Relatório Técnico – Projeto Gestão de Tarefas Simples

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

Esse projeto foi desenvolvido com o objetivo de ilustrar, de forma prática e didática, os impactos de uma arquitetura mal estruturada em sistemas orientados a objetos, bem como os benefícios da aplicação de boas práticas de desenvolvimento. A proposta envolveu a criação de duas versões do mesmo sistema: uma versão inicial propositalmente mal projetada e uma versão refatorada que segue os princípios SOLID e utiliza o padrão Service Locator.

Este relatório apresenta uma análise técnica detalhada das duas versões, abordando os anti-patterns identificados, as violações de princípios, as refatorações aplicadas, os diagramas UML comparativos e reflexões sobre boas práticas para evitar problemas semelhantes no futuro.

1 - Diagnóstico dos anti-patterns

Na versão inicial do projeto, diversos anti-patterns foram identificados. Um dos principais anti-patterns presentes foi o “God Object”, representado pela classe GerenciadorTarefas. Essa classe concentrava múltiplas responsabilidades: criação de usuários, gerenciamento de tarefas e controle de fluxo. Isso viola o princípio da responsabilidade única e torna o código difícil de testar e modificar.

Outro anti-pattern evidente foi o “Spaghetti Code”, caracterizado pela ausência de modularidade e pela mistura de lógica de negócio com lógica de apresentação. O método listarTarefas, por exemplo, imprimia diretamente no console, sem qualquer separação entre os dados e a forma como são exibidos. Além disso, o uso de atributos públicos nas classes Usuario e Tarefa expunha os dados diretamente, violando o encapsulamento e facilitando alterações indevidas.

Esses problemas tornam o sistema frágil e propenso a erros. Qualquer modificação em uma funcionalidade pode impactar outras partes do código, exigindo retrabalho e dificultando a evolução do sistema.

2 - Violações SOLID identificadas.

Na versão inicial do projeto, vários desses princípios foram violados. O primeiro deles foi o Single Responsibility Principle (SRP). A classe GerenciadorTarefas claramente não respeita esse princípio, pois acumula responsabilidades que deveriam estar distribuídas entre diferentes componentes. O SRP preconiza que uma classe deve ter apenas um motivo para mudar, o que não ocorre nesse caso.

O segundo princípio violado foi o Open/Closed Principle (OCP). O sistema não está preparado para ser estendido sem que suas classes sejam modificadas. Por exemplo, se quisermos adicionar um novo tipo de tarefa ou uma nova forma de autenticação, precisaríamos alterar diretamente as classes existentes, o que quebra o OCP.

O Liskov Substitution Principle (LSP) não foi diretamente violado, pois não há uso de herança na versão inicial. No entanto, a ausência de abstrações impede que esse princípio seja sequer considerado.

O Interface Segregation Principle (ISP) também não foi aplicado, já que não há interfaces no projeto. Todas as funcionalidades estão acopladas em classes concretas, o que dificulta a reutilização e a testabilidade.

Por fim, o Dependency Inversion Principle (DIP) foi ignorado. As classes dependem diretamente de implementações concretas, como listas e objetos instanciados manualmente, sem qualquer forma de abstração ou injeção de dependência.

3 - Refatorações Aplicadas

A versão refatorada do projeto foi construída com base em uma série de refatorações que visam corrigir os problemas identificados. A primeira mudança foi a separação de responsabilidades. As funcionalidades foram distribuídas entre as camadas model, repository, service e injector, cada uma com um papel específico.

A classe Usuario passou a encapsular seus atributos, oferecendo métodos públicos para acesso e modificação. A classe Tarefa também foi ajustada para seguir esse padrão. O repositório TarefaRepository foi criado para centralizar o armazenamento e recuperação de dados, enquanto o serviço TarefaService passou a ser responsável pela lógica de negócio.

O padrão Service Locator foi introduzido para gerenciar as dependências entre os componentes, permitindo que o sistema seja mais flexível e desacoplado. Essa abordagem facilita a substituição de implementações e a realização de testes unitários.

As refatorações incluíram técnicas como extração de métodos, criação de classes específicas, encapsulamento de dados e remoção de responsabilidades duplicadas. Como resultado, o sistema tornou-se mais modular, legível e preparado para evoluções futuras.

