Aplicação dos Princípios de Clean Code e Clean Architecture no Desenvolvimento de um Sistema de Dispensa de Medicamentos na Metrópole Salvador – BA

Fabiano Santos Lobo [*1], Filipe Miranda de Oliveira [†1], Ian dos Santos de Almeida [‡1], Lucas de Souza Conceição [§1], Paulo Ricardo de Jesus Santana [¶ 1], Angela Peixoto Santana [‖1*]

[1] Análise e Desenvolvimento de Sistemas
Escola de Tecnologias
Universidade Católica do Salvador (UCSAL)
Av. Prof. Pinto de Aguiar, 2589 Pituaçu, CEP: 41740-090
Salvador/BA, Brasil
[1] *{fabiano.lobo,filipemirando.oliveira,ian.almeida,lucassouza.conceicao ,pauloricardo.santana}@ucsal.edu.br*
[1*] *{angela.santana}@pro.ucsal.edu.br*

Abril 2024

[*]fabiano.lobo@ucsal.edu.br
[†]filipemiranda.oliveira@ucsal.edu.br
[‡]ian.almeida@ucsal.edu.br
[§]lucassouza.conceicao@ucsal.edu.br
[¶]pauloricardo.santana@ucsal.edu.br
[‖]angela.santana@pro.ucsal.br

Resumo

Este trabalho evidencia a relevância da adoção de boas práticas no desenvolvimento de software, como o Clean Code e a Clean Architecture. Essas abordagens asseguram a qualidade e a manutenibilidade do sistema, características essenciais em aplicações voltadas à população. A crescente demanda por soluções tecnológicas que melhorem o acesso aos serviços públicos de saúde reforça a necessidade do desenvolvimento de sistemas eficientes, transparentes e acessíveis.

Nesse contexto, apresenta-se o DispMed, um sistema voltado à otimização do processo de dispensação de medicamentos em unidades públicas de saúde, com o objetivo de ampliar o acesso da população e garantir maior controle e organização.

Além do caráter técnico, o DispMed representa uma iniciativa de impacto social, ao utilizar a tecnologia como instrumento de apoio à gestão pública e à promoção do bem-estar coletivo. A proposta integra boas práticas de engenharia de software à responsabilidade social, reforçando a importância da tecnologia como ferramenta de impacto positivo na gestão da saúde pública.

Assim, o projeto demostra que é possível alinhar excelência técnica à geração de valor social, por meio de soluções tecnológicas simples, sustentáveis e capazes de causar impacto positivo na vida das pessoas.

Palavras-chaves: Clean Code, Clean Architecture, Saúde Pública, Tecnologia Social, Desenvolvimento Sustentável.

Abstract

This work highlights the importance of adopting good practices in software development, such as Clean Code and Clean Architecture. These approaches ensure the quality and maintainability of the system, which are essential characteristics in applications aimed at the population. The growing demand for technological solutions that improve access to public health services reinforces the need to develop efficient, transparent and accessible systems. In this context, we present DispMed, a system designed to optimize the process of dispensing medicines in public health units, with the aim of increasing access for the population and ensuring greater control and organization. In addition to its technical nature, DispMed is an initiative with a social impact, as it uses technology as an instrument to support public management and promote collective well-being. The proposal integrates good software engineering practices with social responsibility, reinforcing the importance of technology as a tool with a positive impact on public health management. In this way, the project demonstrates that it is possible to align technical excellence with the generation of social value, through simple, sustainable technological solutions capable of having a positive impact on people's lives.

Keywords: Clean Code, Clean Architecture, Public Health, Social Technology, Sustainable Development

1 Introdução

O acesso eficiente e transparente a medicamentos é uma das principais demandas da população que depende do Sistema Único de Saúde (SUS) no Brasil. A indisponibilidade de remédios nas unidades públicas, aliada à falta de informação sobre sua localização, compromete a efetividade do atendimento à saúde, sobretudo em grandes centros urbanos, como Salvador/BA. Nesse contexto, a tecnologia surge como uma aliada essencial para otimizar a gestão da saúde pública, promovendo soluções acessíveis e organizadas.

Este trabalho tem como objetivo apresentar o desenvolvimento de um sistema *web* de apoio à população na identificação e localização de medicamentos disponibilizados pela rede municipal de saúde. O projeto adota os princípios do *Clean Code* e da *Clean Architecture* como diretrizes fundamentais para a construção de um *software* de qualidade.

Ao aplicar boas práticas da engenharia de *software*, como os princípios *SOLID, TDD (Test-Driven Development), KISS e YAGNI*, o sistema proposto visa não apenas resolver um problema social concreto, mas também servir como exemplo de desenvolvimento responsável e alinhado com os padrões mais atuais da área de tecnologia. A estrutura adotada, baseada no modelo C4 (Contexto, Contêineres, Componentes e Código), permite uma compreensão clara e escalável da arquitetura do sistema, facilitando sua evolução e manutenção.

O sistema *DispMed* foi concebido com foco na população soteropolitana, mas possui potencial de escalabilidade para outras regiões, reforçando o papel da tecnologia no fortalecimento das políticas públicas de saúde.

Ao longo deste trabalho, serão apresentados os conceitos teóricos fundamentais que embasam o projeto, como Engenharia de Software, qualidade de código, princípios do Clean Code, SOLID e Clean Architecture. Em seguida, descreve-se o desenvolvimento do sistema DispMed, detalhando os requisitos, a modelagem da arquitetura utilizando o modelo C4, a estrutura técnica adotada e as decisões de implementação tanto no backend quanto no frontend. Por fim, são discutidos os resultados obtidos, as contribuições sociais e técnicas do sistema, bem como sugestões de melhorias e direções para trabalhos futuros.

2 Fundamentação Teórica

Este capítulo tem como objetivo apresentar a fundamentação teórica sobre os principais conceitos utilizados no desenvolvimento deste trabalho de conclusão de curso. Serão abordadas as bases necessárias para o desenvolvimento de um sistema de dispensa de medicamentos orientado a essas boas práticas de codificação.

2.1 Engenharia de *Software*

A Engenharia de *Software* é a área responsável pelo estudo e aplicação de princípios e práticas voltadas ao desenvolvimento, operação e manutenção de sistemas de *software*. De acordo com Sommerville (2010), trata-se da "aplicação de uma abordagem sistemática, disciplinada e quantificável ao desenvolvimento, operação e manutenção de *software*", o que ressalta a importância de métodos bem estruturados para garantir a eficiência e a eficácia no processo de criação de *software*.

Essa disciplina abrange diversas etapas do ciclo de vida do *software*, incluindo a análise de requisitos, o *design* do sistema, a implementação, os testes, o controle de versão e a documentação. Como destaca Pressman (2014), "um bom *software* deve ser desenvolvido de acordo com um conjunto de princípios e práticas que garantam sua qualidade e funcionalidade". Isso demonstra que Engenharia de *Software* vai muito além da programação trata-se de um conjunto coordenado de atividades que visam à entrega de soluções tecnológicas de alta qualidade.

A construção de um *software* deve ser guiada pelas necessidades e expectativas do usuário final. Nesse sentido, Pressman (2014) afirma que "o sucesso de um projeto de *software* depende da capacidade de atender às necessidades dos usuários". Assim, o levantamento e a análise de requisitos tornam-se etapas cruciais para assegurar que o sistema realmente resolva os problemas propostos.

Dessa forma, a Engenharia de *Software* utiliza metodologias e práticas rigorosas que não apenas atendem às demandas dos usuários, mas também oferecem suporte à manutenção e evolução do sistema a longo prazo. Essa abordagem contribui para um desenvolvimento mais eficaz, melhora a qualidade do produto final e aumenta as chances de sucesso do projeto como um todo.

2.2 Qualidade de *Software*

A qualidade de *software* representa um conceito que ultrapassa a simples verificação de funcionamento. Está diretamente relacionada à capacidade de um sistema atender às necessidades dos usuários, oferecendo confiabilidade, eficiência e boa usabilidade. De acordo com a norma ISO/IEC (2011), a qualidade de *software* é definida por atributos como funcionalidade, eficiência, compatibilidade, usabilidade, confiabilidade, segurança, manutenibilidade e portabilidade. Esses elementos são essenciais para que o *software* cumpra sua função de forma eficaz, proporcionando também uma experiência satisfatória ao usuário.

A implementação de práticas voltadas à qualidade, como testes automatizados e revisões de código-fonte, é crucial para a identificação e correção de falhas antes da entrega do produto. Segundo Beizer (1990), "a qualidade do *software* é um reflexo da qualidade do

processo de desenvolvimento", o que evidencia a importância de processos estruturados e bem definidos para garantir a qualidade do produto final.

Conforme McBreen (2001), "a qualidade deve ser uma prioridade em toda a organização, e não apenas uma responsabilidade da equipe de desenvolvimento". Dessa forma, é necessário que todos os envolvidos desde a alta gestão, até os profissionais técnicos estejam alinhados e comprometidos com os princípios e práticas que promovem a qualidade contínua do *software*.

2.3 *Clean Code*

No desenvolvimento de *software*, a qualidade do código envolve mais do que apenas sua funcionalidade. O conceito de *Clean Code* - Código Limpo, representa uma filosofia que preconiza a escrita de programas legíveis, organizados e de fácil manutenção, com foco na comunicação clara entre desenvolvedores e sistemas.

Popularizado por Martin (2009) em seu livro *Clean Code: A Handbook of Agile Software Craftsmanship*, esse conceito defende que um código bem estruturado reduz custos de manutenção, facilita a colaboração entre membros da equipe e minimiza a ocorrência de erros. Segundo o autor, um código limpo deve ser simples e direto, permitindo que qualquer desenvolvedor consiga compreendê-lo com facilidade, independentemente de sua familiaridade prévia com o projeto. Nessa perspectiva, simplicidade não implica ausência de sofisticação, mas sim uma estrutura lógica clara, livre de complexidades desnecessárias.

