



**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA  
BAHIA CAMPUS - SANTO ANTÔNIO DE JESUS - BAHIA  
CURSO DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE  
SISITEMAS**

**MARCELO DE JESUS**

**RONALDO CORREIA COUTO**

**FRANKLIN FÉLIX DE OLIVEIRA**

**Criação e Evolução de um Sistema Mal Projetado  
com Aplicação Guiada de Padrões**

**SANTO ANTONIO DE JESUS – BA  
2025**

**MARCELO DE JESUS**  
**RONALDO CORREIA COUTO**  
**FRANKLIN FÉLIX DE OLIVEIRA**

# **Criação e Evolução de um Sistema Mal Projetado com Aplicação Guiada de Padrões**

Pesquisa elaborada como  
requisito parcial de avaliação para  
a disciplina de Padrões  
de Projeto, ministrada pelo Prof.  
Felipe Silva

**SANTO ANTONIO DE JESUS – BA**

**2025**

## 1. INTRODUÇÃO

Este relatório tem como objetivo apresentar a importância de como um software deve ter decisões arquiteturais, tendo em vista que impactam diretamente na qualidade, manutenibilidade e evolução futura do sistema. É comum, haver softwares que mesmo mal estruturados consigam até atender às necessidades imediatas do cliente, mas que, no entanto, com códigos acoplados, de difícil manutenção e com baixa coesão que pode surgir desafios e problemas posteriores.

Para compreender as consequências acima citadas, na prática, a atividade proposta tem como finalidade desenvolver um sistema inicialmente sem arquitetura definida, com problemas visíveis de alto acoplamento em sua classe, forte quebra dos conceitos dos padrões GOF e SOLID. O sistema escolhido para essa análise foi um monitor de salas inteligentes, responsável por funções básicas como: Cadastros de sensores (Temperatura, presença luminosidade), Coleta periódica de dados, Acionamento de ações simples, por exemplo, acender a luz quando presença=true e Geração de relatórios.

A elaboração do software, de acordo com a solicitação da atividade, foi subdividida em 3 fases, onde na primeira fase o sistema foi desenvolvido sem padrões, sem uso de interfaces, forte acoplamento e ausência de modularidade, no entanto, funcional, apenas com intuito de expor problemas de coesão, acoplamento e violações dos princípios SOLID. Já na segunda fase, identificando trechos problemáticos, violações de princípios SOLID, presença de anti-padrões como God Object e Spaghetti Code e através desse relatório trazer o diagrama de UML do código sem arquitetura, da primeira fase. Enquanto na terceira fase, o sistema foi reestruturado com a aplicação de padrões de projeto dos grupos de Criação (Factory Method), Estrutural (Facade/Decorator) e Comportamental (Observer e Strategy)

Dessa forma, com a nova versão, bem como também a inicial, será possível evidenciar problemas e riscos de um projeto mal projetado, em como a adoção de boas práticas interfere diretamente para o bom funcionamento a longo prazo do software tornando-os mais organizados, extensíveis e de fácil manutenção.

## **2. DESENVOLVIMENTO**

### **2.1 Implementação inicial (arquitetural da versão sem padrões)**

Na primeira fase da atividade, foi desenvolvido um Sistema de Monitoramento, sem adoção de padrões, utilizando apenas uma classe, classe essa que é responsável por todas funcionalidade do sistema, como cadastros, coleta de dados e geração de relatório, grande quantidade de If/Else encadeados para atender a lógica dos sensores e diferenciá-los entre si, a parti daí é evidente responsabilidades misturadas em uma única classe violando todos os princípios de responsabilidade única SRP.

Outra característica evidente no código, está relacionado ao forte acoplamento, se caso for incluir novos sensores exigira a modificação de toda a classe além de que os sensores estão apenas representando por variáveis do tipo String

Na primeira fase do trabalho, foi desenvolvido o Sistema Monitoramento de forma propositalmente mal projetada, concentrando todas as funcionalidades em uma única classe.

