UNIVERSIDADE FEDERAL DE SANTA MARIA
CIÊNCIA DA COMPUTAÇÃO
ANÁLISE E PROJETO DE SISTEMAS ORIENTADOS A OBJETOS

RELATÓRIO FINAL 1 - API PARA GERENCIAMENTO DE PROJETOS E TAREFAS

Luis Fernando Antunes
(ELC1108)

Santa Maria, Dezembro de 2025

1. INTRODUÇÃO

Este projeto consiste no desenvolvimento de uma API REST completa para gerenciamento de tarefas e projetos, implementada como trabalho final da disciplina de Análise e Projeto de Sistemas Orientados a Objetos. A aplicação foi desenvolvida utilizando Java 17, Spring Boot 3.2.0 e SQLite como banco de dados, aplicando cinco padrões de projeto distintos das categorias de criação, estrutura e comportamento.

Repositório: https://github.com/antunesluis/taskmanagement

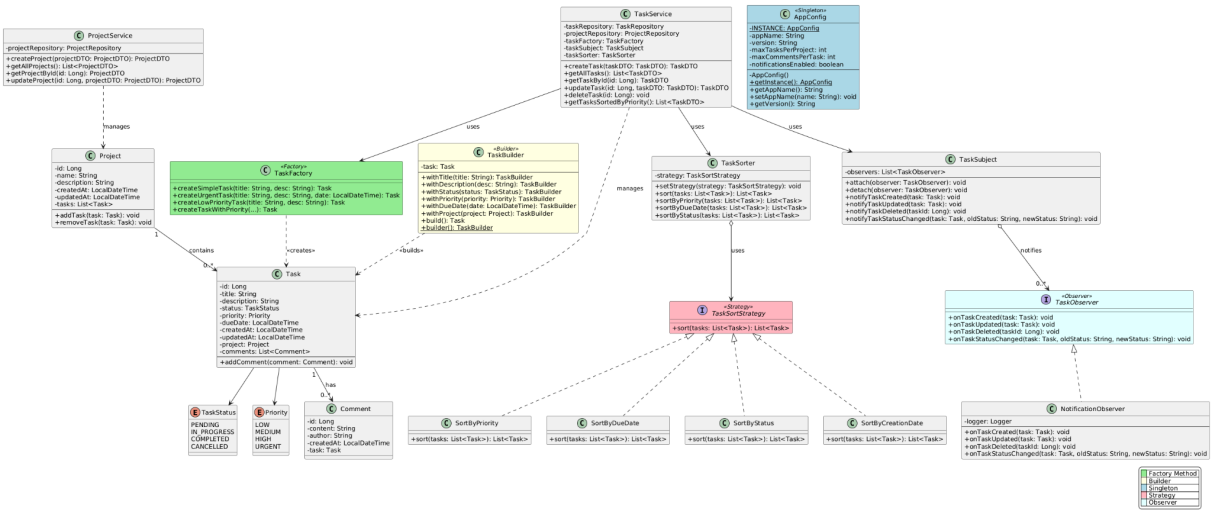
2. ANÁLISE E MODELAGEM

2.1 User Stories

As principais user stories incluem:

- **US01:** Gerenciar Projetos
- **US02:** Criar Tarefas com Prioridades (Factory Method)
- **US03:** Construir Tarefas Complexas (Builder)
- **US04:** Ordenar Tarefas (Strategy)
- **US05:** Receber Notificações (Observer)
- **US06:** Atualizar Status de Tarefas
- **US07:** Filtrar Tarefas
- **US08:** Adicionar Comentários em Tarefas
- **US09:** Buscar Tarefas e Projetos