O *Clean Code* não impõe regras fixas, mas sim princípios que orientam boas práticas de desenvolvimento. Entre os fundamentos mais relevantes, destacam-se:

- Nomes significativos: variáveis, funções e classes devem ser nomeadas de forma clara e intuitiva, promovendo legibilidade e evitando ambiguidades (MARTIN, 2008);
- Funções pequenas e coesas: cada função deve ser responsável por uma única tarefa, de maneira objetiva e sem efeitos colaterais, facilitando a manutenção e a reutilização (FOWLER, 2018);
- Código autoexplicativo: a legibilidade deve prevalecer sobre a necessidade de comentários excessivos, com organização que torne o código intuitivo (MARTIN, 2008);
- Eliminação de código duplicado: a modularização e reutilização evitam redundâncias que dificultam a manutenção (MCCONNELL, 2004);
- Tratamento eficiente de erros: falhas e exceções devem ser tratadas adequadamente, assegurando o bom funcionamento do sistema (SOMMERVILLE, 2010);
- Adoção de convenções e padrões: seguir boas práticas estabelecidas favorece a colaboração e a manutenibilidade do código (PRESSMAN, 2014);

- Testabilidade: o código deve ser projetado para facilitar a realização de testes, promovendo segurança e confiabilidade (ISO/IEC, 2011);
- Princípio do Menor Espanto: o comportamento do sistema deve ser previsível, evitando ambiguidades ou comportamentos inesperados (HUNT; THOMAS, 1999);
- Complexidade controlada: o uso de abstrações adequadas deve reduzir a sobrecarga cognitiva e tornar o código mais compreensível (CUNNINGHAM, 1992).

Essas práticas visam mitigar problemas recorrentes no desenvolvimento de *software*, como a dívida técnica e as dificuldades de manutenção em sistemas legados. Tais desafios comprometem a sustentabilidade e a evolução de aplicações ao longo do tempo. Nesse contexto, Fowler (2018), em sua obra *Refactoring: Improving the Design of Existing Code*, destaca a relevância do *Clean Code* ao afirmar que "qualquer um pode escrever código que um computador entende. Bons programadores escrevem código que humanos entendem", enfatizando a importância da legibilidade e da clareza no desenvolvimento de soluções sustentáveis.

2.4 Princípios SOLID

SOLID é um acrônimo que representa cinco princípios fundamentais de design orientado a objetos, propostos por Robert C. Martin, com o objetivo de tornar o código mais compreensível, flexível e de fácil manutenção (MARTIN, 2002):

- S – *Single Responsibility Principle* - (Princípio da Responsabilidade Única): Cada classe deve possuir apenas uma responsabilidade, ou seja, uma única razão para sofrer modificações. Isso contribui para a coesão do código e facilita a sua manutenção.
- O – *Open/Closed Principle* - (Princípio Aberto/Fechado): Entidades de *software* (como classes, módulos e funções) devem estar abertas para extensão, mas fechadas para modificação. Essa abordagem permite a adição de novos comportamentos sem alterar o código existente, favorecendo a reutilização e reduzindo riscos de regressão.
- L – *Liskov Substitution Principle* - (Princípio da Substituição de Liskov): Subtipos devem ser substituíveis por seus tipos base sem comprometer o funcionamento do sistema. Em outras palavras, uma subclasse deve manter o comportamento esperado da superclasse, garantindo a integridade da aplicação.
- I – *Interface Segregation Principle* - (Princípio da Segregação de Interfaces): Interfaces específicas e especializadas devem ser preferidas a interfaces genéricas e abrangentes. Isso evita que classes sejam forçadas a implementar métodos que não utilizam, promovendo um design mais coeso.

- D – *Dependency Inversion Principle* - (Princípio da Inversão de Dependência): Módulos de alto nível não devem depender de módulos de baixo nível; ambos devem depender de abstrações. Esse princípio reforça o uso de interfaces e injeção de dependência, promovendo um código mais flexível e desacoplado.

Esses princípios contribuem significativamente para a redução do acoplamento entre componentes e promovem a escalabilidade do sistema, possibilitando que o código evolua com menos retrabalho e maior previsibilidade.

2.5 *YAGNI (You Ain't Gonna Need It)*

O princípio *YAGNI* estabelece que funcionalidades devem ser implementadas apenas quando houver uma necessidade comprovada. Essa abordagem evita o excesso de código desnecessário, reduzindo a complexidade e melhorando a manutenibilidade do *software*. Alinha-se diretamente com os fundamentos do *Clean Code*, que valorizam soluções simples, diretas e orientadas à necessidade real do sistema (BECK, 2002).

2.6 *KISS (Keep It Simple, Stupid)*

O princípio *KISS* enfatiza que a simplicidade deve ser priorizada sempre que possível, no desenvolvimento de *software*. Soluções simples tendem a ser mais fáceis de entender, testar e manter, além de reduzirem a incidência de erros. Essa simplicidade está no cerne das práticas de *Clean Code*, que buscam um código mais legível, organizado e sustentável ao longo do tempo (BROOKS, 1995).

2.7 *Test-Driven Development (TDD)*

O desenvolvimento orientado por testes, ou *Test-Driven Development*, propõe que testes automatizados sejam escritos antes da implementação do código. Essa abordagem garante que o sistema seja construído com foco na testabilidade e na prevenção de falhas, contribuindo para a robustez e a confiabilidade do *software*.

O TDD reforça os princípios do *Clean Code*, ao promover um desenvolvimento disciplinado e orientado à qualidade desde as etapas iniciais (BECK, 2002).

2.7.1 Refatoração Contínua

A refatoração contínua consiste na reestruturação sistemática do código-fonte com o objetivo de melhorar sua legibilidade, organização e manutenibilidade, sem alterar seu comportamento externo. Essa prática está fortemente associada aos princípios do *Clean Code*, pois promove a eliminação de duplicações, a simplificação de estruturas complexas e a

remoção de trechos obsoletos. Segundo Fowler (2018), a refatoração é uma estratégia essencial para manter o *Clean Code* e sustentável, garantindo que o *software* possa evoluir com segurança e qualidade ao longo do tempo.

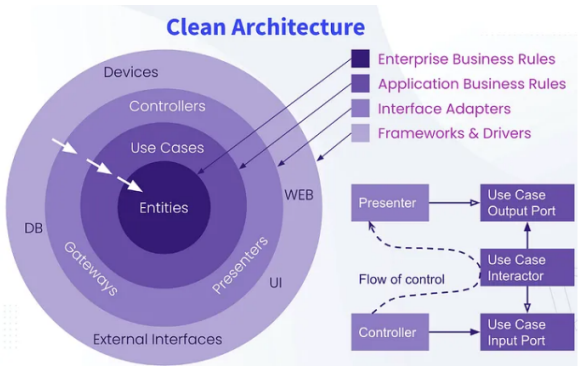
2.8 *Clean Architecture*

A *Clean Architecture*, proposta por Robert C. Martin, apresenta um modelo de organização de *software* baseado na separação de responsabilidades e na independência entre as camadas do sistema. Essa abordagem estabelece que as regras de negócio devem ser isoladas de detalhes externos, como bancos de dados, frameworks, bibliotecas e interfaces gráficas. Ao promover essa separação, o *Clean Architecture* favorece a testabilidade, facilita a manutenção e amplia a flexibilidade da aplicação. Além disso, contribui para a longevidade do sistema, permitindo a substituição de tecnologias com impacto mínimo sobre a lógica central da aplicação. (MARTIN, 2017)

> "A intenção da *Arquitetura Limpa* é separar o código que é central para o funcionamento do sistema (regras de negócio) do código que é periférico (infraestrutura, banco de dados, UI)" — Martin, R.C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.*

Camadas do *Clean Architecture*: A estrutura é dividida em camadas concêntricas, com dependências direcionadas para o centro:

Figura 1 – Clean Architecture



(TEKIN, 2023)

- *Entities* (Entidades): regras de negócio mais gerais e independentes.
- *Use Cases* (Casos de Uso): regras específicas da aplicação.
- *Interface Adapters*: adaptadores que convertem dados entre os casos de uso e o mundo externo.
- *Interface* de Infraestrutura: *Frameworks* e *Drivers*, camada externa com detalhes de implementação.

A divisão proposta reforça o princípio da inversão de dependência, garantindo que os detalhes de implementação dependam das abstrações, e não o inverso. Com isso, os princípios do *Clean Code* são fortalecidos, promovendo uma estrutura mais organizada, clara e de fácil manutenção, ao assegurar que as regras de negócio fiquem desacopladas dos detalhes de implementação.

Nesse contexto, Martin (2017) enfatiza que "a arquitetura deve ser independente de *frameworks*, *interfaces* de usuário, banco de dados e quaisquer outros detalhes externos que possam sofrer alterações".

3 Trabalhos relacionados

Neste capítulo, serão apresentados os principais trabalhos relacionados à temática do *Clean Code*. A revisão da literatura é fundamental para contextualizar a pesquisa, proporcionando uma visão abrangente sobre o estado atual do tema, além de permitir a identificação das contribuições e limitações dos estudos anteriores.

O artigo de Caio e Oswaldo (2022) explora a importância da refatoração no contexto do desenvolvimento de *software*, destacando que a qualidade do código é essencial para a manutenção e evolução das aplicações. A pesquisa, baseada em revisão bibliográfica, discute como a refatoração transforma códigos "sujos"em códigos limpos, aprimorando sua legibilidade e eficiência sem a necessidade de adicionar novas funcionalidades. Os autores defendem que a refatoração deve ser um processo contínuo ao longo do desenvolvimento, contribuindo para a produtividade da equipe e facilitando a colaboração entre os desenvolvedores. O estudo conclui que a adoção de boas práticas de consolidação e o cuidado com os detalhes são fundamentais para garantir um código de alta qualidade, beneficiando tanto os desenvolvedores, quanto as equipes envolvidas.

