UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÂO

IGOR MAURÍCIO DE CAMPOS

ARQUITETURA DE SOFTWARE PARA SISTEMA DE ANÁLISE RESPIRATÓRIA:
Respiratory Diagnostic Assistant

Recife
2025

IGOR MAURÍCIO DE CAMPOS

ARQUITETURA DE SOFTWARE PARA SISTEMA DE ANÁLISE RESPIRATÓRIA: Respiratory Diagnostic Assistant

Trabalho de Conclusão de Curso apresentado ao Departamento de Engenharia Elétrica da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação.

Orientador: Prof. Dr. Geraldo Leite Maia Júnior

Coorientadora: Profa. Dra. Shirley Lima Campos

Recife
2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Campos, Igor Maurício de .
Arquitetura de software para Sistema de análise respiratória: Respiratory Diagnostic Assistant / Igor Maurício de Campos. - Recife, 2025.
66 p. : il., tab.

Orientador(a): Geraldo Leite Maia Júnior
Cooorientador(a): Shirley Lima Campos
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Tecnologia e Geociências, Engenharia de Controle e Automação - Bacharelado, 2025.
Inclui referências.

1. Arquitetura de Software. 2. Sistemas Embarcados. 3. Sistemas em Tempo Real. 4. Dispositivos Médicos. 5. Análise de Padrões Respiratórios. 6. Interface Homem-Máquina (IHM). I. Maia Júnior, Geraldo Leite. (Orientação). II. Campos, Shirley Lima. (Coorientação). IV. Título.

620 CDD (22.ed.)

IGOR MAURÍCIO DE CAMPOS

ARQUITETURA DE SOFTWARE PARA SISTEMA DE ANÁLISE RESPIRATÓRIA: Respiratory Diagnostic Assistant

Trabalho de Conclusão de Curso apresentado ao Departamento de Engenharia Elétrica da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação.

Aprovado em: 11/04/2025

BANCA EXAMINADORA

_____
Prof. Dr. Geraldo Leite Maia Júnior (Orientador)
Universidade Federal de Pernambuco

_____
Prof. Dr. Márcio Rodrigo Santos de Carvalho (Examinador Interno)
Universidade Federal de Pernambuco

_____
Prof. Dr. Calebe Hermann de Oliveira Lima (Examinador Interno)
Universidade Federal de Pernambuco

Dedico esse trabalho à Professora Dra. Shirley Lima Campos, pela orientação e apoio inestimável durante a pesquisa que originou o RDA. Ao Professor Dr. Geraldo Leite Maia, por seu constante suporte e dedicação. Aos meus sócios, Paulo Roberto Sá e José Emmanuel Matias, pela parceria nesta jornada de desenvolvimento. E à minha família, em especial minha mãe, pelo suporte em toda a trajetória acadêmica.

AGRADECIMENTOS

A realização deste trabalho foi possível graças ao apoio e dedicação de muitas pessoas às quais expresso minha gratidão.

Agradeço primeiramente à Professora Doutora Shirley Campos pela orientação incansável, pelo conhecimento compartilhado e pelo incentivo durante toda a pesquisa que originou a RDA. Ao Professor Doutor Geraldo Leite Maia, pelo suporte técnico, pelas discussões enriquecedoras e pelo comprometimento com o desenvolvimento deste projeto.

Aos meus sócios, Paulo Roberto Sá e José Emmanuel Matias, por embarcarem nesta jornada comigo, pelo trabalho conjunto e pela parceria na construção deste projeto pela Add Health. Sou grato pelo comprometimento de vocês, por todas as vezes em que ficamos até tarde, frequentemente ultrapassando os horários para garantir que atendêssemos aos prazos e mantivéssemos a qualidade do nosso trabalho. Sem essa dedicação incansável, este projeto não teria sido possível.

À minha família, em especial à minha mãe, pelo suporte, paciência e incentivo ao longo da minha trajetória acadêmica. Mesmo com a distância entre Arapiraca e Recife, sua presença e apoio nunca faltaram. Sou igualmente grato a todos os familiares que, sempre que necessário, contribuíram de alguma forma, tornando possível a conclusão da minha graduação aqui. Sem esse suporte, esta jornada teria sido ainda mais desafiadora.

Aos amigos, colegas e professores que, de alguma forma, contribuíram com ideias, sugestões e palavras de encorajamento, meu sincero obrigado. Sou especialmente grata por aqueles que me apoiaram mesmo quando minhas ideias eram fora da caixa, incentivando-me a explorá-las e permitindo que eu desenvolvesse minhas habilidades ao longo do caminho. Esse suporte foi fundamental para minha jornada acadêmica e profissional.

Agradeço ao SEBRAE, cuja iniciativa por meio do edital Catalisa ICT tornou possível a execução deste projeto. O apoio oferecido por meio das bolsas de incentivo à pesquisa foi essencial para o desenvolvimento do produto apresentado neste trabalho, permitindo transformar ideias em realidade e contribuir para a inovação.

Por fim, agradeço a todos que, direta ou indiretamente, participaram desta caminhada e ajudaram a tornar este trabalho uma realidade.

RESUMO

Este trabalho apresenta o desenvolvimento da arquitetura de software para uma nova versão do Respiratory Diagnostic Assistant (RDA), um dispositivo projetado para análise quantitativa e qualitativa do padrão respiratório de pacientes. A versão inicial do RDA, embora funcional, apresentava alto acoplamento, dificultando a extensão e a testabilidade do sistema. Dessa forma, o objetivo deste trabalho foi projetar uma arquitetura de software que seja modular, testável, extensível e manutenível para que novas funcionalidades possam ser implementadas em um novo sistema. Para superar essas limitações, a metodologia utilizada foi analisar de forma aprofundada e aplicar as tecnologias, padrões de projeto e boas práticas de desenvolvimento de software que podem ser utilizadas no projeto, incluindo a Arquitetura Hexagonal e os princípios SOLID, além de desenvolver ferramentas para auxiliar o desenvolvimento de software. Como resultado, foi proporcionando uma infraestrutura robusta para a evolução contínua do RDA, com foco na qualidade de software. Com isso, foi possível criar uma nova versão do sistema, que incorpora concorrência em tempo real no dispositivo embarcado, permitindo o desenvolvimento de uma interface gráfica touch, processamento de sinais, persistência de dados e comunicação externa. Paralelamente, uma nova aplicação desktop foi desenvolvida para oferecer sincronização e armazenamento de dados, além de funcionalidades avançadas para análise de padrões respiratórios, priorizando usabilidade e experiência do usuário. Concluiu-se que este trabalho implementa uma arquitetura robusta para a análise de padrões respiratórios, estabelecendo uma base aplicável a outros produtos e contribuindo para pesquisas e inovações em sistemas embarcados e monitoramento respiratório.

Palavras-chaves:Arquitetura de Software; Sistemas Embarcados; Sistemas em Tempo Real; Dispositivos Médicos; Processamento de Sinais; Análise de Padrões Respiratórios; Interface Homem-Máquina (IHM).

