ALUNOS: GUILHERME SANTIAGO, THAYLINY A. MOURA

O Problema e a Solução

Cidades Inteligentes e Sustentabilidade

Como otimizar o uso da iluminação pública para economizar energia e aumentar a eficiência.

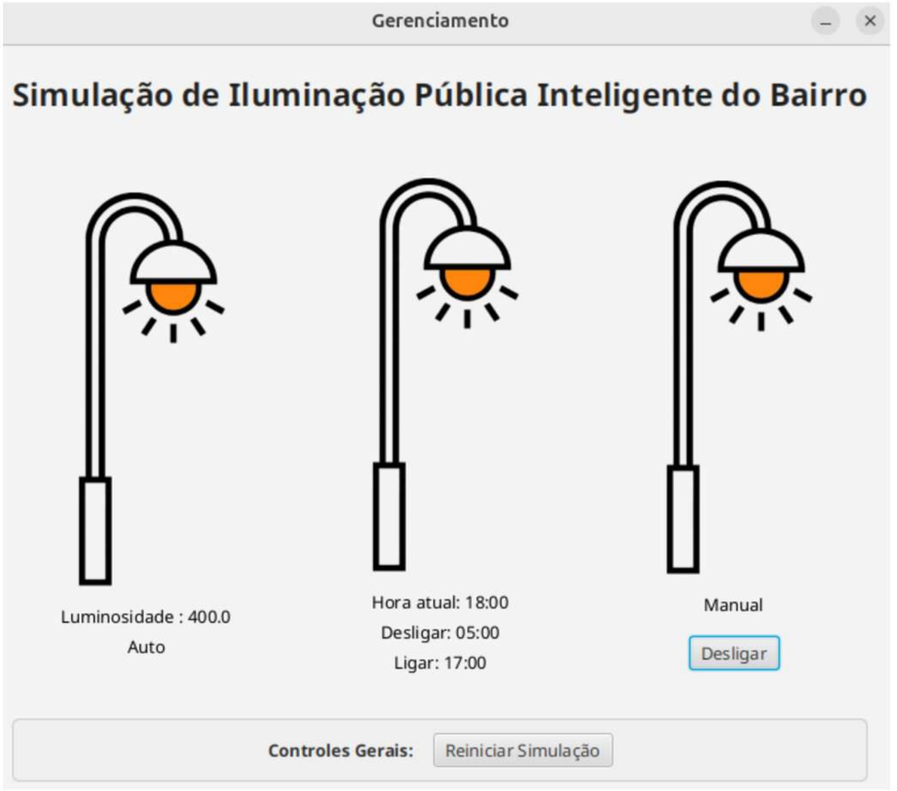
Objetivo da Implementação:

- Simular um sistema de iluminação pública inteligente.
- Demonstrar a aplicação prática de conceitos de Programação Orientada a Objetos.
- Utilizar Padrões de Projeto para criar um sistema flexível, modular e extensível.
- Construir uma interface gráfica interativa para visualização da simulação.

Visão Geral do Sistema

Visão Geral do Sistema

Como Funciona :



Temos uma representação visual de postes de luz. Clicamos no botão para interagir, e o sistema simula o comportamento dos postes.

Conceitos Fundamentais de POO Utilizados

Visão Geral do Sistema

Classes e Objetos:

- Cada poste de luz (**PosteDeLuz**) é um objeto com seu próprio estado (**ligado/desligado**) e comportamento. Outras classes importantes incluem **Lampada, Sensor e ModuloComunicacao.**

Herança

- Reutilizamos código e estendemos funcionalidades. Por exemplo, nossos Decorators (**LampadaComDimmer**) estende de **LampadaDecorator**.

Encapsulamento:

- Garantimos a integridade dos dados, mantendo os atributos privados e expondo apenas o necessário via métodos públicos (getters/setters). Isso é evidente na classe **PosteDeLuz** .

Polimorfismo:

- Objetos de tipos diferentes podem ser tratados de forma uniforme. Uma variável do tipo Lampada pode referenciar **LampadaLED** ou **LampadaVaporSodi**o, e o **método ligar()** se comporta de maneira específica para cada uma, demonstrando polimorfismo.

