Sistema de Monitoramento de Salas Inteligentes

Hildemar Lemos de Santana Júnior, Kleberson de Jesus Sousa e Thiago Sampaio Santos

Diagnóstico arquitetural da versão sem padrões

A versão inicial do sistema de monitoramento de salas inteligentes foi propositalmente construída com uma arquitetura frágil, com o objetivo de evidenciar falhas recorrentes em projetos que ignoram boas práticas de engenharia de software. A seguir, são destacados os principais pontos críticos:

1 - Trechos críticos identificados:

- Classe Sala: Centraliza múltiplas responsabilidades como coleta de dados, tomada de decisão e controle de sensores. Essa concentração viola o princípio da responsabilidade única (SRP), tornando a classe difícil de testar, manter e evoluir.
- Instanciação direta de sensores: O uso de new SensorTemperatura() dentro da classe Sala cria um acoplamento rígido entre componentes, dificultando substituições e testes isolados. Isso infringe o princípio da inversão de dependência (DIP), pois depende de implementações concretas.
- Ausência de interfaces: A falta de abstrações impede a reutilização de código e dificulta a substituição de componentes, como sensores, por versões alternativas ou simuladas.
- Método monitorar(): Agrupa coleta de dados, lógica de decisão e execução de ações em um único método. Essa mistura de responsabilidades compromete a legibilidade e a testabilidade do código.
- Classe Relatorio: Acessa diretamente atributos públicos da classe Sala, quebrando o encapsulamento e expondo detalhes internos que deveriam ser protegidos.

2 - Violações dos Princípios SOLID:

- SRP (Responsabilidade Única): A classe Sala acumula funções distintas, dificultando sua manutenção.
- OCP (Aberto/Fechado): Qualquer adição de funcionalidade exige alteração direta no código existente, o que compromete a extensibilidade.
- DIP (Inversão de Dependência): O sistema depende de classes concretas, o que limita a flexibilidade e dificulta testes automatizados.

Justificativa para cada padrão aplicado

Durante a refatoração, foram aplicados padrões de projeto dos três grupos GOF (Criação, Estrutural e Comportamental), com o objetivo de modularizar o sistema, reduzir acoplamento e facilitar sua evolução.

Factory (Criação)

- Classe: SensorFactory
- Justificativa: Permite instanciar sensores de forma flexível e desacoplada, facilitando testes unitários e substituições futuras sem alterar o código cliente. A fábrica foi integrada diretamente à fachada, eliminando a dependência de instâncias concretas.

Decorator (Estrutural)

- Classes: SensorDecorator, LoggingSensorDecorator
- Justificativa: Adiciona funcionalidades como registro de eventos (logging) sem modificar a lógica original dos sensores. O padrão foi aplicado diretamente na execução principal, evidenciando sua utilidade prática.

Facade (Estrutural)

- Classe: MonitoringFacade
- Justificativa: Fornece uma interface simplificada para o sistema, escondendo a complexidade interna e facilitando o uso por outras camadas. A fachada foi aprimorada com o uso da fábrica e do coletor de dados.

Observer (Comportamental)

- Classes: SensorSubject, SensorObserver, ActionTrigger
- Justificativa: Permite que componentes reajam automaticamente a mudanças nos sensores, promovendo desacoplamento entre produtor e consumidor de eventos.

Strategy (Comportamental)

- Classes: ActionStrategy, LightOnStrategy, ReportStrategy
- Justificativa: Encapsula diferentes comportamentos de resposta, permitindo que o sistema varie suas ações sem alterar sua estrutura.

Descrição dos anti-padrões detectados e como foram eliminados

Na versão inicial, foram identificados dois antipadrões clássicos que comprometiam a qualidade do sistema:

God Object

- Problema: A classe Sala concentrava diversas responsabilidades, tornando-se um ponto único de falha e dificultando a manutenção.
- Solução: As responsabilidades foram distribuídas entre classes especializadas como Sensor, GeradorRelatorio, MonitoringFacade e ColetorDadosSensor, promovendo coesão e modularidade.