ABSTRACT

This work presents the development of the software architecture for a new version of the Respiratory Diagnostic Assistant (RDA), a device designed for the quantitative and qualitative analysis of patients' respiratory patterns. The initial version of the RDA, although functional, exhibited high coupling, making it difficult to extend and test the system. Therefore, the goal of this work was to design a software architecture that is modular, testable, extensible, and maintainable, so that new functionalities can be implemented in a new system. To overcome these limitations, the methodology used was to analyze in depth and apply technologies, design patterns, and best software development practices that could be used in the project, including Hexagonal Architecture and SOLID principles, as well as developing tools to assist software development. As a result, a robust infrastructure was provided for the continuous evolution of the RDA, with a focus on software quality. Thus, it was possible to create a new version of the system, which incorporates real-time concurrency in the embedded device, enabling the development of a touch graphical interface, signal processing, data persistence, and external communication. At the same time, a new desktop application was developed to provide synchronization and data storage, as well as advanced functionalities for respiratory pattern analysis, prioritizing usability and user experience. In conclusion, this work implements a robust architecture for the analysis of respiratory patterns, establishing a foundation applicable to other products and contributing to research and innovations in embedded systems and respiratory monitoring.

Keywords: Software Architecture; Embedded Systems; Real-Time Systems; Medical Devices; Signal Processing; Respiratory Pattern Analysis; Human-Machine Interface (HMI).

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CI	Capacidade Inspiratória
CI/CD	Continuous Integration/Continuous Delivery/Deployment
CSS	Cascading Style Sheets
CVL	Capacidade Vital Lenta
DAO	Data Access Object
FR	Frequência Respiratória
Go	Golang
HAL	Hardware Abstraction Layer
HTML	HyperText Markup Language
I2C	Inter-Integrated Circuit
IHM	Interface Homem-Máquina
Add Health I.S.	Add Health Inova Simples
JSON	JavaScript Object Notation
LINDEF	Laboratório Multiusuário de Inovação Instrumen- tal e Desempenho Físico-Funcional da UFPE
LVGL	Light and Versatile Embedded Graphics Library
RDA	Respiratory Diagnostic Assistant
RTOS	Real Time Operating System
SEBRAE	Serviço Brasileiro de Apoio às Micro e Pequenas Empresas
SPIFFS	Serial Peripheral Interface Flash File System
SQL	Structured Query Language
TDD	Test Driven Development
TE	Tempo Expiratório

TFT	Thin-Film Transistor
TI	Tempo Inspiratório
Tot	Tempo Total
VM	Volume Minuto
Vt	Volume Corrente

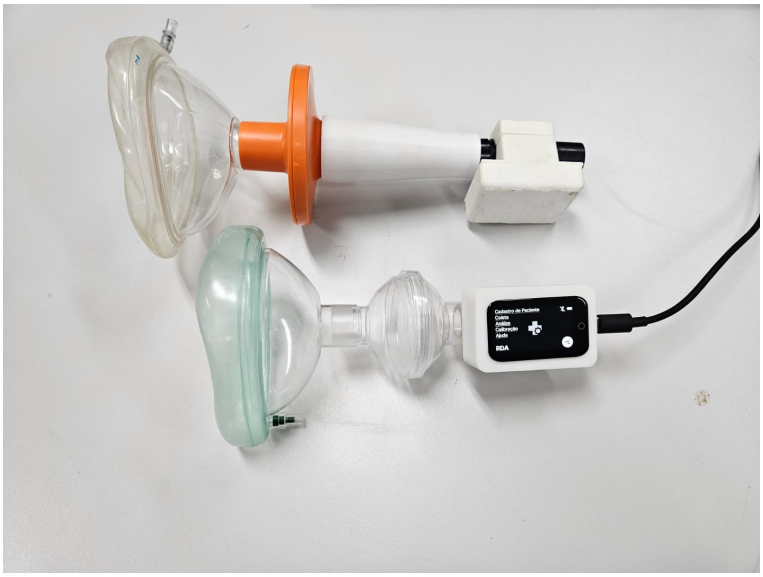
LISTA DE CÓDIGOS

SUMÁRIO

1 INTRODUÇÃO

Respiratory Diagnostic Assistant (RDA) (CAMPOS et al., 2016), trata-se de um instrumento inovador destinado para a avaliação, monitorização e diagnóstico de alterações de padrões da respiração humana. Este instrumento visa fornecer informações qualitativas e quantitativas sobre variáveis do padrão de respiração de forma não invasiva, sem radiação, com portabilidade, de modo a suportar a avaliação de disfunções respiratórias em diferentes populações, como adultos saudáveis, pacientes cardiopatas, pneumopatas e com insuficiência renal crônica. A Figura 1 ilustra o hardware anterior ao projeto no topo da imagem e o hardware novo, na parte inferior, para o qual foi desenvolvido o novo sistema.

Figura 1 – Respiratory Diagnostic Assistant na sua versão anterior e atual



Fonte: Registrado pelo autor (2025).

Além disso, o RDA destaca-se pela sua versatilidade, oferecendo uma contribuição significativa na implementação de abordagens personalizadas e eficazes na avaliação, monitorização, diagnóstico e tratamento diante de alterações de padrões da respiração humana em uma ampla gama de ambientes assistenciais. Pode ser utilizado em níveis ambulatoriais, em enfermarias ou em unidades de terapia intensiva, se adaptando às diferentes necessidades e complexidades de cada contexto.

O equipamento realiza a mensuração de variáveis do padrão respiratório relacionadas a volume e fluxo durante um ou mais ciclos respiratórios, compreendidos por fases inspiratória e expiratória, apresentando-as em termos qualitativos de normalidade e alteração, quantitativos, em termos de valores mínimos, máximos, média, variação e frações em uma escala de tempo e graficamente, em termos de curvas simples no tempo e loops entre as variáveis, com emissão de relatório para auxiliar no diagnóstico do terapeuta.

Em um exame do padrão respiratório em repouso, de duração mínima de um minuto, o RDA mensura a partir da análise de ciclos respiratórios as seguintes variáveis: Frequência Respiratória (FR), Fluxo Inspiratório Médio, Fluxo Expiratório Médio, Tempo Inspiratório (TI), Tempo Expiratório (TE), Relação Tempo de inspiração por Tempo de expiração (I:E), Relação Tempo inspiratório por Tempo total (Ti:Tot), Volume Corrente (Vt), Volume Minuto (VM), Capacidade Inspiratória (CI) e Capacidade Vital Lenta (CVL).

Uma versão inicial do sistema havia sido desenvolvida anteriormente, como resultado de uma parceria entre o Departamento de Fisioterapia e o Departamento de Engenharia Elétrica da Universidade Federal de Pernambuco (UFPE). O autor participou deste projeto por meio de uma iniciação científica realizada no Laboratório Multiusuário de Inovação Instrumental e Desempenho Físico-Funcional (LINDEF/UFPE), contribuindo no refinamento do software desktop responsável pelo controle da coleta de sinais, visualização das curvas respiratórias e realização das análises, denominado RDA Analysis (CAMPOS; CAMPOS; LEITÃO, 2022). Também colaborou no desenvolvimento do RDA Sync (CAMPOS et al., 2022), uma aplicação voltada à análise comparativa de múltiplos exames de um mesmo paciente.