Exemplo no código da fase 1:

The screenshot shows an IDE with a project named 'EVOLUCAO-DE-SISTEMA-MAL-PROJET...'. The left sidebar displays the project structure, including a package 'fase1' containing 'sistemaMonitoramento.java'. The main editor area shows the code for 'sistemaMonitoramento.java' in the package 'main.java.com.fase1'. The code includes imports for 'ArrayList' and 'List', a private list of sensors, and methods for registering sensors, collecting data, and generating a report.

```
src > main > java > com > fase1 > J sistemaMonitoramento.java > {} main.java.com.fase1
1 package main.java.com.fase1;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class sistemaMonitoramento {
7
8     private List<String> sensores = new ArrayList<>();
9
10    public void cadastrarSensor(String tipo) {
11        sensores.add(tipo);
12        System.out.println("Sensor cadastrado: " + tipo);
13    }
14
15    public void coletarDados() {
16        for (String s : sensores) {
17            if (s.equals(anObject:"temperatura")) {
18                System.out.println(x:"Coletando temperatura: 25°C");
19            } else if (s.equals(anObject:"presenca")) {
20                System.out.println(x:"Detectando presença: true");
21                System.out.println(x:">>> Acionando luz");
22            } else if (s.equals(anObject:"luminosidade")) {
23                System.out.println(x:"Luminosidade: 300 lux");
24            }
25        }
26    }
27
28    public void gerarRelatorio() {
29        System.out.println(x:"Relatório:");
30        for (String s : sensores) {
31            System.out.println("Sensor: " + s + " - Dados coletados...");
32        }
33    }
34
35
36 }
37
```

## 2. 2 Diagnóstico arquitetural da versão sem padrões

### 2.2.1. Trechos Críticos do Código

Trecho 1:

```
public void coletarDados() {
    for (String s : sensores) {
        if (s.equals("temperatura")) {
            System.out.println("Coletando temperatura: 25°C");
        } else if (s.equals("presenca")) {
            System.out.println("Detectando presença: true");
            System.out.println(">>> Acionando luz");
        } else if (s.equals("luminosidade")) {
            System.out.println("Luminosidade: 300 lux");
        }
    }
}
```

- **Problema:** O método coletarDados vem com encadeamento gigante de if/else para diferenciar os sensores por Strings, tornando por exemplo a lógica de atuação (Acionando luz) misturada com coleta de dados, violando a coesão, onde uma função faz mais de uma coisa.
- **Justificativa:** Torna o código difícil para testes e expansão/atualização sem quebrar outras funcionalidades. Importante ressaltar que por exemplo, com a criação de um novo sensor, exige a edição da classe, violando assim os princípios da OCP (Open/Closed Principle)

### Trecho 2:

```
} else if (s.equals("luminosidade")) {  
    System.out.println("Luminosidade: 300 lux");  
}
```

- **Problema:** impressão direta no console dentro da classe Sensor.
- **Justificativa:** mistura de lógica de negócio com interface, prejudicando reutilização e manutenção.

### 2.2.2 Violação do SOLID

A classe SistemaMonitoramento tem a responsabilidade de fazer três processos ao mesmo tempo, cadastrar sensores, coleta dados e gerar relatórios, com essa mistura de responsabilidades a mesma acaba violado diretamente um dos princípios SOLID, a Single Responsibility Principle, que é um conceito fundamental da orientação a objetos que afirma que uma classe ou módulo deve ter apenas uma responsabilidade, ou seja, um único motivo para mudar, de acordo com Robert C. Martin (Uncle Bob) em seu livro Clean Code, uma função deve ter apenas uma responsabilidade. Como correção da problemática, seria dividir as responsabilidades ou funções em classes separadas, com interfaces e classes para sua implementação, por exemplo:

- SensorManager → cadastro de sensores
- ColetorDados → coleta de dados
- Relatório → geração e apresentação dos dados