O artigo de Phelipe e Fernando (2024) discute a importância de manter padrões do *Clean Code* e boas práticas no desenvolvimento de *software*, reforçando que a qualidade do código é crucial para a manutenção e evolução das aplicações. Com base em uma revisão bibliográfica, a pesquisa revela que códigos mal estruturados podem resultar em dificuldades de interpretação e manutenção, o que gera aumento nos custos e atrasos nas entregas. O texto apresenta princípios fundamentais do *Clean Code*, como a definição apropriada de nomes, a importância de comentários, a refatoração contínua e os princípios *SOLID*, que visam aprimorar a legibilidade e organização do código. Os autores concluem que a adoção dessas práticas facilita o trabalho dos desenvolvedores, além de contribuir para a eficiência e agilidade das equipes de desenvolvimento, resultando em benefícios para as empresas.

O artigo de Azevedo (2023) descreve o desenvolvimento da aplicação *web* Oratio, projetada para melhorar a gestão dos Trabalhos de Conclusão de Curso na Universidade Federal de Campina Grande. A aplicação adota os princípios do *Clean Code* e da arquite-

tura *SOLID*, assegurando a qualidade, eficiência e escalabilidade do código. Por meio da centralização de informações, a Oratio aprimorou a comunicação entre professores e alunos, facilitando o processo de avaliação. A metodologia ágil adotada possibilitou ajustes contínuos durante o desenvolvimento, resultando em uma ferramenta eficiente e de fácil manutenção.

O artigo de Jordan et al. (2024) discute o desenvolvimento da plataforma "Voluntária Tech", que conecta profissionais da área de tecnologia a organizações sociais, promovendo o engajamento em projetos de impacto social. Utilizando a *Clean Architecture*, o sistema foi projetado para ser escalável e de fácil manutenção, promovendo a colaboração entre voluntários e ONGs. O estudo destaca a importância de boas práticas de engenharia de *software* e propõe uma solução inovadora para fortalecer o voluntariado tecnológico, contribuindo para a transformação social. Além disso, o artigo sugere melhorias futuras para a plataforma, incluindo aprimoramentos no *feedback* e a integração de novas tecnologias.

A partir dos estudos analisados, observa-se uma forte ênfase na importância do *Clean Code* e na aplicação de seus princípios e metodologias associadas, incluindo o uso da *Clean Architecture*, para garantir a qualidade e eficiência dos sistemas desenvolvidos.

Este trabalho de pesquisa se destaca ao focar no desenvolvimento de um sistema de dispensa de medicamentos, com um objetivo social claro: aprimorar a pesquisa da sociedade sobre os medicamentos dispensados nas unidades de saúde da metrópole de Salvador/BA. O projeto não se limita à criação de um sistema, mas também à implementação de uma solução que atenda diretamente às necessidades das comunidades, promovendo transparência e eficiência na gestão de medicamentos.

4 Desenvolvimento do DispMed

Neste capítulo serão descritas todas as atividades relacionadas ao desenvolvimento do Sistema DispMed, abordando desde o contexto da sua criação até as decisões técnicas adotadas para sua implementação.

4.1 Introdução do DispMed

O acesso a medicamentos é um dos pilares fundamentais para a efetivação do direito à sáude garantido pela Constituição Federal. No entanto, a escassez ou indisponibilidade de remédios em farmácias públicas ainda é um desafio recorrente enfrentado por milhões de brasileiros, muitas vezes por não possuir o conhecimento sobre o local de fornecimento de determinado remédio. Por este motivo, os cidadãos deslocam-se de suas regiões desnecessariamente, em busca de um medicamento que poderia ser encontrado mais próximo de sua residência.

Aliado à este problema, este projeto propõe o desenvolvimento de um sistema *web* com intuito de mapear os locais de dispensa e quais são os medicamentos oferecidos pela rede municipal de saúde, com o objetivo de oferecer mais transparência à população e otimizar o processo para encontrar a localidade mais próxima que oferte determinado medicamento.

Com isso, espera-se promover maior eficiência no alcance da informação aos seus usuários, garantindo que os medicamentos cheguem efetivamente à população mais necessitada.

4.2 Requisitos Funcionais

Os requisitos funcionais identificados para construção da aplicação são:

- **RF001:** O sistema deve permitir ao usuário verificar se o medicamento desejado é disponibilizado pela prefeitura.
- **RF002:** O sistema deve informar os locais (farmácias ou unidades de saúde) onde o medicamento está disponível.
- **RF003:** O sistema deve informar os documentos exigidos para que o usuário consiga retirar a medicação.
- **RF004:** O sistema deve fornecer informações sobre a concentração, composição e forma farmacêutica do medicamento.
- **RF005:** O sistema deve permitir a visualização das farmácias cadastradas, incluindo suas informações básicas (nome, endereço, telefone, horário de funcionamento).
- **RF006:** O sistema deve disponibilizar um botão que direcione o usuário para um aplicativo de localização (*Google Maps, Waze,* entre outros).

4.3 Requisitos não funcionais

Os requisitos não funcionais estabelecem critérios técnicos e de qualidade para o funcionamento adequado da aplicação:

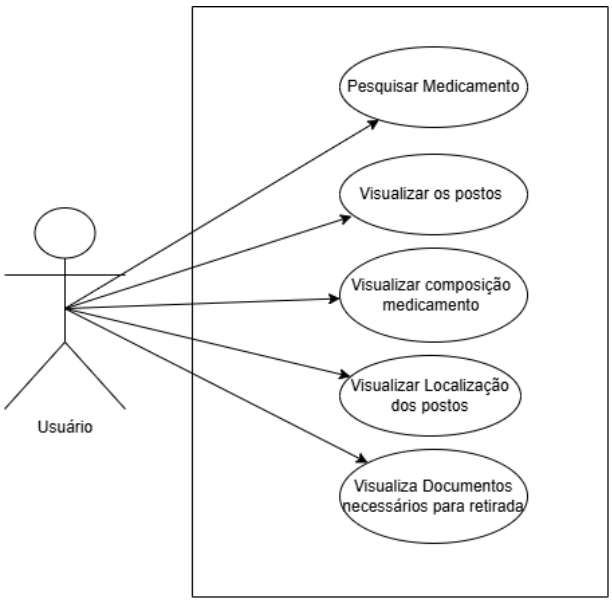
- **RNF006:** O sistema deve estar em conformidade com a Lei Geral de Proteção de Dados (LGPD), assegurando a privacidade dos dados dos usuários.
- **RNF007:** O sistema deve ser compatível com os principais navegadores de internet: *Google Chrome, Mozilla Firefox, Safari* e *Microsoft Edge*.
- **RNF008:** O sistema deve funcionar corretamente em dispositivos móveis com sistemas operacionais *Android* e *iOS*.

4.4 Caso de Uso

Para representar de forma clara e objetiva as funcionalidades do nosso sistema, foi elaborado um Diagrama de Caso de Uso. Esse tipo de diagrama é bastante utilizado na modelagem de sistemas orientados a objetos e tem como objetivo descrever as interações entre os atores (usuários ou sistemas externos) e o sistema em si, destacando os principais serviços que a aplicação irá oferecer.

Neste sistema, o ator principal é o usuário, representando o cidadão que acessa o site com a finalidade de consultar medicamentos oferecidos por unidades médicas do governo municipal de Salvador. O sistema permitirá que esse usuário busque medicamentos específicos e visualize informações, como a unidade responsável pela dispensação e a localização dessas unidades. A Figura 2 ilustra o caso de uso da aplicação DispMed.

Figura 2 – DispMed - Caso de uso



Autoria própria

4.5 Modelo da arquitetura - C4

O Modelo C4 é uma abordagem moderna para descrever arquiteturas de *software*, estruturada em diferentes níveis de detalhe: Contexto, Contêineres, Componentes e Código. Neste trabalho, foram elaborados os quatro níveis do modelo, cada um oferecendo uma perspectiva complementar sobre a estrutura e funcionamento do sistema.

O objetivo da utilização do Modelo C4 é fornecer uma visão clara, objetiva e compreensível da arquitetura do sistema desenvolvido, que tem como finalidade intermediar a busca por medicamentos disponibilizados em locais públicos.

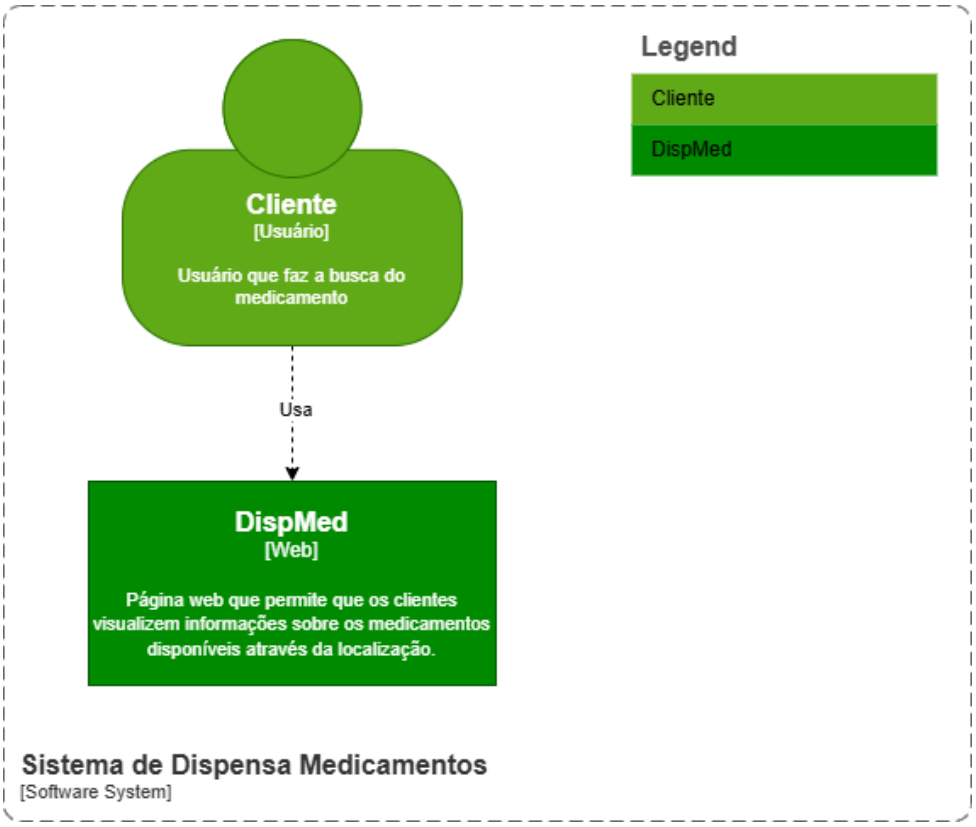
4.5.1 Nível C1: Diagrama de Contexto

Este nível representa a interação entre os usuários e o sistema como um todo, destacando os limites do sistema e suas principais dependências externas. A seguir, são apresentados os principais elementos envolvidos:

- **Usuários:** pessoas que buscam por medicamentos. Elas acessam o sistema *DispMED* para localizar medicamentos disponíveis em locais públicos.
- **DispMED:** sistema que atua como intermediador, conectando os usuários aos locais que possuem os medicamentos em estoque. A aplicação é desenvolvida com as tecnologias *Quasar* e *Vue.js*.