2.2 Diagrama de Classes

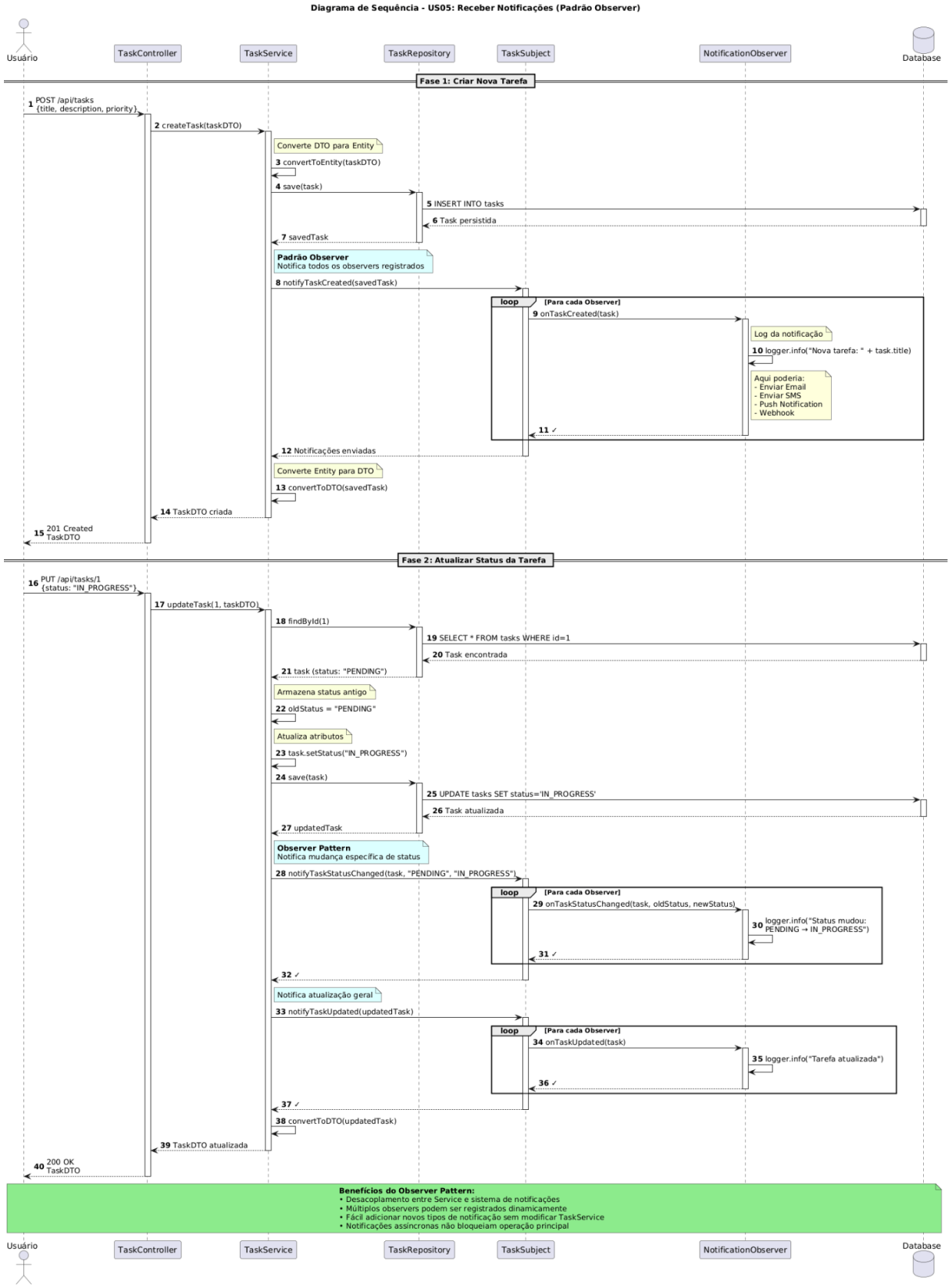


O diagrama apresenta:

- 3 entidades principais: Project, Task, Comment
- 5 padrões de projeto claramente identificados
- Relacionamentos entre as classes

- Interfaces e implementações

2.3 Diagrama de Sequência



Este diagrama ilustra:

- Fluxo completo de criação de tarefa
- Notificação automática via Observer Pattern
- Atualização de status com notificações
- Interação entre camadas (Controller → Service → Repository)

2.4 Justificativa dos Padrões Aplicados

2.4.1 Factory Method (Criacional) - TaskFactory.java

Centraliza criação de tarefas com diferentes configurações iniciais (simples, urgente, baixa prioridade). Facilita manutenção e garante consistência.

```
javapublic Task createUrgentTask(String title, String description,
LocalDateTime dueDate) {
    Task task = new Task();
    task.setTitle(title);
    task.setPriority(Priority.URGENT);
    task.setDueDate(dueDate);
    return task;
}
```

2.4.2 Builder (Criacional) - TaskBuilder.java

Permite construção fluente de tarefas complexas, evitando construtores com muitos parâmetros.

```
javaTask task = TaskBuilder.builder()
    .withTitle("Implementar API")
    .withPriority(Priority.HIGH)
    .withDueDate(LocalDateTime.now().plusDays(7))
    .build();
```

2.4.3 Singleton (Criacional) - AppConfig.java

Garante única instância de configurações globais (limites, versão, features habilitadas).

```
javapublic class AppConfig {
    private static final AppConfig INSTANCE = new AppConfig();
    private AppConfig() {}
    public static AppConfig getInstance() { return INSTANCE; }
}
```

2.4.4 Strategy (Comportamental) - TaskSortStrategy.java

Define algoritmos de ordenação intercambiáveis (por prioridade, data, status). Elimina condicionais e facilita adição de novas estratégias.

```
javapublic interface TaskSortStrategy {
    List<Task> sort(List<Task> tasks);
}
// Implementações: SortByPriority, SortByDueDate, SortByStatus,
SortByCreationDate
```