Além da violação do princípio SRP, podemos identificar também através do método coletarDados, a violação do princípio Open/Closed Principle (OCP), onde propõe que entidades (classes, funções, módulos, etc.) devem ser abertas para extensão, mas fechadas para modificação, que o mesmo não é aberto para extensão, por exemplo para incluir um novo método (Ex: "umidade"), é necessário modificar o código fonte, como resolução, poderia utilizar polimorfismo, assim cada sensor saberia como coletar seus próprios dados. E o princípio DIP, Dependency Inversion Principle, na qual o princípio de Inversão de Dependência possui duas definições:

- (1) módulos de alto nível não devem depender de módulos de baixo nível e ambos devem depender de abstrações;
- (2) abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações.

Com isso é perceptível no código a dependência direta de Strings literais que representam sensores, criando assim dependência de baixo nível, onde o correto seria depender de uma abstração (Sensor) e não de detalhes (String+if/else).

Lembrando que com essas violações, temos como impacto enorme a dificuldade de testes, além da dificuldade de expandir novas funcionalidades sem quebrar ou prejudicar outras partes do sistema, já que a lógica de negócios está em apenas um único lugar.

### 2.2.3 Identificação do Anti-padrão

A classe sistemaMonitoramento concentra diversas responsabilidades e controla toda a lógica do sistema, faz cadastros, coleta e gera relatórios, há forte acoplamento entre as funcionalidades e dificulta a manutenção bem como a evolução, isso acaba gerando um sistema rígido, frágil e de difícil reuso, além de ter métodos longos. Com isso foi identificado alguns Anti-Padrões como:

- **Spaghetti Code** → condicional excessiva em coletarDados().
- **God Class (Classe Deus)** → Classe se torna excessivamente grande e complexa, acumulando muitas responsabilidades e centralizando grande parte da funcionalidade de um sistema.
- **Primitive Obsession** → uso de String para representar sensores.
- **Shotgun Surgery** → cada novo sensor exige mudanças em vários pontos.

- **Hard-Coding** → valores fixos no código, nada dinâmico.

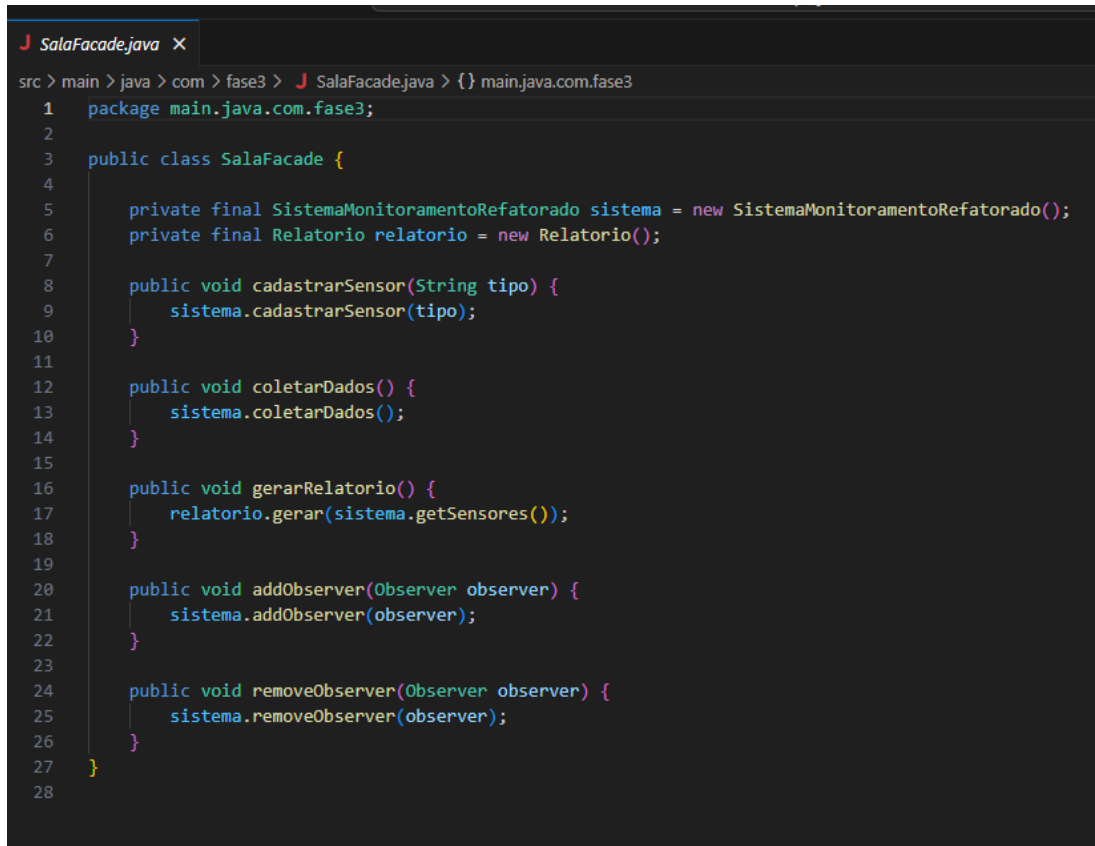
#### 2.2.4 UML Inicial

```
+-----+
|  sistemaMonitoramento|
+-----+
| - sensores: List<String> |
+-----+
| + cadastrarSensor()  |
| + coletarDados()      |
| + gerarRelatorio()    |
+-----+
```