Com a finalização dessa etapa inicial, os integrantes da equipe original saíram do projeto. Após isso, a startup Add Health I.S., da qual o autor é sócio co-fundador, assumiu a continuidade do desenvolvimento por meio do edital Catalisa ICT do SEBRAE, em parceria com o LINDEF, laboratório responsável pela criação da patente e pelas primeiras versões do RDA. O objetivo dessa nova fase foi aprimorar o sistema, transformando-o em um produto mais robusto e adequado para aplicação comercial.

1.1 JUSTIFICATIVA

Os softwares anteriores foram utilizados em ambientes reais para coletas de pesquisas, por exemplo, para o rastreamento de disfunções respiratórias em indivíduos na condição pós-covid-19 (SILVA, 2023). As críticas e sugestões com relação à usabilidade e às funcionalidades deles por profissionais da área de Respiratória foram registradas em formulários de pesquisa, com auxílio de uma orientadora, que possui expertise na área de Respiratória e é uma das autoras da patente do RDA. As informações foram analisadas pelo time de desenvolvimento de hardware e software da Add Health para o projeto de uma nova versão que atenda as sugestões levantadas e seja mais robusta e comercializável. Desse modo, o desenvolvimento do sistema é de grande interesse, pois visa melhorar a usabilidade e a funcionalidade do software com base nos feedbacks recebidos de profissionais da área de saúde.

O sistema anterior apresentava críticas relacionadas à perda de coleta de dados quando a conexão com o computador falhava e à falta de portabilidade, pois o dispositivo precisava de um computador com conectividade Bluetooth para realizar as coletas. A proposta de melhoria foi adicionar uma tela touch ao dispositivo embarcado, permitindo que o usuário

interagisse diretamente com o dispositivo e dispensasse o computador, o que representava uma mudança significativa, pois a versão anterior apenas coletava o fluxo respiratório e enviava os dados via Bluetooth para processamento no desktop.

Outra sugestão foi melhorar o armazenamento de coletas a longo prazo, que antes era feito em arquivos JSON. A nova solução propôs armazenar os exames no próprio dispositivo, com a possibilidade de enviá-los para um banco de dados local no computador, como SQLite, e exportá-los para JSON, quando necessário. Essa abordagem aumentaria a autonomia do dispositivo e permitiria uma análise robusta dos exames, além de facilitar a migração de dados para futuras versões de software.

Outras solicitações incluíam a internacionalização do software, permitindo a troca de idioma para inglês, e a adição de novas variáveis na análise, como o pico de fluxo expiratório, e gráficos de volume respiratório e fluxo x volume respiratório durante a coleta. Para isso, foi necessário garantir que o software fosse extensível, permitindo a modificação de textos, a adição de gráficos e novas variáveis de análise, aspectos que não eram facilmente modificáveis no sistema anterior.

Além disso, o time de desenvolvimento identificou que o software embarcado e desktop anteriores possuíam uma estrutura altamente acoplada, dificultando a realização de testes. No dispositivo embarcado, não havia abstração em relação ao hardware, e todas as rotinas utilizavam diretamente os recursos do microcontrolador ESP-32, o que, embora preciso, dificultava a extensibilidade e a manutenibilidade do código. O RDA Analysis, desenvolvido com os frameworks Electron e React, também possuía alta dependência de recursos externos, tornando-o suscetível a falhas de compatibilidade de versões e tornando a estrutura do código frágil. Isso tornava difícil verificar se uma mudança em uma parte do sistema causaria efeitos colaterais em outras partes.

A arquitetura de software refere-se à estrutura que os componentes de um sistema são organizados e a forma com que eles interagem entre si e com o ambiente externo (MARTIN, 2017). Também envolve decisões de design que afetam o comportamento e a qualidade do sistema, como modularidade, escalabilidade, performance e segurança. Uma boa arquitetura busca garantir que o software seja robusto, manutenível e flexível o suficiente para evoluir com o tempo, visando atender aos requisitos do sistema.

Diante dessas dificuldades, tornou-se necessário redesenhar a arquitetura do sistema, visando maior modularidade, escalabilidade e facilidade de manutenção, para que o software atendesse aos requisitos dos profissionais de saúde e fosse facilmente expansível nas versões futuras.

A implementação do novo sistema considera as limitações técnicas e os recursos disponíveis. O desenvolvimento foi tecnicamente viável, pois o hardware utilizado possui alto nível de processamento, suporte integrado a sistema operacional de tempo real e outros recursos fundamentais para a execução do projeto, o que possibilitou melhorar a interatividade do dispositivo, mantendo a precisão das coletas respiratórias. Além disso, a

utilização de um banco de dados local, como o SQLite, também foi viável em termos de recursos de hardware, através de um armazenamento em memória flash, permitindo um armazenamento mais robusto dos dados.

Outrossim, o desenvolvimento do sistema priorizou a segurança e a privacidade dos dados dos pacientes. Dessa forma, o os exames são armazenados localmente, garantindo total controle dos clientes com seus dados sensíveis e assegurando que as informações respiratórias coletadas sejam tratadas de forma confidencial e em conformidade com as regulamentações vigentes. As atividades realizadas não necessitaram de comitê de ética por serem específicas ao desenvolvimento de software.

Portanto, o trabalho é relevante tanto para a área da saúde quanto para o desenvolvimento de tecnologias embarcadas, visto que a metodologia utilizada pode ser aplicada a outros sistemas, visando a qualidade de software na forma de escalabilidade, modularidade, testabilidade e robustez.

1.2 OBJETIVOS

1.2.1 Geral

Este trabalho tem como objetivo desenvolver e implementar uma arquitetura de software para um sistema embarcado e uma aplicação desktop, destinados à nova versão do RDA. A solução será projetada de forma modular, testável, extensível e de fácil manutenção, garantindo maior confiabilidade e eficiência na avaliação e monitoramento dos padrões respiratórios.

1.2.2 Específicos

Projetar e implementar um software embarcado capaz de aquisitar e processar, em tempo real, os dados de fluxo e volume respiratório.

Desenvolver uma interface gráfica embarcada que permita o controle do processo de coleta e a visualização dos gráficos relacionados tanto à coleta atual quanto às anteriores.

Criar uma aplicação desktop que priorize a usabilidade, permitindo a realização de análises detalhadas dos dados respiratórios.

Construir arquitetura adotando padrões de projeto e baseada na Arquitetura Hexagonal.

Implementar um módulo de comunicação entre o software embarcado e a aplicação desktop, assegurando a sincronização dos dados entre os dois sistemas.

Elaborar e integrar módulos específicos para o gerenciamento de pacientes, exames e análises, permitindo que os usuários registrem e organizem as informações de forma estruturada.

1.3 ORGANIZAÇÃO TEXTUAL

Este trabalho está estruturado em cinco capítulos, além das referências, conforme descrito a seguir.

O Capítulo 1 (Introdução) apresenta uma visão geral do projeto, incluindo a justificativa, os objetivos gerais e específicos e a estrutura do trabalho.

O Capítulo 2 (Revisão da Literatura) aborda os conceitos teóricos fundamentais para o desenvolvimento do sistema. São discutidos princípios de arquitetura de software, como Inversão de Dependências e Arquitetura Hexagonal, além de padrões de projeto utilizados, incluindo Mediator, Adapter, Singleton e Repository. Também são explorados tópicos relacionados a testes de software, como Pirâmide de Testes, Test Doubles e Desenvolvimento Orientado a Testes (TDD). Por fim, é abordada a avaliação de padrões respiratórios, essencial para a interpretação dos dados do sistema.