A Figura 3 representa o Diagrama da aplicação DispMed.

Figura 3 – C1 - Diagrama de Contexto do DispMed



Autoria própria

No diagrama de contexto, observa-se o fluxo principal: o usuário acessa o sistema *DispMed*, que por sua vez interage com o *backend* para validar e processar as informações necessárias.

4.5.2 Nível C2: Diagrama de Contêineres

Este nível do modelo C4 descreve a composição do sistema a partir de múltiplos contêineres, tais como aplicações, serviços e bases de dados, que interagem entre si para atender aos requisitos funcionais. A seguir, apresentam-se os principais elementos dessa estrutura.

- **Usuários:** continuam sendo os iniciadores das interações, acessando o sistema por meio de uma *interface web*.
- **DispMed (*Frontend*):** *contêiner* desenvolvido com o *Quasar Framework* e *Vue.js*, responsável por gerenciar a *interface* de comunicação entre o usuário e os serviços do sistema.
- ***Backend:*** contêiner implementado com *Spring Boot*, responsável pela lógica de negócios, controle de usuários e gestão das informações sobre a disponibilidade de medicamentos e suas respectivas localidades.
- **Banco de Dados:** banco de dados relacional *PostgreSQL*, utilizado para armazenar informações sobre medicamentos, disponibilidade por localidade, quantidades e dados dos distritos sanitários.

A Figura 4 representa o Diagrama de Contêineres do DispMed.

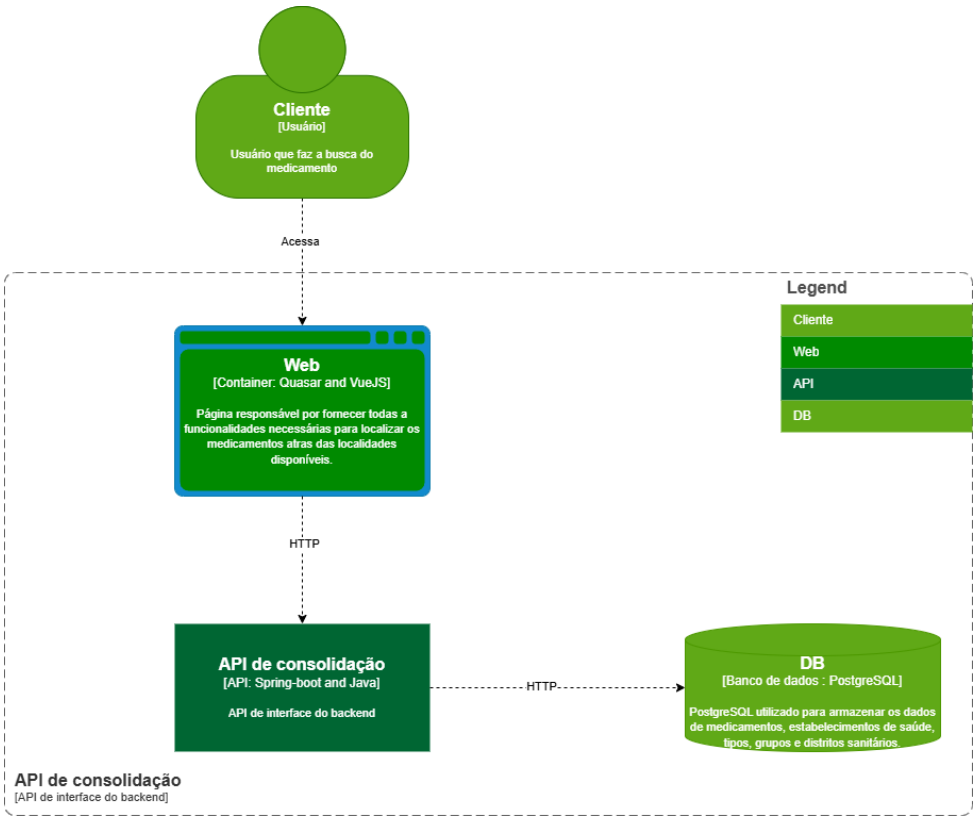
O diagrama de contêineres destaca como o sistema é estruturado em contêineres (módulos ou aplicações executáveis) com responsabilidades bem definidas, favorecendo a organização do software, a separação de preocupações e a escalabilidade da solução.

4.5.3 Nível C3: Diagrama de Componentes

Na camada de componentes, é possível visualizar detalhes dos principais componentes do *backend*. Neste diagrama, estão presentes os seguintes elementos:

- ***Controller***: responsável por receber as requisições e gerenciar as funcionalidades do sistema.
- ***EstabeleciSaudeController***: gerencia as requisições relacionadas aos estabelecimentos de saúde.
- ***MedicamentoController***: gerencia as requisições relacionadas aos medicamentos.
- ***Service***: a camada *Service* (*EstabelecimentoSaudeService*, *MedicamentoService*) é inteiramente responsável pela implementação das regras de negócio. Ou seja, nesta camada está empregada toda a lógica necessária utilizando os respectivos repositórios, de forma que o *controller* possa se manter focado apenas na recepção das requisições e no envio das respostas.

Figura 4 – C2 -Diagrama de Contêineres do DispMed

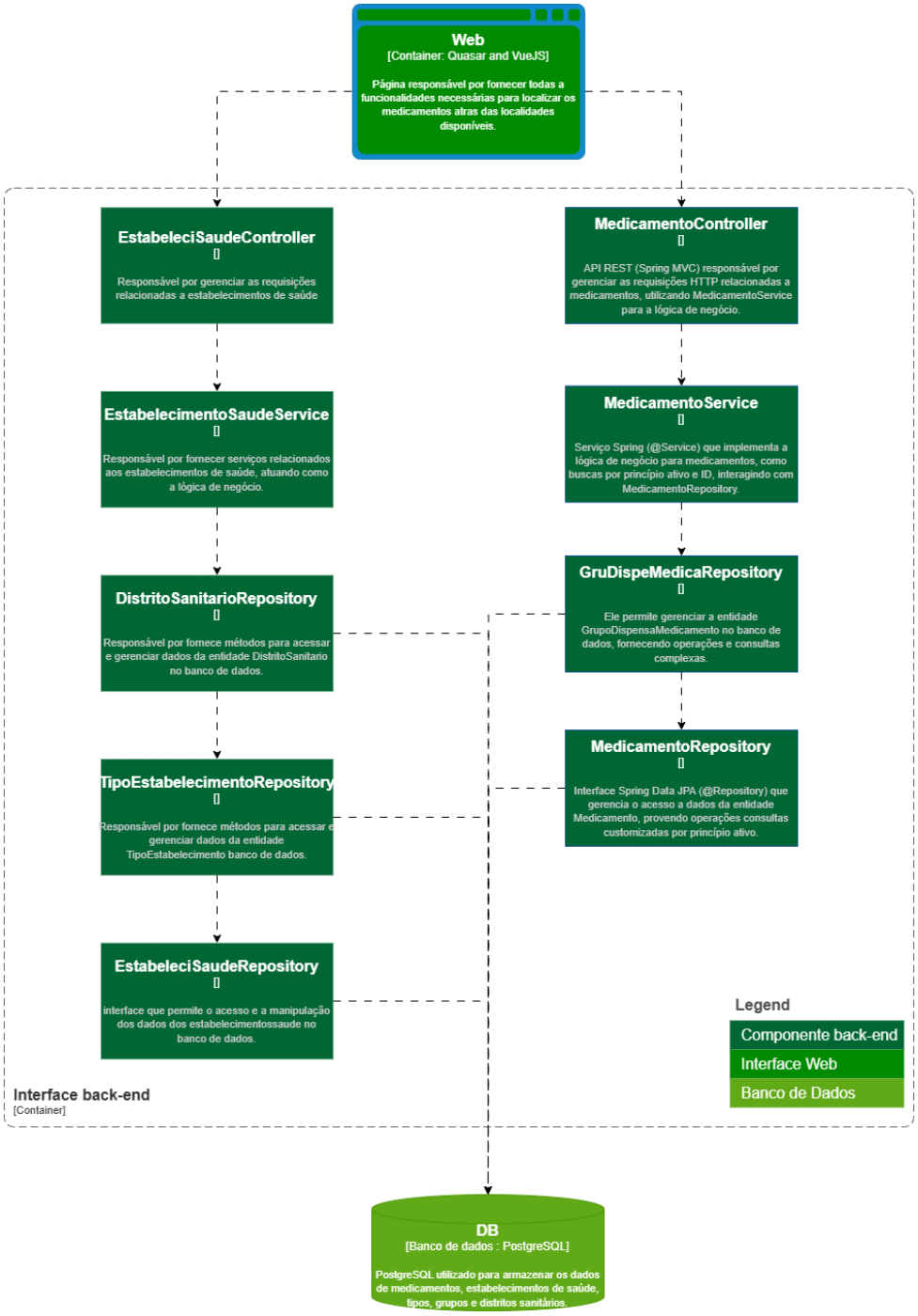


Autoria própria

- ***Repository***: responsável pela camada de acesso aos dados. Atua sobre as entidades do sistema, que representam as tabelas do banco de dados. Os repositórios, como *EstabeleciSaudeRepository*, *MedicamentoRepository*, *DistritoSanitarioRepository*, *GrupoDispensaMedicamentoRepository* e *TipoEstabelecimentoRepository*, têm como objetivo manipular os dados das respectivas entidades. O acesso real ao banco de dados é realizado de forma indireta, por meio do *JPA/Hibernate*.