### 3. JUSTIFICATIVA PARA CADA PADRÃO APLICADO

- **Padrão Facade** aplicado na classe SalaFacade:



```
J SalaFacade.java X
src > main > java > com > fase3 > J SalaFacade.java > {} main.java.com.fase3
1 package main.java.com.fase3;
2
3 public class SalaFacade {
4
5     private final SistemaMonitoramentoRefatorado sistema = new SistemaMonitoramentoRefatorado();
6     private final Relatorio relatorio = new Relatorio();
7
8     public void cadastrarSensor(String tipo) {
9         sistema.cadastrarSensor(tipo);
10    }
11
12    public void coletarDados() {
13        sistema.coletarDados();
14    }
15
16    public void gerarRelatorio() {
17        relatorio.gerar(sistema.getSensores());
18    }
19
20    public void addObserver(Observer observer) {
21        sistema.addObserver(observer);
22    }
23
24    public void removeObserver(Observer observer) {
25        sistema.removeObserver(observer);
26    }
27 }
28
```

- **Objetivo:** fornecer uma interface simplificada para o sistema de sensores, escondendo a complexidade da coleta de dados e geração de relatórios.
- **Benefício:** reduz o acoplamento entre a interface e o sistema principal, facilita testes e manutenção, e organiza melhor o código.
- **Implementação:** criamos a classe Relatorio para separar a apresentação da lógica de negócio, integrando o padrão Facade (estrutural) com princípios de boa coesão.