O Capítulo 3 (Metodologia) descreve os métodos e técnicas aplicados no desenvolvimento da arquitetura e implementação do sistema. São detalhados o plano de trabalho, a criação da camada de abstração do hardware (Hardware Abstraction Layer - HAL), os processos de calibração e persistência de dados, além das estratégias para comunicação entre o dispositivo embarcado e o software desktop. O capítulo também cobre o design e a implementação das interfaces gráficas, tanto para a aplicação desktop quanto para o sistema embarcado, destacando aspectos como prototipação, componentização, gerenciamento de eventos e arquitetura orientada a eventos.

O Capítulo 4 (Resultados) apresenta a avaliação do sistema desenvolvido, analisando aspectos como testabilidade, modularidade, manutenibilidade e eficiência da solução. São apontados os desafios enfrentados ao longo do projeto e as limitações observadas, além de descrever a interface gráfica embarcada e desktop.

O Capítulo 5 (Conclusão e Trabalhos Futuros) resume os principais achados do trabalho e propõe direções para melhorias e futuras evoluções do sistema.

Por fim, são apresentadas as Referências, contendo os materiais utilizados como base para o desenvolvimento do projeto.

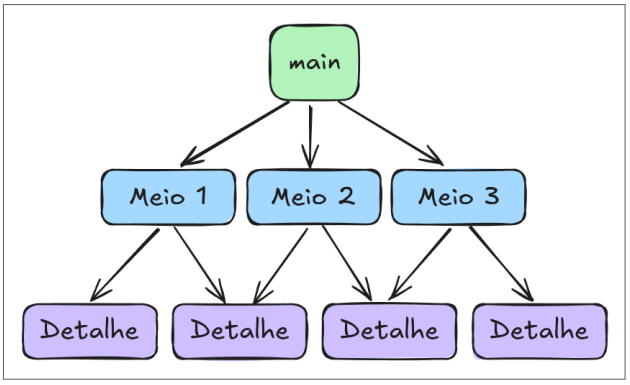
2 FUNDAMENTAÇÃO TEÓRICA

2.1 INVERSÃO DE DEPENDÊNCIAS

O princípio da inversão de dependências, criado por Robert C. Martin como um dos princípios SOLID em seu livro Clean Architecture (MARTIN, 2017), propõe que código deva depender apenas de abstrações, não de implementações concretas. Dessa forma, caso uma unidade necessite de um recurso que não faz parte das suas obrigrações, seja um algoritmo ou acesso a um serviço externo, por exemplo, deve ser criada uma interface que fornece um contrato que qualquer implementação desse recurso deve seguir. Dessa forma, a unidade depende apenas do contrato e a implementação pode ser alterada livremente.

Arquiteturas procedurais seguem a estrutura presente na Figura 2, na qual ocorre dependência direta da implementação dos componentes. Essa estrutura indica que os detalhes são livres para serem alterados e não sofrem consequências, contudo, suas mudanças são propagadas para as camadas superiores. Isso torna as camadas superiores frágeis, de modo que são propensas a serem alteradas por detalhes de implementação.

Figura 2 – Estrutura de uma Arquitetura Procedural

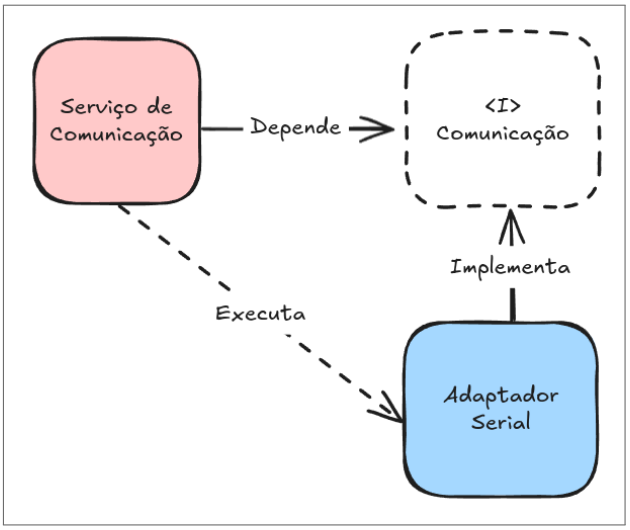


Fonte: Elaborado pelo autor (2025)

A estrutura presente na Figura 3 demonstra como ocorre a inversão de dependências. Desse modo, o código depende de um contrato e o torna a parte mais sensível do fluxo e remove a dependência direta da implementação concreta. Dessa forma, mudanças de requisitos e de regras de negócio podem ser aplicadas de forma isolada na implementação concreta, sem afetar a estrutura do programa.

A implementação do princípio da inversão de dependências pode ser realizada através do padrão de injeção de dependências. Existem diferentes implementações possíveis, que podem variar de acordo com linguagem e ferramentas utilizadas. O Código 1 ilustra a injeção de dependências realizada pelo construtor de uma classe na linguagem C++. Nesse exemplo, a classe de serviço depende da classe abstrata que contém o contrato da dependência. Há duas implementações para esse contrato, dependência A e B. Na função

Figura 3 – Princípio da Inversão de Dependências



Fonte: Elaborado pelo autor (2025)

principal, a implementação de dependência A foi injetada no objeto de serviço criado através do construtor, logo ao executar, a mensagem enviada para a stream de saída do programa é da dependência A.

Código Fonte 1 – Exemplo de Inversão de Dependências em C++

```
#include <iostream>

class Dependencia {
public:
    virtual void FazerTrabalho() = 0;
};

class DependenciaA : public Dependencia {
public:
    void FazerTrabalho() override {
        printf("DependenciaA fazendo trabalho\n");
    }
};

class DependenciaB : public Dependencia {
public:
    void FazerTrabalho() override {
        printf("DependenciaB fazendo trabalho\n");
    }
};

class Servico {
public:
    Servico(Dependencia& dep) : dependencia(dep) {}

    void Executar() {
        dependencia.FazerTrabalho();
    }

```

```
    private:
31      Dependencia& dependencia;
    };
33
    int main() {
35      DependenciaA dep;
        Servico servico(dep);
37      servico.Executar();
    }
```

Fonte: Elaborado pelo autor (2025)

2.2 ARQUITETURA HEXAGONAL

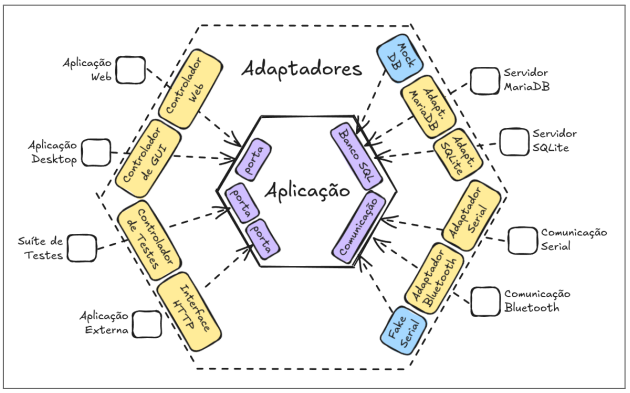
A Arquitetura Hexagonal (COCKBURN, 2005), proposta por Alistair Cockburn, é um modelo de design de software que visa criar sistemas mais flexíveis, desacoplados e testáveis ao separar a lógica central da aplicação das interfaces externas, como bancos de dados, interfaces gráficas, APIs, etc. Essa abordagem é baseada no conceito de portas e adaptadores, onde as portas definem interfaces de entrada e saída para a aplicação e os adaptadores as implementam, permitindo a comunicação entre o núcleo da aplicação e o mundo externo.