A Figura 5 representa o Diagrama de Componentes do DispMed.

Figura 5 – C3 - Diagrama de Componentes do DispMed



Autoria própria

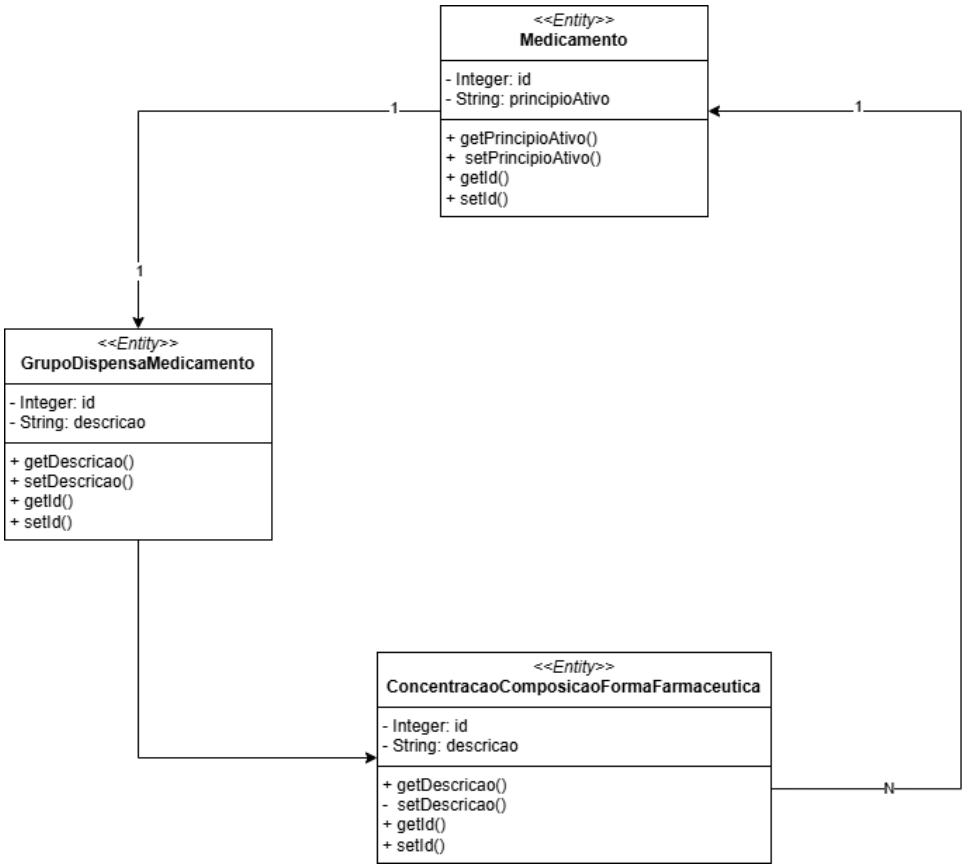
Por fim, o Diagrama de Código, que consiste na seleção de um componente específico, representado por meio de um diagrama *UML*, permitindo uma compreensão clara da estrutura e funcionalidade do código-fonte.

4.5.4 Nível C4: Diagrama de Código

Este diagrama de classe representa conforme a Figura 6 o nível mais detalhado do modelo C4, sendo denominado *diagrama de classe*. Nele, estão descritas as principais entidades do

sistema relacionadas aos medicamentos, bem como os seus respectivos relacionamentos.

Figura 6 – C4 - Diagrama de Código do DispMed



Autoria própria

Entidades

- ***Medicamento***: representa um medicamento individual, contendo atributos como um identificador único (*id*) e o *principioAtivo* (princípio ativo). Inclui métodos para acessar e modificar esses atributos.

- ***GrupoDispensaMedicamento***: representa o grupo de dispensa ao qual um medicamento pertence, com atributos como *id* e *descricao*. Possui métodos como *getDescricao*, *setDescricao*, *getId* e *setId*. Existe um relacionamento de "1 para 1" entre *Medicamento* e *GrupoDispensaMedicamento*, indicando que cada medicamento pertence a um único grupo de dispensa.

- ***ConcentracaoComposicaoFormaFarmaceutica***: detalha informações relacionadas à concentração, composição e forma farmacêutica do medicamento, com atributos como *id* e *descricao*. Os métodos disponíveis incluem *getDescricao*, *setDescricao*, *getId* e *setId*. Existe um relacionamento de "1 para N" entre *Medicamento* e *ConcentracaoComposicaoFormaFarmaceutica*, ou seja, um medicamento pode ter

múltiplas concentrações, composições ou formas farmacêuticas associadas, e uma mesma composição pode estar vinculada a diversos medicamentos.

Atributos e Métodos

Cada entidade define atributos essenciais, como *id* e *descricao* (ou *principioAtivo*, no caso da entidade *Medicamento*). Além disso, cada uma implementa métodos padrão de consulta (*get*) e atualização (*set*) desses atributos.

O diagrama da figura 5 ilustra a estrutura fundamental do sistema, evidenciando como essas entidades se relacionam para viabilizar o gerenciamento eficaz das informações referentes aos medicamentos.

5 Desenvolvimento

Esta seção tem como objetivo descrever o processo de desenvolvimento do projeto, com ênfase na arquitetura *backend*, abordando suas etapas fundamentais e a estrutura adotada. Serão também descritas em detalhes todas as tecnologias utilizadas, evidenciando sua relevância e a forma como foram aplicadas no contexto da solução proposta.

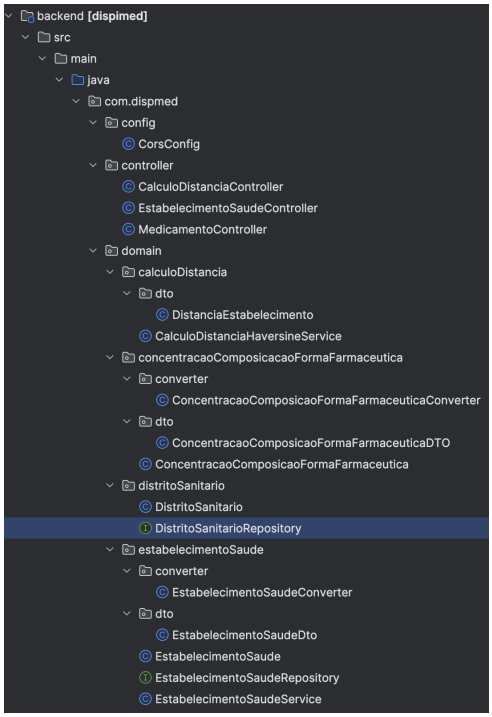
5.1 *BackEnd*

Para o desenvolvimento do *backend*, foram utilizadas as tecnologias *Java* e o *framework Spring Boot*, que oferece suporte e simplifica a construção de aplicações robustas. O *Spring Boot* facilita a criação de *APIs* escaláveis, seguras e de fácil manutenção, oferecendo suporte integrado à injeção de dependência, gerenciamento de configurações e ferramentas que promovem um desenvolvimento ágil.

Além disso, foi escolhido o *PostgreSQL* como sistema gerenciador de banco de dados. A integração entre o *PostgreSQL* e o *Spring Boot* é facilitada pelo uso do *Spring Data JPA*, que permite o mapeamento eficiente das entidades, a abstração da camada de persistência e maior produtividade no desenvolvimento das operações com o banco de dados.

A Figura 7 ilustra a organização do projeto.

Figura 7 – DispMed - Estrutura do Projeto



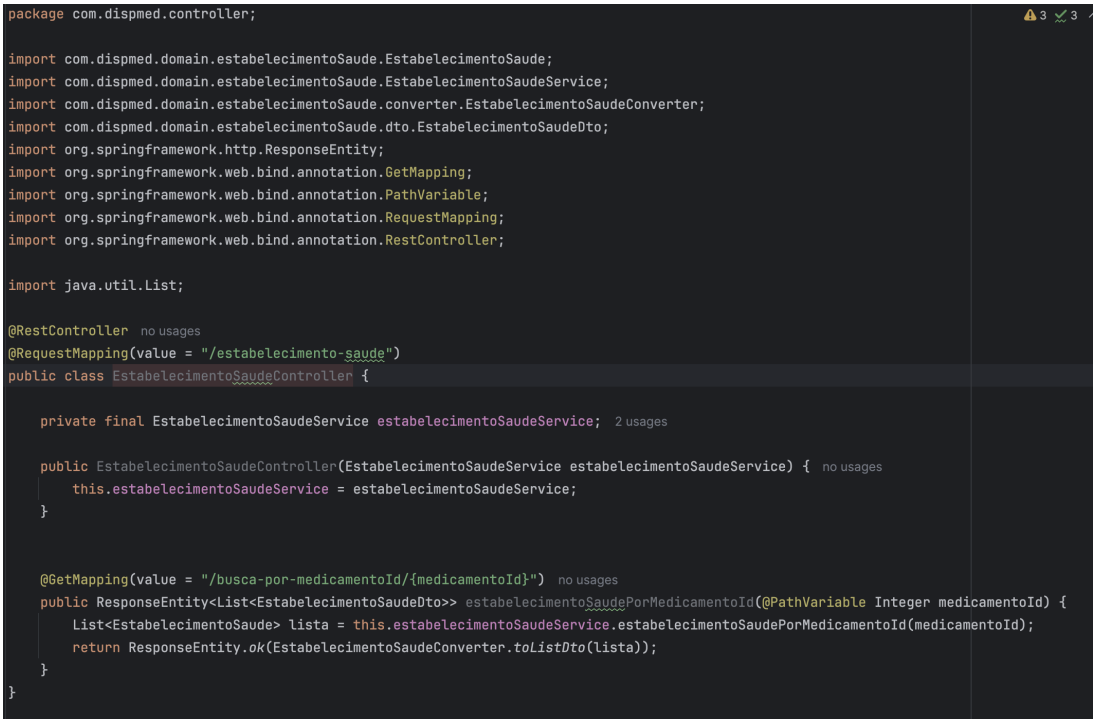
Autoria própria

É importante destacar que o conceito de *Clean Architecture* vai além da simples organização de pastas no projeto, essa estrutura visual representa apenas uma camada superficial de sua aplicação. A essência da *Clean Architecture* reside na separação clara de responsabilidades e na independência entre as camadas da aplicação. Essa abordagem garante que o núcleo do sistema composto pelas regras de negócio e pela lógica de domínio permaneça isolado e protegido de alterações nas camadas externas, como *interfaces* gráficas, *frameworks*, bibliotecas ou sistemas de banco de dados.