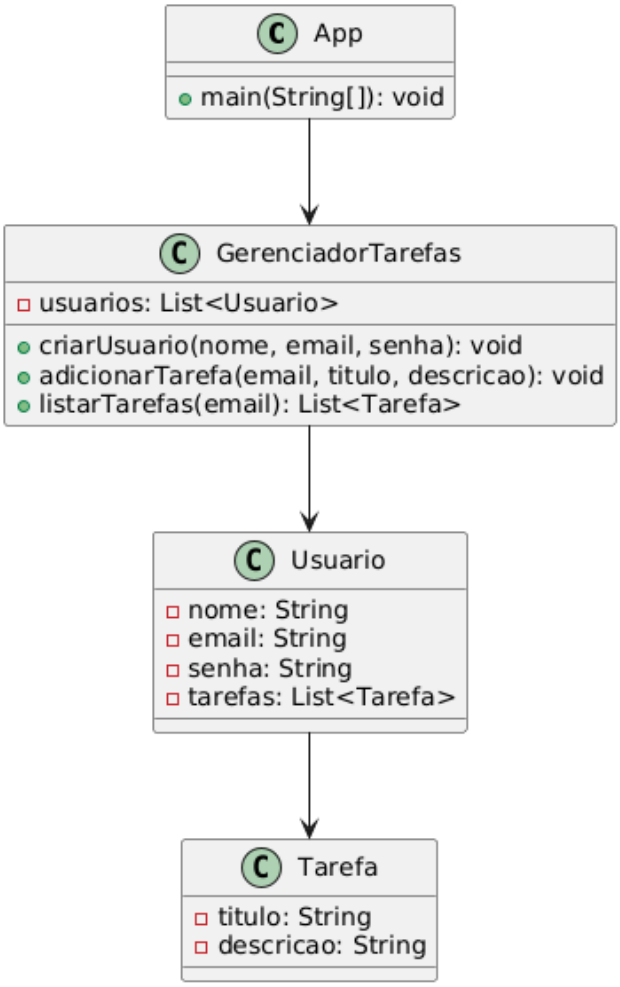
4 - UML antes/depois + código UML.

Os diagramas UML das duas versões evidenciam claramente a evolução da arquitetura. Na versão inicial, o diagrama mostra uma estrutura centralizada em GerenciadorTarefas, com dependências diretas entre todas as classes. Já na versão refatorada, o diagrama apresenta uma arquitetura em camadas, com separação entre dados, lógica e controle de dependências.

4.1 - Código UML versão inicial

```
@startuml
class App {
  +main(String[]): void
}

class GerenciadorTarefas {
  -usuarios: List<Usuario>
  +criarUsuario(nome, email, senha): void
  +adicionarTarefa(email, titulo, descricao): void
  +listarTarefas(email): List<Tarefa>
}

class Usuario {
  -nome: String
  -email: String
  -senha: String
  -tarefas: List<Tarefa>
}

class Tarefa {
  -titulo: String
  -descricao: String
}

App --> GerenciadorTarefas
GerenciadorTarefas --> Usuario
Usuario --> Tarefa
@enduml
```