A Arquitetura Hexagonal promove a inversão de dependências, de forma que o núcleo da aplicação seja independente dos detalhes de infraestrutura. Isso facilita a substituição de componentes, como trocar um banco de dados ou mudar a interface de usuário, sem impactar a lógica da aplicação. Além disso, facilita os testes da aplicação por isolar as dependências e ser simples de substituir as dependências por implementações falsas.

Outro conceito importante é que as tecnologias que se integram à aplicação principal são divididos em sistemas externos que dirigem e que são dirigidos pela aplicação, de forma que a aplicação só contenha o código específico do sistema. Exemplos de sistemas que dirigem a aplicação são sistemas web, interfaces gráficas, suíte de testes, filas, etc. Exemplos de sistemas dirigidos pela aplicação são banco de dados, sistema de pagamento, etc. Um diagrama pode ser observado na Figura 4.

Na prática, a aplicação é estruturada em duas camadas principais: o núcleo da aplicação, que contém as regras de negócio e as portas, que são interfaces pelas quais o sistema espera se comunicar com o ambiente externo; e os adaptadores, que implementam essas interfaces utilizando tecnologias específicas, como frameworks web, bancos de dados e APIs. Além disso, o núcleo da aplicação não está restrito a uma estrutura específica, então pode ter sub-camadas e ser implementado segundo outros padrões arquiteturais, como Transaction Script ou Domain Model, disponíveis no livro Design Patterns for Enterprise Application Architecture (FOWLER, 2002).

Figura 4 – Arquitetura Hexagonal

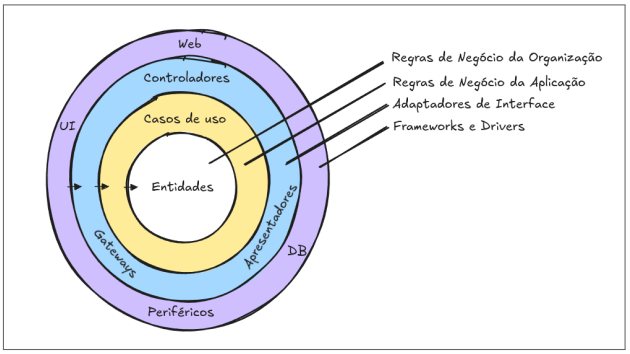


Fonte: Elaborado pelo autor (2025)

2.3 ARQUITETURA LIMPA

Arquitetura Limpa (MARTIN, 2017), proposta por Robert C. Martin, é um modelo de design de software que busca manter a lógica de negócio independente de frameworks, bancos de dados, interfaces gráficas e outras tecnologias externas, além de segmentar de forma lógica a aplicação em camadas com propósitos distintos, como pode ser visto na Figura 5.

Figura 5 – Arquitetura Limpa



Fonte: Elaborado pelo autor (2025)

A estrutura da Arquitetura Limpa é composta por diferentes camadas concêntricas. No centro estão as entidades, que representam os conceitos fundamentais do domínio e são independentes de regras específicas da aplicação. Ao redor da camada de entidades, os casos de uso são responsáveis pelas funcionalidades da aplicação através da orquestração das interações entre as entidades e o mundo externo. Essas duas camadas representam o núcleo da aplicação, pois contém todas as regras de negócio e lógica inerente à aplicação. As camadas externas ao núcleo são relativas a detalhes de implementação da aplicação.

Envolvendo o núcleo, está a camada de adaptadores, que fazem a ponte entre o núcleo da aplicação e os sistemas externos, como interfaces gráficas, bancos de dados e serviços de terceiros. Os adaptadores implementam as interfaces esperadas pelas camadas internas

utilizando os recursos da camada mais externa, de infraestrutura e frameworks, onde estão localizadas as ferramentas e bibliotecas específicas que a aplicação utiliza.

Um dos princípios fundamentais da Arquitetura Limpa é a dependência unidirecional, ou seja, as camadas externas podem conhecer as internas, mas o contrário não ocorre, seguindo o princípio da inversão de dependências. Isso permite testar de forma simplificada e evoluir a lógica central da aplicação sem estar preso a decisões tecnológicas específicas, além de mitigar a ocorrência de referências circulares. Com essa abordagem, a manutenção do software se torna mais previsível e sustentável, facilitando alterações e expansões sem modificar a estrutura do projeto.

2.4 DESIGN PATTERNS

Design Patterns (Padrões de Projeto) são estruturas de código que solucionam problemas recorrentes no desenvolvimento de software (GAMMA et al., 1994). Esses padrões encapsulam boas práticas e auxiliam na organização do código, promovendo modularidade, reutilização e facilidade de manutenção. A seguir, são descritos alguns padrões relevantes utilizados neste trabalho.

2.4.1 Mediator

O padrão Mediator (GAMMA et al., 1994) tem como objetivo reduzir o acoplamento entre objetos, delegando a comunicação entre eles para um componente central, chamado de mediador. Ao invés dos objetos interagirem diretamente, podem se comunicar por meio do mediador, que gerencia as interações. Isso facilita a manutenção do código, pois mudanças nos componentes não afetam diretamente outros elementos do sistema.

Uma interpretação possível do padrão é um modelo publicador-assinante. Dessa forma, objetos podem se inscrever para eventos, fornecendo funções de callback para quando o evento ocorrer, executar a função enviada, com o evento como argumento. Por outro lado, outros objetos podem publicar esses eventos para serem repassados para os assinantes. Uma implementação do padrão em C++ pode ser vista no Código 2.

Código Fonte 2 – Exemplo do Design Pattern Mediator em C++

```
#include <any>
#include <functional>
#include <string>
#include <unordered_map>
#include <vector>

using Callback = std::function<void(std::any)>;

class Mediator {
   private:
    std::unordered_map<std::string, std::vector<Callback>> callbacks;

   public:
```

```
14      void subscribe(const std::string& event, Callback callback) {
            callbacks[event].push_back(callback);
16      }

18      void notify(const std::string& event, std::any args) {
            if (callbacks.find(event) != callbacks.end()) {
20              for (auto& callback : callbacks[event]) {
                    callback(args);
22              }
            }
24      }
};
```

Fonte: Elaborado pelo autor (2025)

Como um publicador não conhece os assinantes, módulos distintos podem consumir os eventos sem impactar no código de publicação, que pode não ter relação direta. Um exemplo é um sistema de gamificação de uma plataforma, que escuta os eventos em paralelo com a execução principal, validando conquistas do usuário. Esse tipo de sistema geraria problemas ao estar acoplado com outros componentes do projeto.

2.4.2 Adapter

O padrão Adapter (GAMMA et al., 1994), base da Arquitetura Hexagonal, é utilizado como um tradutor para permitir a utilização de uma dependência em uma interface incompatível. Por exemplo, um caso de uso de checkout possui como dependência uma API de pagamentos. Para aderir à arquitetura hexagonal, o caso de uso depende de um contrato de API de pagamentos a ser seguido. Dessa forma, é criado um adaptador, que adere a esse contrato usando uma API de pagamentos específica e pode ser injetado como dependência. Uma implementação do padrão em C++ pode ser vista no Código 3.