5.2 *Controller:*

Na camada de *Controllers*, representada através de Figura 8, a estrutura foi concebida com foco na organização e na alta eficiência do projeto, visando sempre manter a manutenibilidade do sistema. Essa abordagem facilita a identificação e correção de problemas futuros.

Figura 8 – DispMed - Controller



```java
package com.dispmed.controller;

import com.dispmed.domain.estabelecimentoSaude.EstabelecimentoSaude;
import com.dispmed.domain.estabelecimentoSaude.EstabelecimentoSaudeService;
import com.dispmed.domain.estabelecimentoSaude.converter.EstabelecimentoSaudeConverter;
import com.dispmed.domain.estabelecimentoSaude.dto.EstabelecimentoSaudeDto;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController  no usages
@RequestMapping(value = "/estabelecimento-saude")
public class EstabelecimentoSaudeController {

    private final EstabelecimentoSaudeService estabelecimentoSaudeService;  2 usages

    public EstabelecimentoSaudeController(EstabelecimentoSaudeService estabelecimentoSaudeService) {  no usages
        this.estabelecimentoSaudeService = estabelecimentoSaudeService;
    }


    @GetMapping(value = "/busca-por-medicamentoId/{medicamentoId}")  no usages
    public ResponseEntity<List<EstabelecimentoSaudeDto>> estabelecimentoSaudePorMedicamentoId(@PathVariable Integer medicamentoId) {
        List<EstabelecimentoSaude> lista = this.estabelecimentoSaudeService.estabelecimentoSaudePorMedicamentoId(medicamentoId);
        return ResponseEntity.ok(EstabelecimentoSaudeConverter.toListDto(lista));
    }
}
```

Autoria própria

Ao optar por centralizar a lógica de controle em uma camada específica, promove-se uma separação clara entre a interface de entrada e as regras de negócio, contribuindo para um sistema mais coeso e testável. Essa organização também permite maior escalabilidade, uma vez que novos controladores podem ser adicionados de forma padronizada e com baixo impacto nas demais camadas da aplicação.

5.3 DTO:

O *DTO* (*Data Transfer Object*) é um padrão de projeto utilizado para transportar dados entre diferentes camadas de um sistema, especialmente entre a camada de apresentação (como *APIs* ou interfaces gráficas) e as camadas de negócio ou persistência.

Um *DTO* é, basicamente, um objeto que contém apenas dados, sem lógica de negócio. O DTO do DispMed esta representado na Figura 9.

Figura 9 – DispMed - DTO

```java
package com.dispmed.domain.estabelecimentoSaude.dto;

import com.dispmed.domain.tipoEstabelecimento.dto.TipoEstabelecimentoDto;
import lombok.*;

import java.math.BigDecimal;

@Getter  6 usages
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class EstabelecimentoSaudeDto {

    private Integer id;
    private String nome;
    private Double latitude;
    private Double longitude;
    private String endereco;
    private String linkGoogle;
    private String distritoSanitario;
    private TipoEstabelecimentoDto tipoEstabelecimento;
}
```

Autoria própria

No projeto, o principal objetivo da utilização dos *DTOs* é evitar a exposição direta das entidades, facilitar a organização e a estrutura dos dados que são enviados, além de melhorar a performance do sistema, uma vez que apenas as informações necessárias são transportadas entre as camadas.

5.4 *Service*:

A camada *Service* desempenha um papel fundamental no sistema, sendo responsável por conter a lógica de negócio. Funciona como um intermediário entre os *controllers* e os *repositórios*. A Figura 10 representa o *service* de Estabelecimento do sistema DispMed.

Figura 10 – DispMed - SERVICE

```java
package com.dispmed.domain.estabelecimentoSaude;

import com.dispmed.domain.medicamento.Medicamento;
import com.dispmed.domain.medicamento.MedicamentoService;
import org.springframework.stereotype.Service;

import java.util.List;

@Service  3 usages
public class EstabelecimentoSaudeService {

    private final EstabelecimentoSaudeRepository estabelecimentoSaudeRepository;  1 usage
    private final MedicamentoService medicamentoService;  2 usages
    public EstabelecimentoSaudeService(EstabelecimentoSaudeRepository estabelecimentoSaudeRepository, MedicamentoService medicamentoService) {  no usages
        this.estabelecimentoSaudeRepository = estabelecimentoSaudeRepository;
        this.medicamentoService = medicamentoService;
    }

    public List<EstabelecimentoSaude> estabelecimentoSaudePorMedicamentoId(Integer medicamentoId) {  1 usage
        Medicamento medicamento = medicamentoService.buscarPorId(medicamentoId);
        return medicamento.getGrupoDispensaMedicamento().getEstabelecimentosSaude();
    }

}
```

Autoria própria

Essa camada é essencial para o tratamento das regras específicas do domínio, como validações, autorizações e interações entre entidades, antes que os dados sejam persistidos no banco de dados ou retornados ao usuário.

5.5 *Repository*:

A camada de *Repository* é responsável por realizar as operações de persistência de dados. As interfaces de repositório estendem a classe *JpaRepository*, fornecendo métodos prontos para a manipulação das entidades.

A Figura 11 uma classe *repository* do DispMed.

Figura 11 – DispMed - REPOSITORY

```java
package com.dispmed.domain.medicamento;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository  2 usages
public interface MedicamentoRepository extends JpaRepository<Medicamento, Integer> {

    @Query(nativeQuery = true, value = "SELECT * FROM medicamento m WHERE m.principio_ativo ILIKE :principioAtivo")
    List<Medicamento> findMedicamentoPorPrincipioAtivo(@Param("principioAtivo") String principioAtivo);
}
```

Autoria própria

No desenvolvimento do DispMed, adotou-se uma arquitetura em camadas fundamentada no princípio da separação de responsabilidades. Essa abordagem estruturada favoreceu a organização do código-fonte, facilitando a manutenção evolutiva do sistema, além de promover a reutilização de componentes e a escalabilidade da aplicação. Com isso, foi possível garantir maior clareza na estrutura do projeto e contribuir para a qualidade técnica da solução desenvolvida.

5.6 *Entity*:

As *entities* representam a estrutura e a organização dos dados no banco de dados, refletindo os conceitos fundamentais do domínio da aplicação. Elas são responsáveis por modelar os dados persistentes, incluindo atributos, relacionamentos e restrições, servindo como base para a construção das regras de negócio e garantindo a integridade da informação ao longo do ciclo de vida do sistema.

A Figura 12 uma classe *entity* do DispMed.

Figura 12 – DispMed - Entity

```java
package com.dispmed.domain.medicamento;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository 2 usages
public interface MedicamentoRepository extends JpaRepository<Medicamento, Integer> {

    @Query(nativeQuery = true, value = "SELECT * FROM medicamento m WHERE m.principio_ativo ILIKE :principioAtivo")
    List<Medicamento> findMedicamentoPorPrincipioAtivo(@Param("principioAtivo") String principioAtivo);
}
```

Autoria própria

No projeto, elas modelam os elementos centrais da aplicação, refletindo diretamente as tabelas e os relacionamentos utilizados para armazenar e acessar as informações de forma eficaz.

- ***DistritoSanitario***: responsável por definir as regiões administrativas responsáveis por agrupar os estabelecimentos de saúde.
- ***EstabelecimentoSaude***: representa os locais onde os medicamentos são fornecidos à população.
- ***TipoEstabelecimento***: realiza a categorização do tipo de unidade de saúde, como UBS, UPA e CAPS.
- ***Medicamento***: contém os dados dos medicamentos disponíveis, como nome, princípio ativo, forma farmacêutica e concentração.

No geral, essas entidades se relacionam entre si de forma lógica, para que assim possam permitir consultas e gerenciar com eficiência os dados.

5.7 PostgreSQL:

De acordo com Momjian (2001), o nome *PostgreSQL* deriva de *Post-Ingres*, em referência ao sistema de gerenciamento de banco de dados *Ingres* (*INteractive Graphics and REtrieval System*).

A principal motivação para a escolha do *PostgreSQL* como sistema gerenciador de banco de dados neste projeto foi sua robustez e reputação consolidada no mercado. Por se tratar de um banco de dados relacional e de código aberto, a decisão precisava ser assertiva, visando garantir desempenho, confiabilidade e segurança.

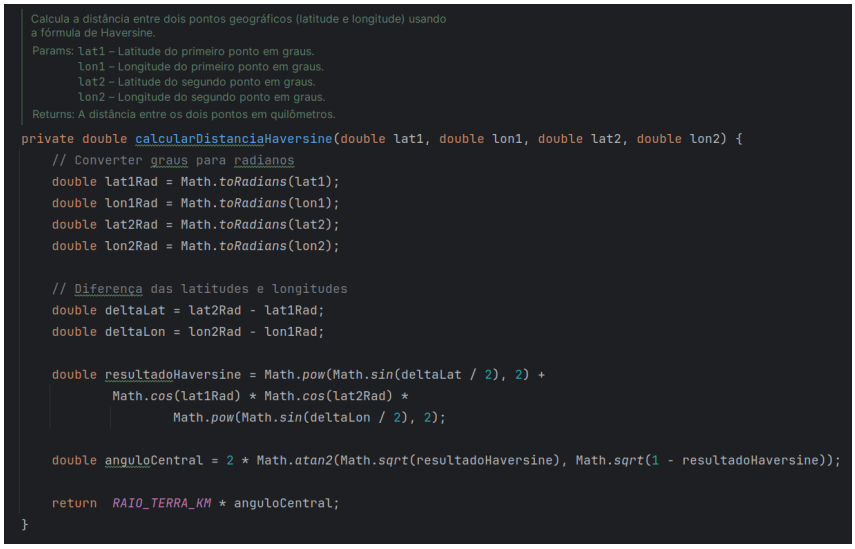
Adicionalmente, sua integração com o ecossistema Java é extremamente eficiente, contando com suporte a *drivers JDBC* e compatibilidade total com os *frameworks Hibernate* e *JPA*. No contexto deste projeto, o *PostgreSQL* não atua apenas como repositório de dados, mas configura-se como um componente ativo e fundamental para o funcionamento e a organização da aplicação.