Lava Flow

- Problema: Existência de código redundante e não reutilizável nos sensores e na lógica de decisão.
- Solução: A criação da SensorFactory e a adoção dos padrões Strategy e Decorator permitiram encapsular comportamentos e eliminar duplicações.

Outras melhorias implementadas

- Encapsulamento de atributos com métodos de acesso, protegendo o estado interno das classes.
- Extração de métodos para modularizar a lógica e facilitar testes.
- Substituição de código repetitivo por polimorfismo, utilizando a interface Sensor como abstração.
- Criação da classe ColetorDadosSensor para centralizar a coleta de dados, promovendo reutilização e separação de responsabilidades.

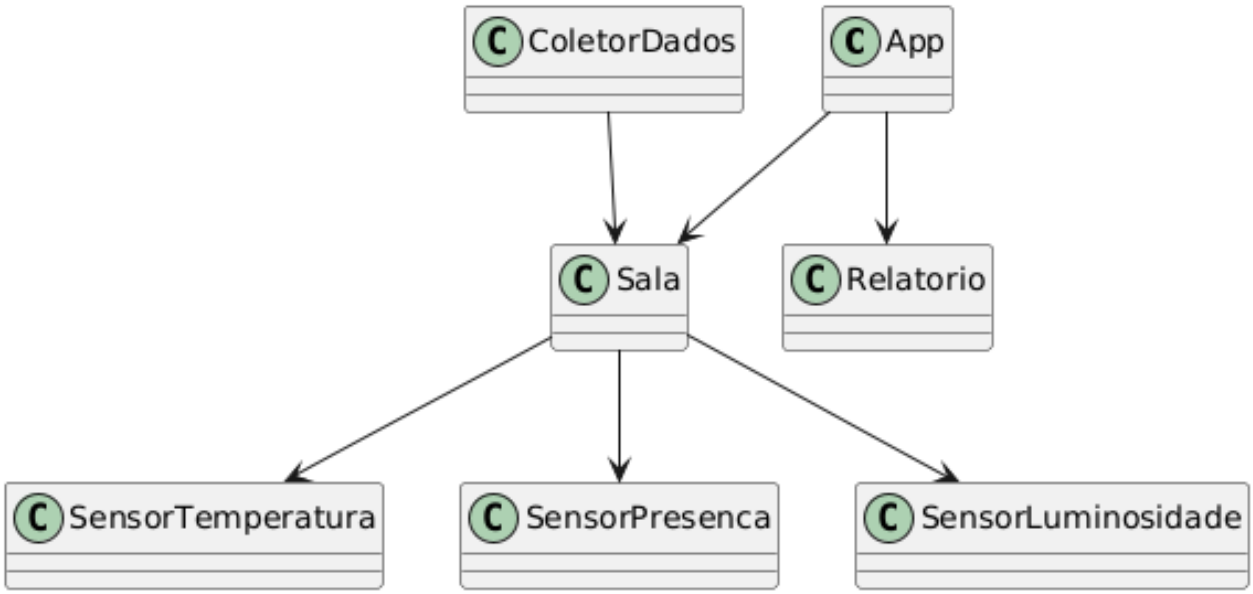
UML antes/depois (com codificação de criação)

UML da implementação inicial sem Padrões

```
@startuml
class App
class Sala
class SensorTemperatura
class SensorPresenca
class SensorLuminosidade
class Relatorio
class ColetorDados

App --> Sala
Sala --> SensorTemperatura
Sala --> SensorPresenca
Sala --> SensorLuminosidade
App --> Relatorio
ColetorDados --> Sala
@enduml
```