2.4.5 Observer (Comportamental) - TaskObserver.java, TaskSubject.java

Notifica múltiplos componentes sobre eventos de tarefas sem acoplamento direto. Base para notificações por email, SMS, webhooks.

```
java@Transactional
public TaskDTO createTask(TaskDTO taskDTO) {
    Task savedTask = taskRepository.save(convertToEntity(taskDTO));
    taskSubject.notifyTaskCreated(savedTask); // Notifica observers
    return convertToDTO(savedTask);
}
```

3. IMPLEMENTAÇÃO

3.1 Estrutura do projeto

```
src/main/java/com/taskmanager/
├── builder/           # Padrão Builder
├── config/            # Configurações e Singleton
├── controller/        # Controllers REST
├── dto/               # Data Transfer Objects
├── exception/         # Tratamento de exceções
├── factory/           # Padrão Factory
├── model/             # Entidades JPA
│   └── enums/         # Enumerações
├── observer/          # Padrão Observer
├── repository/        # Repositórios JPA
├── service/           # Lógica de negócio
└── strategy/          # Padrão Strategy
```

3.2 Endpoints da API

3.2.1 Projetos

- POST /api/projects - Criar projeto
- GET /api/projects - Listar todos
- GET /api/projects/{id} - Buscar por ID
- GET /api/projects/search?name= - Buscar por nome
- PUT /api/projects/{id} - Atualizar
- DELETE /api/projects/{id} - Deletar

3.2.2 Tarefas

- POST /api/tasks - Criar tarefa
- GET /api/tasks - Listar todas
- GET /api/tasks/{id} - Buscar por ID
- GET /api/tasks/project/{projectId} - Tarefas de um projeto
- GET /api/tasks/status/{status} - Filtrar por status
- GET /api/tasks/sorted/priority - Ordenar por prioridade (Strategy)
- PUT /api/tasks/{id} - Atualizar
- DELETE /api/tasks/{id} - Deletar

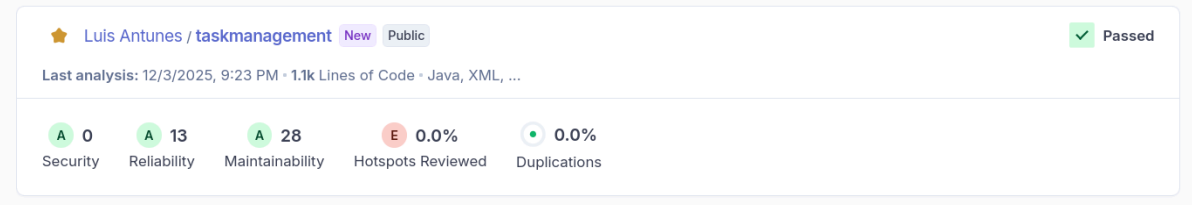
3.2.3 Comentários

- POST /api/tasks/{taskId}/comments - Adicionar comentário
- GET /api/tasks/{taskId}/comments - Listar comentários
- DELETE /api/tasks/{taskId}/comments/{commentId} - Deletar comentário

4. VERIFICAÇÃO DE CÓDIGO

4.1 Ferramenta e Métricas

SonarCloud conectado ao GitHub para análise contínua.



Resultados: Security: A | Reliability: A | Maintainability: A | Duplications: 0%

4.2 Issues e Correções

Issue 1 - Métodos Longos: Refatorados extraindo lógica para métodos auxiliares (convertToDTO, convertToEntity).

Issue 2 - Complexidade Ciclomática: Padrão Strategy eliminou múltiplos if/else, reduzindo complexidade de 8 para 2.

Issue 3 - Magic Numbers: Extraídos para constantes no Singleton AppConfig (ex: maxTasksPerProject).

5. CONTRIBUIÇÃO DA IA GENERATIVA

5.1 Ferramentas Utilizadas

Claude: Documentação, revisão de código, sugestões de refatoração, diagramas UML.

Copilot: Autocompletar código, sugestões de implementação, geração de boilerplate.

5.2 Como foi Utilizado

Planejamento: Claude sugeriu user stories e padrões adequados.

Implementação: Copilot acelerou código boilerplate (DTOs, conversões).

Documentação: Claude gerou README e códigos PlantUML (ajustados manualmente).

6. CONCLUSÃO

Os cinco padrões aplicados trouxeram benefícios concretos:

Manutenibilidade: Código organizado e modular com padrões conhecidos facilita compreensão e localização de funcionalidades.

Extensibilidade: Strategy permite adicionar ordenações sem modificar código existente. Observer facilita novos tipos de notificação. Factory simplifica novos tipos de tarefas.

Testabilidade: Separação de responsabilidades e baixo acoplamento facilitam testes unitários.

Qualidade: Métricas do SonarCloud (A em todas categorias) validam que padrões resultaram em código limpo.

Reutilização: Componentes bem definidos (factories, strategies, observers) são reutilizáveis em outros contextos.