5.8 Aplicação do *Clean Code*:

Durante o desenvolvimento do sistema, a adoção dos princípios do *Clean Code* foi fundamental para garantir a clareza, a manutenibilidade e a testabilidade do código.

No código implementado, buscou-se sempre utilizar nomes de variáveis e funções que fossem curtos, porém descritivos, facilitando o entendimento. Na Figura 13 mostra a aplicação do *Clean Code* em um trecho do código do DispMed.

Figura 13 – Aplicação do *Clean Code* no DispMed

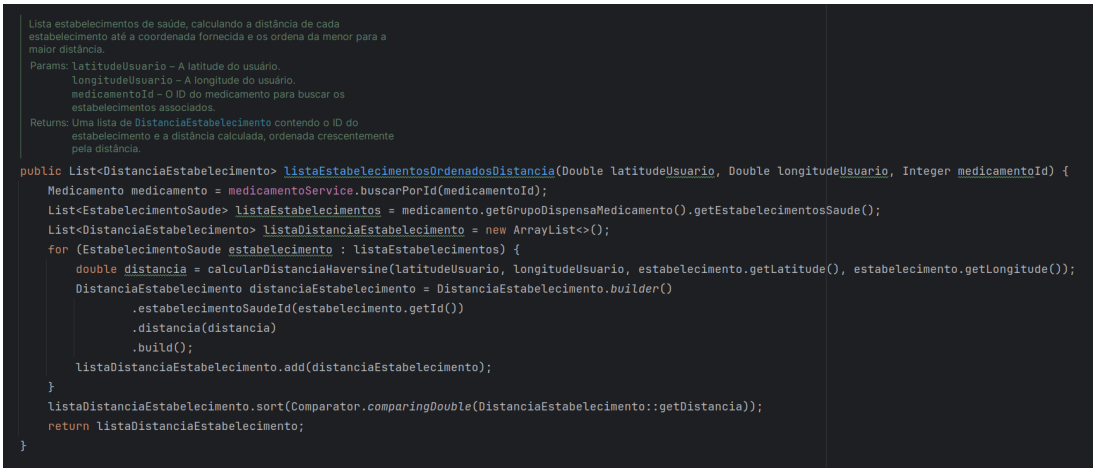


```
Calcula a distância entre dois pontos geográficos (latitude e longitude) usando
a fórmula de Haversine.
Params: lat1 – Latitude do primeiro ponto em graus.
        lon1 – Longitude do primeiro ponto em graus.
        lat2 – Latitude do segundo ponto em graus.
        lon2 – Longitude do segundo ponto em graus.
Returns: A distância entre os dois pontos em quilômetros.
private double calcularDistanciaHaversine(double lat1, double lon1, double lat2, double lon2) {
    // Converter graus para radianos
    double lat1Rad = Math.toRadians(lat1);
    double lon1Rad = Math.toRadians(lon1);
    double lat2Rad = Math.toRadians(lat2);
    double lon2Rad = Math.toRadians(lon2);

    // Diferença das latitudes e longitudes
    double deltaLat = lat2Rad - lat1Rad;
    double deltaLon = lon2Rad - lon1Rad;

    double resultadoHaversine = Math.pow(Math.sin(deltaLat / 2), 2) +
            Math.cos(lat1Rad) * Math.cos(lat2Rad) *
                    Math.pow(Math.sin(deltaLon / 2), 2);

    double anguloCentral = 2 * Math.atan2(Math.sqrt(resultadoHaversine), Math.sqrt(1 - resultadoHaversine));

    return  RAIO_TERRA_KM * anguloCentral;
}
```

Autoria própria

Outro princípio importante aplicado foi a separação de responsabilidades. Ao dividir as funcionalidades do sistema em componentes especializados, foi possível promover maior coesão interna e baixo acoplamento entre as partes do código. Essa organização favorece tanto a escalabilidade, quanto a reutilização de componentes, além de simplificar a realização de testes unitários, conforme visualizado na Figura 14.

Figura 14 – *Clean Code*:Princípio de Responsabilidade única.



```
Lista estabelecimentos de saúde, calculando a distância de cada
estabelecimento até a coordenada fornecida e os ordena da menor para a
maior distância.
Params: latitudeUsuario – A latitude do usuário.
        longitudeUsuario – A longitude do usuário.
        medicamentoId – O ID do medicamento para buscar os
        estabelecimentos associados.
Returns: Uma lista de DistanciaEstabelecimento contendo o ID do
        estabelecimento e a distância calculada, ordenada crescentemente
        pela distância.
public List<DistanciaEstabelecimento> listaEstabelecimentosOrdenadosDistancia(Double latitudeUsuario, Double longitudeUsuario, Integer medicamentoId) {
    Medicamento medicamento = medicamentoService.buscarPorId(medicamentoId);
    List<EstabelecimentoSaude> listaEstabelecimentos = medicamento.getGrupoDispensaMedicamento().getEstabelecimentosSaude();
    List<DistanciaEstabelecimento> listaDistanciaEstabelecimento = new ArrayList<>();
    for (EstabelecimentoSaude estabelecimento : listaEstabelecimentos) {
        double distancia = calcularDistanciaHaversine(latitudeUsuario, longitudeUsuario, estabelecimento.getLatitude(), estabelecimento.getLongitude());
        DistanciaEstabelecimento distanciaEstabelecimento = DistanciaEstabelecimento.builder()
                .estabelecimentoSaudeId(estabelecimento.getId())
                .distancia(distancia)
                .build();
        listaDistanciaEstabelecimento.add(distanciaEstabelecimento);
    }
    listaDistanciaEstabelecimento.sort(Comparator.comparingDouble(DistanciaEstabelecimento::getDistancia));
    return listaDistanciaEstabelecimento;
}
```

Autoria própria

O método apresentado, `listaEstabelecimentosOrdenadosDistancia`, exemplifica outra aplicação prática dos princípios de *Clean Code* no desenvolvimento do sistema. Sua principal função é listar estabelecimentos de saúde associados a um determinado medicamento, calcular a distância de cada um em relação à localização do usuário e ordenar o resultado de forma crescente pela distância.

A clareza e a legibilidade do código são garantidas pelo uso de nomes descritivos para variáveis e métodos, tornando o propósito de cada elemento facilmente compreensível. O método é coeso, realizando apenas uma tarefa principal, o que segue o princípio da responsabilidade única e facilita a manutenção futura.

Além disso, a modularização é evidenciada pela delegação do cálculo de distância para uma função específica (`calcularDistanciaHaversine`) e pela utilização do padrão *builder* para a criação dos objetos `DistanciaEstabelecimento`, promovendo reutilização e organização. Por fim, a ordenação da lista por meio de uma expressão *lambda* contribui para um código mais conciso e moderno, conforme apresentado na Figura 15.

Figura 15 – Clean Code: Utilização de Lambda

```
listaDistanciaEstabelecimento.sort(Comparator.comparingDouble(DistanciaEstabelecimento::getDistancia));
return listaDistanciaEstabelecimento;
```

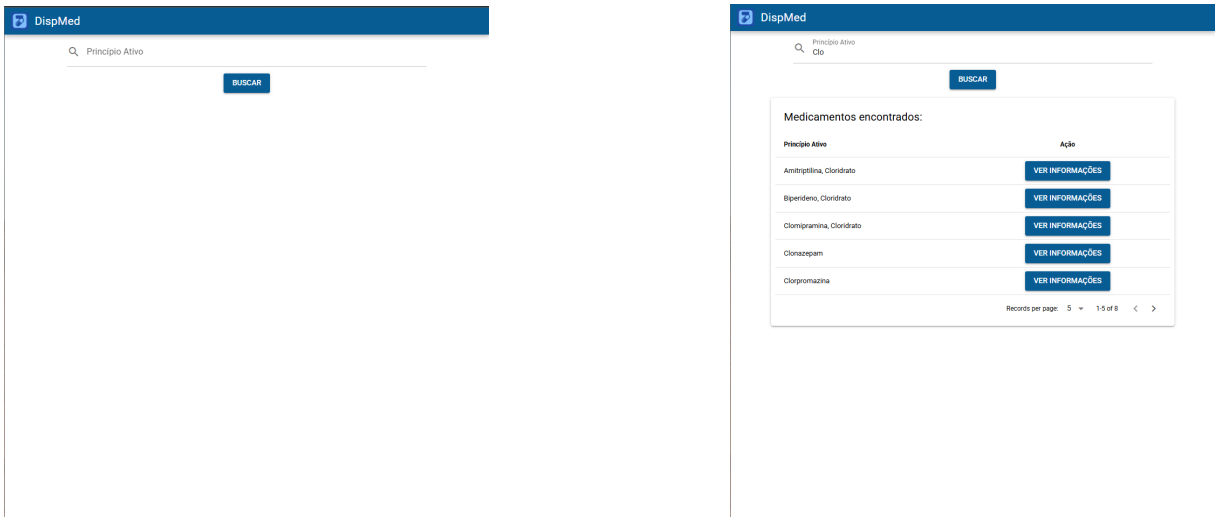
Autoria própria

Essas práticas, em conjunto, resultam em um código limpo, de fácil entendimento e preparado para futuras evoluções, alinhado com as melhores práticas de engenharia de *software*.