Imagem do diagrama da implementação inicial



UML da arquitetura refatorada

```
@startuml
package app {
  class App
}
package facade {
  class MonitoringFacade
}
package model {
  interface Sensor
  class SensorTemperatura
  class SensorPresenca
  class SensorLuminosidade
}
package factory {
  class SensorFactory
}
package observer {
  interface SensorObserver
  class SensorSubject
  class ActionTrigger
}
package strategy {
  interface ActionStrategy
  class LightOnStrategy
  class ReportStrategy
}
package decorator {
  abstract class SensorDecorator
  class LoggingSensorDecorator
}
package report {
  class GeradorRelatorio
}
App --> MonitoringFacade
MonitoringFacade --> SensorTemperatura
MonitoringFacade --> SensorPresenca
MonitoringFacade --> SensorLuminosidade
MonitoringFacade --> GeradorRelatorio
SensorFactory --> Sensor
SensorTemperatura ..|> Sensor
SensorPresenca ..|> Sensor
SensorLuminosidade ..|> Sensor
@enduml
```

Imagem do diagrama da implementação inicial

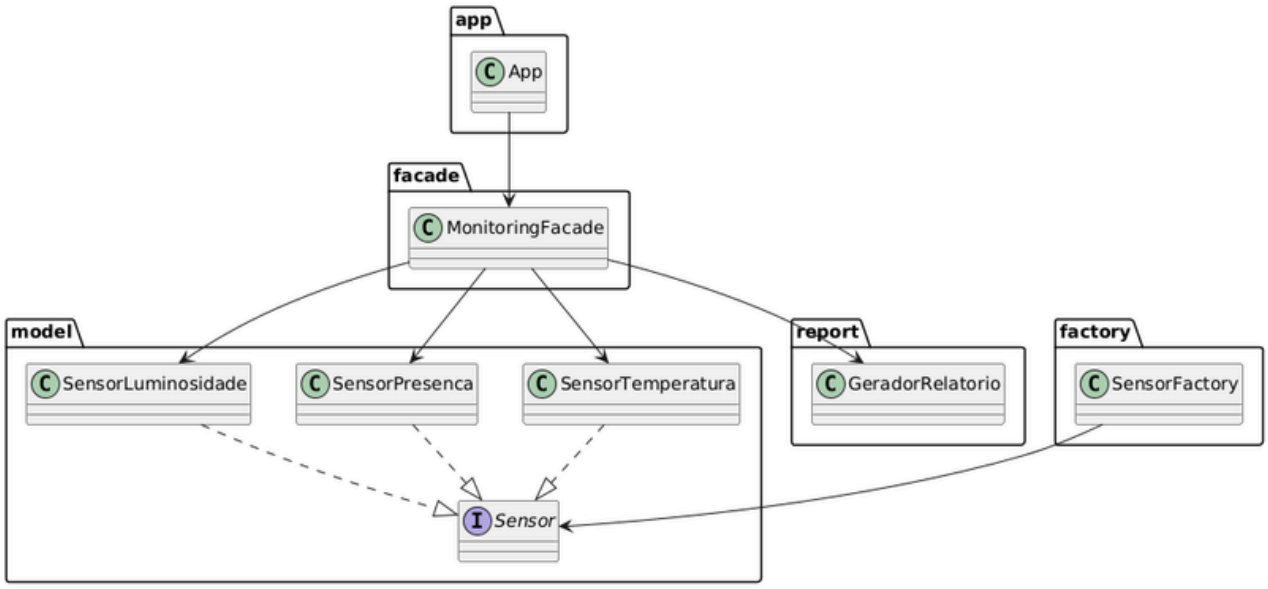


Tabela de Refatorações:

Refatoração Aplicada	**Impacto no Sistema**
Extração de métodos	Melhora a legibilidade do código, facilita testes unitários e promove reutilização.
Criação da interface Sensor	Permite o uso de polimorfismo, reduz acoplamento e facilita substituição de sensores.
Aplicação do padrão Factory	Centraliza a lógica de criação de objetos, tornando o sistema mais flexível e testável.
Uso do padrão Decorator	Adiciona funcionalidades (como logging) sem alterar as classes originais, promovendo extensão.
Introdução do padrão Facade	Simplifica o uso do sistema ao oferecer uma interface única e coesa para operações complexas.
Implementação do padrão Observer	Permite que componentes reajam automaticamente a mudanças, promovendo modularidade e desacoplamento.
Estratégias com o padrão Strategy	Encapsula comportamentos variáveis, tornando o sistema mais extensível e configurável.