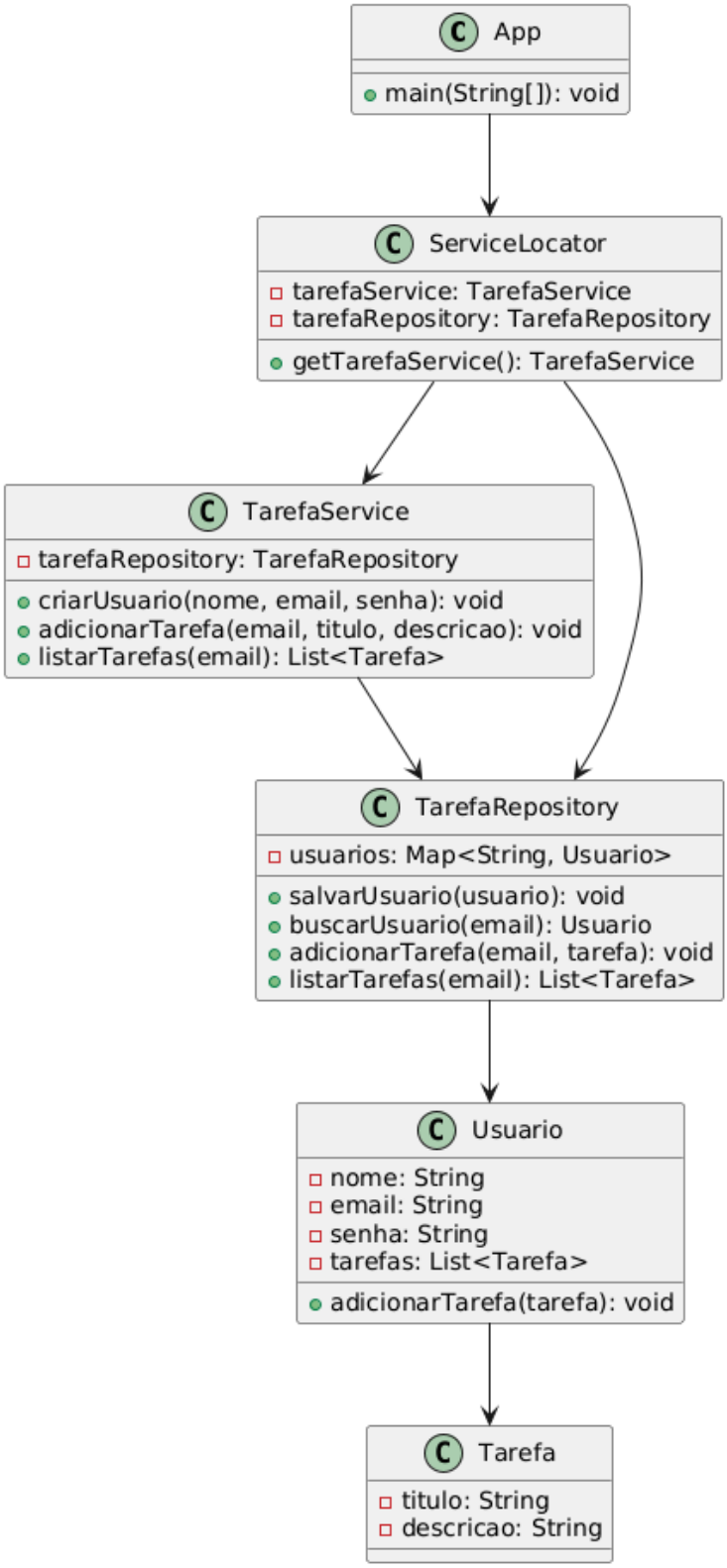
UML versão inicial



4.2 - Código UML versão refatorada

```
@startuml
class App {
  +main(String[]): void
}

class ServiceLocator {
  -tarefaService: TarefaService
  -tarefaRepository: TarefaRepository
  +getTarefaService(): TarefaService
}

class TarefaService {
  -tarefaRepository: TarefaRepository
  +criarUsuario(nome, email, senha): void
  +adicionarTarefa(email, titulo, descricao): void
  +listarTarefas(email): List<Tarefa>
}

class TarefaRepository {
  -usuarios: Map<String, Usuario>
  +salvarUsuario(usuario): void
  +buscarUsuario(email): Usuario
  +adicionarTarefa(email, tarefa): void
  +listarTarefas(email): List<Tarefa>
}

class Usuario {
  -nome: String
  -email: String
  -senha: String
  -tarefas: List<Tarefa>
  +adicionarTarefa(tarefa): void
}

class Tarefa {
  -titulo: String
  -descricao: String
}

App --> ServiceLocator
ServiceLocator --> TarefaService
ServiceLocator --> TarefaRepository
TarefaService --> TarefaRepository
TarefaRepository --> Usuario
Usuario --> Tarefa
@enduml
```

UML refatorado



5 - Comparativo entre IoC e Service Locator (vantagens, riscos, quando aplicar).

Durante a refatoração, optamos por utilizar o padrão Service Locator como forma de gerenciar dependências. Esse padrão oferece uma centralização das instâncias e facilita o acesso aos serviços, especialmente em projetos pequenos ou acadêmicos. No entanto, é importante entender que o Service Locator é uma forma de aplicar o conceito de Inversão de Controle (IoC).

O IoC é um princípio que sugere que as dependências de uma classe devem ser fornecidas externamente, e não criadas internamente. A forma mais comum de aplicar IoC é por meio da Injeção de Dependência (DI), que pode ser feita manualmente ou com frameworks como Spring.

O Service Locator tem como vantagem a simplicidade e a centralização, mas pode esconder dependências e dificultar testes se não for bem utilizado. Já a Injeção de Dependência é mais explícita e favorece a testabilidade, mas exige mais estrutura e configuração.

Em projetos maiores, a DI tende a ser mais indicada. No nosso caso, o Service Locator foi suficiente para demonstrar os benefícios da inversão de controle sem adicionar complexidade desnecessária.

6 - Reflexão sobre como evitar retorno de anti-patterns em futuras versões.

Para evitar que os anti-patterns reapareçam em versões futuras do projeto, é fundamental adotar boas práticas desde o início do desenvolvimento. A primeira delas é aplicar os princípios SOLID como guia para a estruturação das classes. Além disso, é importante manter uma arquitetura em camadas, com separação clara entre dados, lógica e apresentação.

O uso de testes automatizados ajuda a garantir que o sistema continue funcionando após modificações. A revisão de código entre os colegas também é uma prática valiosa para identificar problemas antes que se tornem críticos.

Documentar o projeto, manter os diagramas atualizados e utilizar ferramentas de análise estática são outras formas de manter a qualidade do código. Por fim, cultivar uma cultura de aprendizado contínuo e refatoração constante é essencial para o crescimento técnico e a manutenção de sistemas saudáveis.