Código Fonte 3 – Exemplo do Design Pattern Adapter em C++

```
1  #include <iostream>

3  class Dependencia {
   public:
5      virtual void FazerTrabalho() = 0;
   };
7
   class DependenciaExterna {
9  public:
       void TrabalhoEspecifico() {
11         printf("ClasseExistente fazendo trabalho espec fico\n");
       }
13 };

15 class Adaptador : public Dependencia {
   public:
17     Adaptador(DependenciaExterna& classeExistente) : classeExistente(
           classeExistente) {}
```

```
19      void FazerTrabalho() override {
            classeExistente.TrabalhoEspecifico();
21      }

23  private:
        DependenciaExterna& classeExistente;
25  };

27  class Servico {
    public:
29      Servico(Dependencia& dep) : dependencia(dep) {}

31      void Executar() {
            dependencia.FazerTrabalho();
33      }

35  private:
        Dependencia& dependencia;
37  };

39  int main() {
        DependenciaExterna classeExistente;
41      Adaptador adaptador(classeExistente);
        Servico servico(adaptador);
43      servico.Executar();

45      return 0;
    }
```

Fonte: Elaborado pelo autor (2025)

2.4.3 Singleton

O padrão Singleton (GAMMA et al., 1994) permite que uma classe tenha apenas uma única instância e fornece um ponto global de acesso a essa instância. Com isso, é possível aplicar o padrão para reutilizar recursos comuns em diferentes pontos do sistema, como acesso ao banco de dados, sem a necessidade de transportar instâncias ao longo da estrutura de dependências. Portanto, reduz a quantidade de parâmetros de unidades que utilizem a dependência e flexibiliza a estrutura da aplicação.

Um exemplo de implementação em C++ pode ser visto no Código 4. Na qual, é utilizada a diretiva "static"para criar a função "getInstance", que fornece a instância única do Singleton, que é criada estaticamente dentro da função, logo não é alterada. Além disso, é deletado o operador de atribuição e a chamada de construtor externa, de modo que não seja possível criar ou alterar a instância do Singleton. A classe possui métodos como setProperty e getProperty, que são operações específicas desse exemplo e poderiam ser alterados de acordo com a aplicação.

Código Fonte 4 – Exemplo do Design Pattern Singleton em C++

```
    #include <iostream>
2
```

```
class Singleton {
private:
    int property;
    Singleton() {
        std::cout << "Singleton constructor called" << std::endl;
    }
public:

    static Singleton& getInstance() {
        static Singleton instance;
        return instance;
    }

    void setProperty(int value) {
        property = value;
    }

    int getProperty(){
        return property;
    }

    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
};

int main() {
    Singleton& singleton1 = Singleton::getInstance();
    singleton1.setProperty(10);
    std::cout << singleton1.getProperty() << std::endl;
    return 0;
}
```

Fonte: Elaborado pelo autor (2025)

2.4.4 Repository

O padrão Repository (FOWLER, 2002) atua como uma camada intermediária entre o domínio da aplicação e a persistência de dados, fornecendo uma abstração para operações de banco de dados. De forma simplificada, fornece uma interface pela qual a aplicação pode utilizar para realizar operações de persistência utilizando objetos de domínio, sem depender diretamente da tecnologia utilizada. Para sistemas baseados em Transaction Script (FOWLER, 2002), por exemplo, que não possuem um domínio, o conceito de abstração para o banco de dados por meio de uma classe é chamado de DAO (Data Access Object).

Um exemplo de implementação do padrão Repository está presente no Código 5, na forma de um repositório de usuários. No código, uma estrutura é criada para representar a entidade de usuário, junto com uma interface para o repositório, que possui métodos para criar e recuperar usuário pelo identificador. Daí, é criada uma implementação concreta do repositório, que implementa os métodos para um armazenamento em memória. Poderiam

existir implementações com adaptadores de bancos de dados relacionais ou outro tipo de tecnologia, desde que o contrato seja cumprido.

Código Fonte 5 – Exemplo do Design Pattern Repository em Go

```go
package main

import (
    "fmt"
)

type User struct {
    ID   int
    Name string
}

type UserRepository interface {
    GetByID(id int) (*User, error)
    Create(user *User) error
}

type InMemoryUserRepository struct {
    users map[int]*User
}

func NewInMemoryUserRepository() *InMemoryUserRepository {
    return &InMemoryUserRepository{
        users: make(map[int]*User),
    }
}

func (repo *InMemoryUserRepository) GetByID(id int) (*User, error) {
    user, exists := repo.users[id]
    if !exists {
        return nil, fmt.Errorf("usuario nao encontrado")
    }
    return user, nil
}

func (repo *InMemoryUserRepository) Create(user *User) error {
    repo.users[user.ID] = user
    return nil
}

func main() {
    repo := NewInMemoryUserRepository()
    user := &User{ID: 1, Name: "Igor Mauricio"}
    err := repo.Create(user)
    if err != nil {
        fmt.Println("Erro salvando usuario:", err)
        return
    }
    retrievedUser, err := repo.GetByID(1)
    if err != nil {
        fmt.Println("Erro recuperando usuario:", err)
        return
    }
```

```
53      fmt.Printf("Usuario recuperado: ID=%d, Nome=%s\n", retrievedUser.ID,
            retrievedUser.Name)
    }
```

Fonte: Elaborado pelo autor (2025)

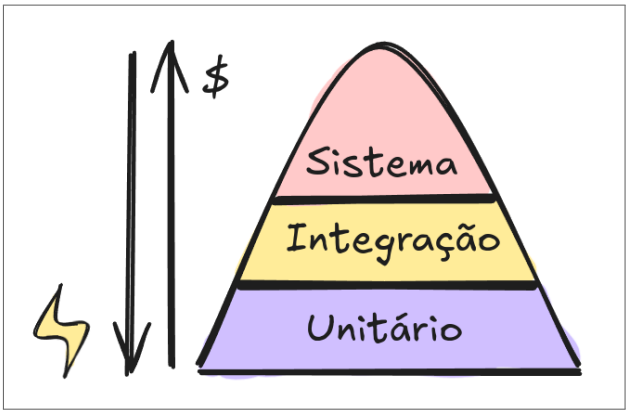
2.5 TESTES DE SOFTWARE

Os testes de software desempenham um papel fundamental na garantia da qualidade e confiabilidade de um sistema. Com eles, é possível identificar falhas, validar o comportamento esperado, verificar se o sistema está atingindo os requisitos de performance, entre outros. Várias categorias de testes podem ser automatizados utilizando frameworks para testes, como o Google Test (Google Test Authors, 2025).

2.5.1 Pirâmide de Testes

Existem diferentes níveis para testes, como testes de unidade, de integração e de sistema, ilustrado na Figura 6. Testes de unidade são relativos a uma unidade de código, que não possui relação com outras partes do projeto, como algoritmos e classes isoladas. Testes de integração são testes que verificam a interação entre diferentes partes do código, como em uma classe de serviço que integra com repositórios, gateways e outras unidades, além de testes de performance. Já Testes de sistema, ou fim a fim, envolvem toda a aplicação, do ponto de vista de quem irá utilizar (COHN, 2010). Por exemplo, para uma aplicação web, o teste fim a fim é feito na interface gráfica, seja manualmente ou de forma automatizada com uma ferramenta como Selenium (Selenium Project, 2025).