#### 4. DESCRIÇÃO DOS ANTI-PADRÕES DETECTADOS E COMO FORAM ELIMINADOS

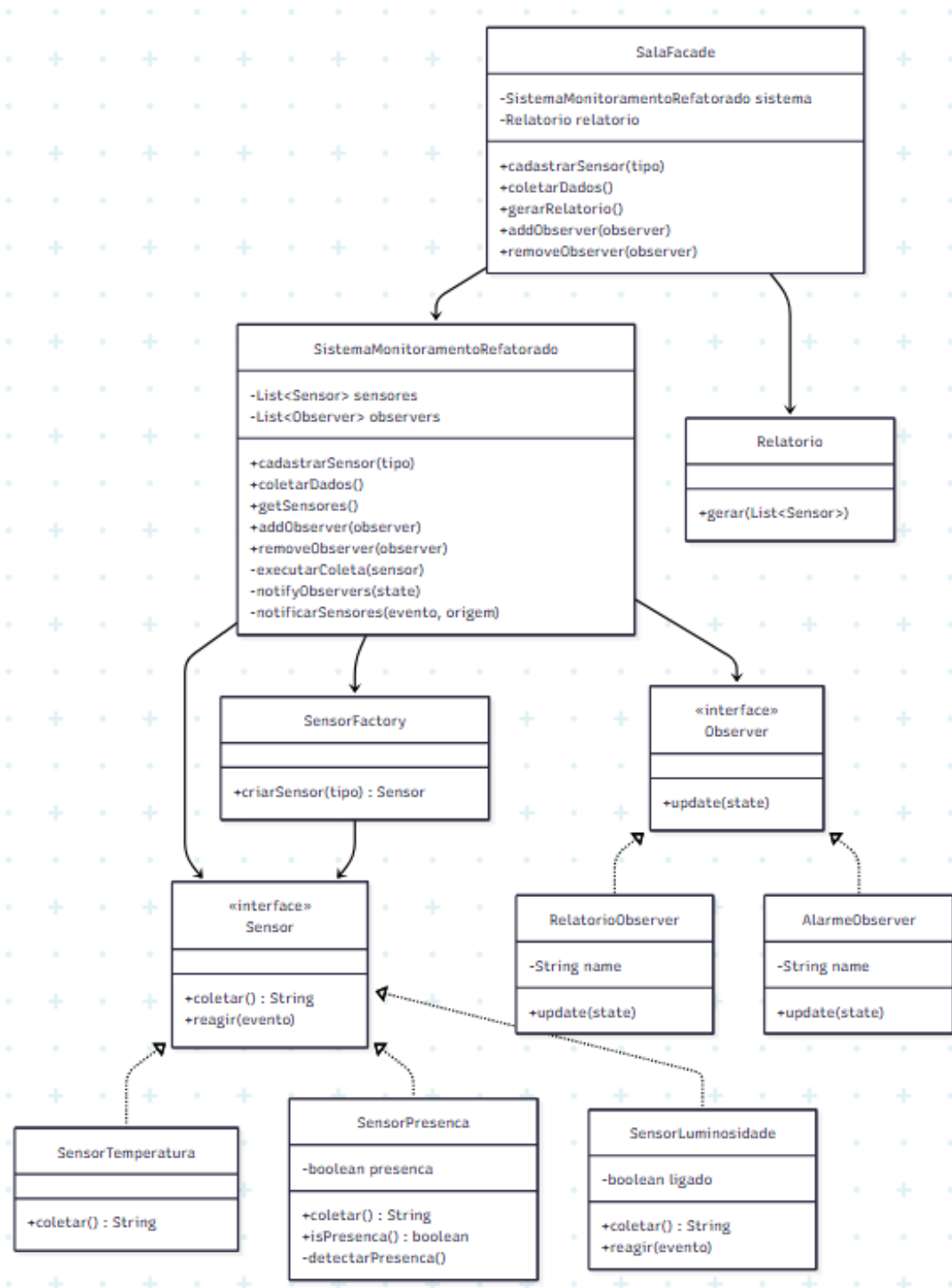
- **Anti-padrão identificado (Spaghetti Code)**

- **Problema:** métodos longos, muitos if/else, lógica duplicada e espalhada em várias classes, dificultando compreensão e manutenção.
- **Consequência:** alta chance de erros ao adicionar novos sensores ou relatórios; difícil testar e estender o sistema.
- **Solução:**
  - Aplicação do **Facade** (SalaFacade) para centralizar operações comuns (cadastrarSensor, coletarDados, gerarRelatorio).
  - Encapsulamento de campos e realocação de métodos para classes específicas, reduzindo acoplamento e melhorando coesão.

- **Anti-padrão identificado (GOD Class - Classe Deus)**

- **Problema:** sensores eram identificados por strings e tratados em if/else.
- **Solução:** criação de uma interface Sensor e classes concretas (SensorTemperatura, SensorPresenca, SensorLuminosidade). Uma fábrica (SensorFactory) é responsável por instanciar os sensores, eliminando o acoplamento direto.