5.9 *FrontEnd*

A tela inicial(Figura 16) da aplicação permite ao usuário selecionar o tipo de medicamento desejado e realizar uma busca pelos locais de dispensa mais próximos à sua localização. Após a seleção do medicamento, é exibida a opção "Ver informações", que apresenta uma lista dos estabelecimentos de saúde que realizam a distribuição do respectivo princípio ativo.

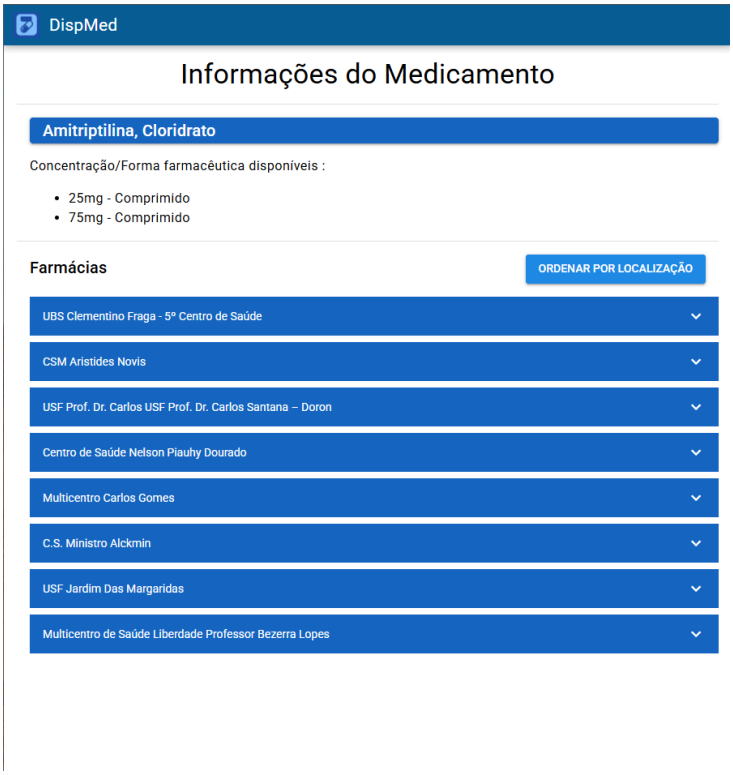
Figura 16 – Tela de busca



Autoria Própria

Essa funcionalidade visa facilitar o acesso da população às informações sobre a disponibilidade de medicamentos (Figura 17), promovendo maior eficiência na localização dos pontos de distribuição e contribuindo para a otimização do atendimento nas Unidades de Saúde.

Figura 17 – Concentração / Farmácias



Autoria Própria

Em seguida, são exibidos os locais de dispensação que disponibilizam o princípio ativo previamente selecionado pelo usuário. A aplicação integra a funcionalidade de geolocaliza-

ção do navegador, conforme ilustrado na Figura 18, a qual é ativada mediante autorização do próprio usuário. Com a permissão concedida, os estabelecimentos são automaticamente ordenados por proximidade em relação à localização atual, facilitando a identificação do ponto de retirada mais acessível.

Figura 18 – Mapa

Autoria Própria

Essa funcionalidade tem como objetivo oferecer uma experiência mais eficiente e personalizada, permitindo que o usuário identifique com agilidade os pontos de distribuição mais acessíveis. Com isso, busca-se otimizar o tempo de deslocamento e facilitar o acesso ao medicamento necessário, contribuindo para a efetividade do atendimento.

Ao expandir as divisões referentes aos locais de dispensa, o sistema exibe o endereço completo de cada estabelecimento, fornecendo informações detalhadas que facilitam a identificação do ponto de distribuição. Esses detalhes pdem ser visualizados no DispMed, conforme ilustrado na Figura 19.

Figura 19 – Informações do Medicamento

Autoria Própria

O sistema também é capaz de ordenar as farmácias mais próximas do usuário, de acordo a sua localização, facilitando que o mesmo encontre o local na distância o mais curta possível. Além disso, é incorporado um *iframe* com a localização do estabelecimento no *Google Maps*, permitindo ao usuário visualizar o local diretamente no mapa. Essa funcionalidade também atua como um *link*, redirecionando o usuário para a plataforma oficial do *Google Maps*, caso deseje obter uma visão mais ampla da região ou traçar rotas até o local. Essa abordagem visa oferecer maior praticidade e acessibilidade, integrando recursos tecnológicos que facilitam o deslocamento até o ponto de dispensa mais adequado.

6 Conclusão

O desenvolvimento do sistema *DispMed* demonstrou, na prática, como os princípios do *Clean Code* e do *Clean Architecture* podem ser aplicados com eficácia para resolver um problema social relevante: a dificuldade de acesso da população às informações sobre a disponibilidade de medicamentos nas unidades públicas de saúde. Ao unir boas práticas da engenharia de *software* a um propósito social, o projeto conseguiu entregar uma solução tecnológica organizada, de fácil manutenção e com grande potencial de impacto.

O uso de princípios como *SOLID*, *KISS*, *YAGNI* e *TDD* não apenas elevou a qualidade técnica do código, como também facilitou a escalabilidade e a sustentabilidade

da aplicação. A adoção do *Modelo C4* permitiu uma visualização clara da arquitetura do sistema em seus diferentes níveis, promovendo a compreensão por parte de todos os envolvidos no projeto e fortalecendo a documentação da solução.

Além da base técnica sólida, o projeto destacou-se por seu foco no usuário final, sendo eles os cidadãos da cidade de Salvador, que enfrentam diariamente dificuldades para localizar medicamentos básicos. Ao centralizar informações sobre disponibilidade, localização e dados farmacêuticos, o sistema *DispMed* oferece praticidade e eficiência, contribuindo diretamente para o fortalecimento das políticas públicas de saúde.

Portanto, este trabalho evidencia que a combinação entre boas práticas de desenvolvimento e sensibilidade social pode resultar em soluções tecnológicas eficazes, sustentáveis e com alto valor para a comunidade. A continuidade e evolução deste projeto abrem portas para sua replicação em outras regiões, reforçando a importância da tecnologia como aliada da saúde pública.

7 Trabalhos Futuros

Com a implementação da versão inicial do sistema *DispMed*, surgem diversas oportunidades para expansão e aprimoramento da solução, tanto do ponto de vista técnico quanto funcional. Como trabalhos futuros, destacam-se as seguintes possibilidades:

- Implementação da Lei Geral de Proteção de Dados Pessoais (LGPD), Lei nº 13.709/2018;
- Implementação de um painel administrativo para os gestores das unidades de saúde, possibilitando o gerenciamento de estoque em tempo real e o envio de notificações à população;
- Inclusão de funcionalidades de acessibilidade, como leitura de tela e tradução em Libras, tornando o sistema mais inclusivo;
- Desenvolvimento de um aplicativo móvel multiplataforma (*Android e iOS*), com notificações *push* e uso intensivo de geolocalização para facilitar o acesso em regiões de baixa conectividade;
- Aplicação de técnicas de inteligência artificial, como previsão de escassez de medicamentos com base em dados históricos e sazonais, auxiliando no planejamento logístico das unidades de saúde;
- Ampliação do escopo geográfico, permitindo replicar o sistema para outras cidades ou estados;
- Inclusão de funcionalidades de *feedback* dos usuários, permitindo avaliar a precisão das informações e a qualidade do atendimento nas unidades;

- Integração com sistemas de gestão da saúde municipal, possibilitando atualização automática de dados.

Referências

AZEVEDO, B. A. A. S. *Clean Code e SOLID na construção da aplicação Oratio: melhorando a manutenibilidade e escalabilidade*. Trabalho de Conclusão de Curso — Universidade Federal de Campina Grande, Campina Grande, Brasil, 2023.

BECK, K. *Test-Driven Development: By Example*. [S.l.]: Addison-Wesley, 2002.

BEIZER, B. *Software Testing Techniques*. 2. ed. New York: Van Nostrand Reinhold, 1990.

BROOKS, F. P. *The Mythical Man-Month: Essays on Software Engineering*. [S.l.]: Addison-Wesley, 1995.

CAIO, A. M. P.; OSWALDO, L. M. CÓdigo limpo: a importância da refatoração. *Faculdade de Tecnologia de Taquaritinga (Fatec)*, Taquaritinga, SP, Brasil, v. 19, n. 1, 2022.

CUNNINGHAM, W. The wycash portfolio management system. In: *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. [S.l.: s.n.], 1992.

FOWLER, M. *Refactoring: Improving the Design of Existing Code*. 2nd. ed. [S.l.]: Addison-Wesley, 2018.

HUNT, A.; THOMAS, D. *The Pragmatic Programmer*. [S.l.]: Addison-Wesley, 1999.

ISO/IEC. *Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models*. 2011. ISO/IEC 25010:2011. International Organization for Standardization.

JORDAN, A. et al. *Desenvolvimento de um Sistema de Voluntários para a Área Tech: um estudo de caso com Clean Architecture*. Salvador, Brasil: [s.n.], 2024. Escola de Tecnologias, Universidade Católica do Salvador.

MARTIN, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall, 2002. ISBN 9780135974445.

MARTIN, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st. ed. [S.l.]: Prentice Hall, 2008.

MARTIN, R. C. *Clean code: a handbook of agile software craftsmanship*. [S.l.]: Pearson Education, 2009.

MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. [S.l.]: Prentice Hall, 2017.

MCBREEN, P. J. *Software Craftsmanship: The New Imperative*. Boston: Addison-Wesley, 2001.

MCCONNELL, S. *Code Complete: A Practical Handbook of Software Construction*. 2nd. ed. [S.l.]: Microsoft Press, 2004.

PHELIPE, A. L.; FERNANDO, T. CÓdigo limpo: padrões e técnicas no desenvolvimento de software. *Faculdade de Tecnologia de Taquaritinga (FATEC)*, SP, Brasil, 2024.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 8ª. ed. [S.l.]: McGraw-Hill, 2014.

SOMMERVILLE, I. *Software Engineering*. 9. ed. Boston: Pearson, 2010.

TEKIN, S. *What is the Clean Architecture?* 2023. Accessed: 2025-07-10. Disponível em: <https://semihtekin.medium.com/what-is-the-clean-architecture-c80c2a2ff69a>.