Figura 6 – Pirâmide de testes



Fonte: Elaborado pelo autor (2025)

Como testes de unidade não tem dependências, são rápidos para serem executados, então podem ser feitos com maior volume durante o desenvolvimento. Testes de integração podem ser mais lentos por depender da latência de outro sistema, como banco de dados. Testes fim a fim são os mais custosos e lentos para execução, pois utilizam todos os

recursos do projeto (COHN, 2010). Além disso, podem envolver interface gráfica, que tem maior grau de complexidade, fragilidade pela constante alteração dos elementos gráficos e pode ser menos custoso realizar testes manuais ou mesmo inviável de realizar automação, como na maioria das interfaces gráficas embarcadas.

2.5.2 Test Doubles

Existem várias técnicas para conduzir testes em um sistema. Nesse contexto, é comum realizar testes de integração que utilizam banco de dados, APIs e outras dependências externas. Realizar testes com as implementações concretas dos elementos de produção põe em risco o sistema em produção, além de ter atrasos na execução por realizar a comunicação com outros sistemas e apresentarem fragilidade por depender de outros sistemas. Para isso, existem os Test Doubles, que são técnicas utilizadas para substituir dependências reais durante a execução dos testes, permitindo que o código seja validado sem a necessidade de interagir com componentes externos (MESZAROS, 2007).

Dentre os principais Test Doubles, está o Fake, que é uma implementação falsa da dependência, como um repositório que utiliza um banco de dados em memória. Além disso, há o Stub, que implementa o contrato da dependência retornando valores constantes. Pode ser útil, por exemplo, para retornar sempre a mesma data em um teste que envolve a data de execução da funcionalidade. Outro Test Double é o Spy, que permite a verificação de chamadas de métodos e de parâmetros utilizados durante a execução do programa (MESZAROS, 2007). Pode ser utilizado para assegurar que um serviço está acessando o banco de dados, por exemplo, e enviando parâmetros específicos para a dependência.

Por fim, há o Mock, que é a união de um stub com spy, logo possui características de prover respostas padrões para métodos, além de verificar se foram executados na quantidade de vezes certa e com os parâmetros corretos (MESZAROS, 2007). Dessa forma, o Mock é o Test Double mais robusto.

Código Fonte 6 – Trecho de testes do serviço de calibração do RDA com o framework Google Test

```
#include <test/utils.h>
#include <Registry.h>
#include <db/repo/repoMocks.h>
#include <calibration/calibration.h>

namespace {

class CalibrationServiceTest : public ::testing::Test {
protected:
    repo::MockRepository repository;
    Mediator mediator;
    calibration::Service calibrationService = calibration::Service(repository);

    void SetUp() override {}
    void TearDown() override {}
```

```
};

TEST_F(CalibrationServiceTest, ShouldGetCalibration) {
    EXPECT_CALL(repository, GetCalibration())
        .Times(1)
        .WillOnce([this]() {
            return repo::GetCalibrationParams{1, 42, 0.5};
        });
    auto [calibration, err] = calibrationService.get();
    EXPECT_EQ(err, nil);
    EXPECT_EQ(calibration.offset, 0.5);
    EXPECT_EQ(calibration.amplitudeGain, 42);
}

TEST_F(CalibrationServiceTest, ShouldErrorWhenNoCalibrationFound) {
    EXPECT_CALL(repository, GetCalibration())
        .Times(1)
        .WillOnce([this]() {
            repository.err = 1;
            return repo::GetCalibrationParams{};
        });
    auto [calibration, err] = calibrationService.get();
    EXPECT_EQ(err, Errors::NotFound);
}

TEST_F(CalibrationServiceTest, ShouldResetCalibrationIfAlreadyExists) {
    EXPECT_CALL(repository, GetCalibration())
        .Times(1)
        .WillOnce([this]() {
            return repo::GetCalibrationParams{42, -2};
        });
    EXPECT_CALL(repository, UpdateCalibration(1, 0, 1))
        .Times(1);
    calibrationService.resetCalibration();
}

TEST_F(CalibrationServiceTest, ShouldCalibrateAmplitude) {
    double referenceVolume = 10;
    std::vector<double> amplitudeCalibration = {0, -1, -2, -1, 1, 3, 2, 1, 0};
    EXPECT_CALL(repository, GetCalibration())
        .Times(1)
        .WillOnce([this]() {
            return repo::GetCalibrationParams{1, 1, 0};
        });
    EXPECT_CALL(repository, UpdateCalibration(2, 0, 1))
        .Times(1);
    Error err = calibrationService.calibrateAmplitude(referenceVolume,
        amplitudeCalibration);
    EXPECT_EQ(err, nil);
}
...

}
```

Fonte: Elaborado pelo autor (2025)

2.5.3 Desenvolvimento Orientado a Testes

O TDD, ou Desenvolvimento Orientado a Testes (BECK, 2003), é uma abordagem onde os testes são escritos antes da implementação do código. O método consiste em um ciclo de três etapas: escrever um teste falho, implementar o código necessário para que o teste passe e, por fim, refatorar o código para melhorar sua estrutura sem alterar seu comportamento. Dessa forma, essa abordagem permite que o sistema tenha seu design projetado na etapa de testes, iniciando com testes de integração para as funcionalidades a serem projetadas, garantindo que os elementos sejam testáveis e promovendo modularidade. Além disso, o TDD reduz a probabilidade de regressões, pois cada alteração no código é validada automaticamente pelos testes existentes.

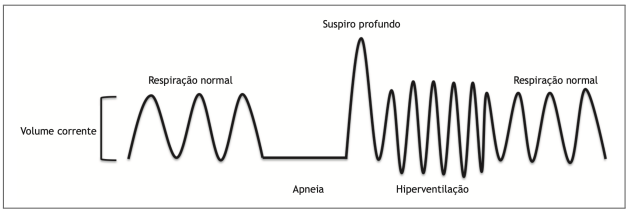
2.6 AVALIAÇÃO DE PADRÕES RESPIRATÓRIOS

A avaliação dos padrões respiratórios desempenha um papel essencial no diagnóstico e monitoramento de distúrbios respiratórios. A curva de fluxo e o volume respiratório podem ser analisados para determinar se a respiração de um indivíduo ocorre dentro da normalidade ou apresenta alterações que possam indicar doenças ou disfunções pulmonares. Para isso, diversas tecnologias foram desenvolvidas com o objetivo de capturar e processar sinais respiratórios de forma precisa e confiável (WEST, 2012).

Em indivíduos saudáveis, a respiração segue um padrão rítmico e eficiente, garantindo a oxigenação adequada do organismo. O ciclo respiratório normal é composto por uma fase inspiratória e uma fase expiratória, ocorrendo de maneira involuntária sob o controle do sistema nervoso central. A frequência respiratória varia conforme a idade e o estado fisiológico do indivíduo, com valores de normalidade tabelados (GANONG, 2019). Além disso, o volume de ar inspirado e expirado em cada ciclo respiratório, conhecido como volume corrente, gira em torno de 500 mL em um adulto médio. Outro parâmetro importante é a relação inspiração/expiração (I:E), onde a expiração dura aproximadamente o dobro do tempo da inspiração (MEHTA; DHOORIA, 2019).