Padrão de Projeto

Abstract Factory

Abstract Factory (Padrão de Criação)

Este padrão é ideal para criar famílias de objetos relacionados sem especificar suas classes concretas. Na iluminação pública, podemos ter diferentes "tipos" de postes de luz (por exemplo, postes com lâmpadas LED, postes com lâmpadas de vapor de sódio, postes"inteligentes" com sensores, etc.). Cada tipo de poste pode ter componentes diferentes (lâmpadas, sensores, módulos de comunicação).

Por que escolhemos:

- Flexibilidade! Podemos ter diferentes 'linhas de produção' de postes (LED, Vapor de Sódio) com conjuntos de componentes compatíveis.
- Facilita a adição de novos tipos de tecnologia no futuro sem mexer no código principal do poste.

Exemplo no Projeto:

- FabricaPoste (interface da fábrica)
- FabricaPosteLED (produz lâmpadas LED, sensores de presença, módulos Wi-Fi)
- FabricaPosteVaporSodio (produz lâmpadas de vapor de sódio, sem sensor/módulo)

Observer

Observer (Comportamental)

O padrão Observer é ótimo para situações onde um objeto (o "assunto") precisa notificar outros objetos (os "observadores") sobre mudanças em seu estado, sem saber exatamente quem são esses observadores. O observer é aplicado no comportamento dos postes.

Por que escolhemos:

- Monitoramento em tempo real. Um poste pode avisar a diversos 'centros' (painel de controle, sistema de log) quando ele liga ou desliga.
- Desacoplamento: O poste não precisa saber quem está observando suas mudanças, apenas que existem observadores que precisam ser notificados.

Exemplo no Projeto:

- **PosteDeLuz** (o assunto) notifica
- **PainelDeControle** e **CentroDeMonitoramento** (os observadores) sobre mudanças de estado.

Strategy

Strategy (Comportamental)

Este padrão permite definir uma família de algoritmos, encapsular cada um deles e torná-los intercambiáveis. O Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam. No nosso projeto usamos o Strategy para definir comportamentos configuráveis para os postes.

Por que escolhemos:

- Flexibilidade no controle! Um poste pode ter diferentes lógicas de ligamento/desligamento (manual, por horário, por luminosidade).
- É fácil adicionar novas estratégias (ex: por detecção de movimento) sem modificar a classe **PosteDeLuz**.

Exemplo no Projeto:

- ComportamentoLigamento (interface da estratégia).
- **EstrategiaManual**, **EstrategiaPorHorario**, **EstrategiaPorLuminosidade** (implementações).
- O **PosteDeLuz** recebe e executa a estratégia configurada.

Decorator

Decorator (Estrutural)

Este padrão permite adicionar novas responsabilidades a um objeto dinamicamente. Os decorators fornecem uma alternativa flexível à herança para estender funcionalidades. No nosso projeto ele adiciona funcionalidades dinamicamente às lâmpadas.

Por que escolhemos:

- Evita explosão de classes! Em vez de criar **LampadaLEDComDimmer**, **LampadaLEDComContador**, podemos combinar funcionalidades.
- Funcionalidades podem ser adicionadas ou removidas em tempo de execução.

Exemplo no Projeto:

- **Lampada** (o componente base)
- **LampadaDecorator** (o decorator abstrato)
- **LampadaComDimmer** (adiciona controle de intensidade)
- **LampadaComContador** (adiciona contagem de ciclos)
- Podemos criar uma **LampadaComDimmer(new LampadaComContador(new LampadaLED())).**

Outros Conceitos Importantes

Robustez e Interatividade

- **Interfaces:** Usadas extensivamente para definir contratos e promover o polimorfismo e o baixo acoplamento entre os módulos (Lampada, Sensor, FabricaPoste, SujeitoPoste, etc.).

- **Tratamento de Exceções:** Nosso código inclui blocos try-catch para lidar com situações inesperadas, como IOException ao carregar recursos FXML ou arquivos de propriedades. Também inicializamos coleções para evitar NullPointerExceptions.

Classes Abstratas: Utilizamos **LampadaDecorator** como uma classe abstrata para fornecer uma implementação padrão para o comportamento de delegar à lâmpada decorada.

Robustez e Interatividade

- **Leitura de Arquivos:** A classe **ConfiguracaoLuzes** lê o arquivo **luzes.properties** para carregar configurações de simulação (horários, limiares de luminosidade), tornando o sistema configurável.

Threads: Utilizamos uma Thread separada no BairroController para simular a passagem do tempo e permitir que as estratégias dos postes sejam avaliadas periodicamente sem bloquear a interface do usuário, garantindo a fluidez da aplicação.

Obrigada!