## 5. UML depois



## 6. TABELA COM REFATORAÇÕES APLICADAS E IMPACTO

REFATORAÇÃO	ONDE FOI APLICADA	IMPACTO NA ARQUITETURA
<b>Factory Method</b>	SensorFactory	Removeu acoplamento direto na criação de sensores, permitindo adicionar novos sensores sem alterar o SistemaMonitoramentoRefatorado.
<b>Observer</b>	Observer, AlarmeObserver, RelatorioObserver, usado em SistemaMonitoramentoRefatorado	Permitiu notificação desacoplada de eventos (ex: detecção de presença) para múltiplos observadores. Elimina polling e promove baixo acoplamento.
Strategy	Método reagir em Sensor + implementação em SensorLuminosidade	Permitiu variar comportamento dos sensores diante de eventos sem condicional fixa, substituindo código por polimorfismo.
Facade	SalaFacade	Simplificou a interação com o sistema, expondo apenas métodos essenciais (cadastrar, coletar, relatar), escondendo complexidade.
Extração de método	executarColeta em SistemaMonitoramentoRefatorado	Reduziu duplicação de código, aumentou legibilidade e isolamento de responsabilidades.
Mover método	Notificação de sensores (reagir) movida para dentro dos próprios sensores	Melhorou a coesão: cada sensor é responsável por sua própria reação.
Encapsulamento	isPresenca em SensorPresenca	Protegeu acesso ao estado interno, promovendo boas práticas de encapsulamento.

## 7. CONCLUSÃO

O Desenvolvimento dessa atividade, possibilitou através de evidências na prática como um sistema mal projetado compromete em vários sentidos, como em atualizações futuras, reutilização de códigos e manutenção. A parti da refatoração realizada, mesmo que parcialmente, apresentou que padrões com o Factory e o Facade tornam o software mais organizado, flexível e preparado para futuras evoluções, sem que os princípios básicos do SOLID sejam quebrados.

Assim a atividade cumpriu com o objetivo à evidência de forma pratica a importância da arquitetura de software e aplicação de boas práticas, principalmente quando se parte da fase inicial, não só focando no funcionamento prévio, mais sim a uma visão futura do funcionamento e continuidade do software.

## 8. LINK GIT HUB

9.

➤ <https://github.com/marcelotecas/Evolucao-de-sistema-mal-projetado-com-Padrees.git>

## 10. REFERÊNCIAS

1. CAMPUS CODE. S.O.L.I.D.: Princípio da Responsabilidade Única. Disponível em: <https://www.campuscode.com.br/conteudos/s-o-l-i-d-principio-da-responsabilidade-unica>. Acesso em: 23 ago. 2025.
2. CAMPUS CODE. S.O.L.I.D.: Princípio Aberto/Fechado. Disponível em: <https://www.campuscode.com.br/conteudos/s-o-l-i-d-principio-aberto-fechado>. Acesso em: 23 ago. 2025.
3. CAMPUS CODE. S.O.L.I.D.: Princípio de Inversão de Dependência. Disponível em: <https://www.campuscode.com.br/conteudos/s-o-l-i-d-principio-de-inversao-de-dependencia>. Acesso em: 23 ago. 2025.
4. DEVMEDIA. Design Patterns: Descrições dos Padrões “GoF”. Disponível em: <https://www.devmedia.com.br/design-patterns-padrees-gof/16781>. Acesso em: 23 ago. 2025.
5. MERMAID CHART. Criar diagramas UML com Mermaid. Disponível em: <https://www.mermaidchart.com/app/projects/2c39c97e-72ed-4d0e-adde-49318fd6ce64/diagrams/952ea021-3f33-47e4-84e1-478184922f59/version/v0.1/edit>. Acesso em: 23 ago. 2025.
6. SOFTPLAN. Single Responsibility Principle: conheça esse princípio SOLID. Disponível em: <https://www.softplan.com.br/tech-writers/conheca-um-dos-principios-solid-mais-importantes-single-responsibility-principle/>. Acesso em: 23 ago. 2025.