Alterações nesses parâmetros podem indicar a presença de distúrbios respiratórios. Padrões anormais de respiração incluem a taquipneia, caracterizada pelo aumento anormal da frequência respiratória acima de 20 incursões por minuto, geralmente associada a febre, ansiedade ou doenças pulmonares; a bradipneia, que consiste na redução da frequência respiratória para menos de 12 incursões por minuto e pode estar relacionada à depressão do sistema nervoso central ou a distúrbios metabólicos; e a apneia, interrupção temporária da respiração, frequentemente observada em casos de obstruções das vias aéreas ou falhas no controle neural da respiração, como ocorre na apneia do sono (BARRETT et al., 2020). A diferença na curva respiratória entre padrões respiratórios normais e anormais pode ser observada na Figura 7.

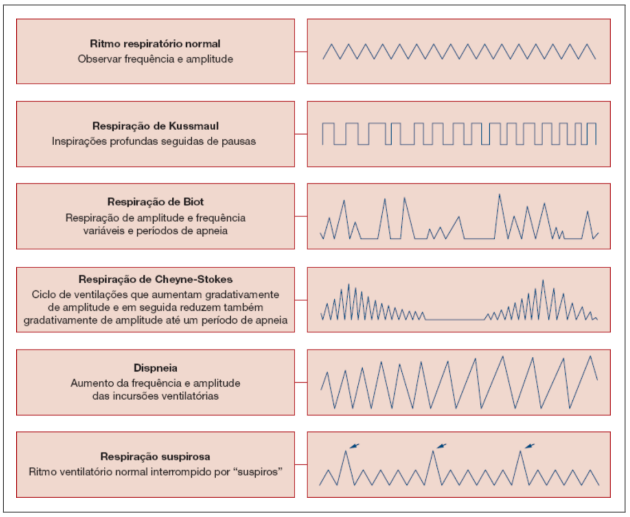
Figura 7 – Padrões respiratórios normais e anormais



Fonte: (VIDOTTO; OUTROS, 2019)

Além das alterações mais comuns, como taquipneia, bradipneia e apneia, existem outros padrões respiratórios anormais que indicam condições clínicas graves. Entre eles estão a respiração de Cheyne-Stokes, que apresenta fases de hiperventilação seguidas por apneia, comum em pacientes com insuficiência cardíaca; a respiração de Biot, que tem respirações rápidas e profundas seguidas de pausas irregulares, geralmente associada a lesões neurológicas; a respiração de Kussmaul, que é profunda e rápida, característica de pessoas com acidose metabólica, como na cetoacidose diabética; e a respiração suspirosa, marcada por suspiros frequentes, muitas vezes ligados ao estresse ou ansiedade (LAGE; SILVA; CAMPOS, 2020). Esses padrões anormais podem ser observados na Figura 8. A identificação precoce desses padrões é importante para que os profissionais de saúde possam agir rapidamente e oferecer o tratamento adequado. Por isso, ferramentas que ajudam a identificar esses sinais com precisão são fundamentais para melhorar o cuidado com os pacientes.

Figura 8 – Representação esquemática de ritmos respiratórios e tipos de dispneia



Fonte: (PORTO, 2019)

Para monitoramento e análise dos padrões respiratórios, diversas tecnologias foram desenvolvidas, cada uma com aplicações específicas. A espirometria é uma das principais ferramentas para avaliação da função pulmonar, medindo o volume e a velocidade do fluxo de ar expirado, sendo amplamente utilizada no diagnóstico de doenças como asma e doença pulmonar obstrutiva crônica (DPOC) (HANKINSON et al., 1999). A pletismografia

de corpo inteiro permite a medição das mudanças no volume torácico, possibilitando a avaliação da resistência das vias aéreas e da capacidade pulmonar total. Outra técnica relevante é a capnografia, que monitora a concentração de dióxido de carbono no ar exalado, sendo fundamental no acompanhamento ventilatório de pacientes sob anestesia ou suporte respiratório (GRAVENSTEIN; COLLEAGUES, 2004).

O sistema desenvolvido neste trabalho se insere nesse contexto, utilizando um sensor de fluxo respiratório embarcado para a aquisição e o processamento de sinais respiratórios. Se diferencia das soluções apresentadas por ser um dispositivo de baixo custo, portátil, não intrusivo e voltado para a análise do padrão respiratório.

3 METODOLOGIA

Para a execução deste trabalho, a metodologia seguiu uma abordagem iterativa, fundamentada na análise de tecnologias, padrões de projeto e boas práticas de desenvolvimento de software. A Tabela 1 descreve as etapas principais que foram realizadas no projeto.

Tabela 1 – Atividades do Projeto

Atividade	Descrição
Revisão bibliográfica sobre arquitetura de software e análise de sistemas respiratórios.	Estudo de modelos arquiteturais e padrões de projeto aplicáveis ao sistema.
Definição dos requisitos e especificações do sistema.	Estabelecimento das funcionalidades essenciais e estrutura geral do software.
Design da arquitetura.	Modelagem da organização modular e das interações entre os componentes, incluindo Arquitetura Hexagonal e Arquitetura Limpa.
Seleção de tecnologias e configuração de ambiente.	Definição de ferramentas, bibliotecas e estruturação do ambiente de desenvolvimento, considerando os requisitos para persistência de dados e comunicação entre sistemas.
Implementação da camada de abstração de hardware (HAL).	Desenvolvimento da interface de abstração entre o hardware e a aplicação, garantindo a flexibilidade e testabilidade do sistema.
Desenvolvimento do sistema de calibração.	Implementação do sistema de calibração para garantir a precisão e confiabilidade das medições respiratórias.
Implementação de persistência de dados.	Utilização de banco de dados e ferramentas de migração, tanto para o software desktop quanto para o embarcado.
Desenvolvimento da interface gráfica.	Prototipação e implementação da interface gráfica para o software desktop e dispositivo embarcado.
Implementação de comunicação entre dispositivo embarcado e software desktop.	Definição de um protocolo de comunicação para troca de dados entre os dois sistemas, com foco em eficiência e clareza na troca de informações.
Implementação de multitasking e RTOS no sistema embarcado.	Desenvolvimento de funcionalidades que permitam a execução de múltiplas tarefas de forma eficiente no dispositivo embarcado.

Fonte: Elaborada pelo autor (2025)

O processo teve início com o levantamento de padrões de projeto e modelos arquiteturais aplicáveis no projeto. Isso foi realizado através da análise de artigos e bibliografias da área de arquitetura de software, como Patterns of Enterprise Application Architecture,

Design Patterns: Elements of Reusable Object-Oriented Software e Design Patterns for Embedded Systems in C. Esses materiais forneceram uma base para a construção de um sistema modular, extensível e com manutenção facilitada.

A arquitetura do sistema foi projetada com base em modelos arquiteturais como a Arquitetura Hexagonal e a Arquitetura Limpa, garantindo a separação de responsabilidades entre os componentes e a inversão de dependências, favorecendo a testabilidade e a flexibilidade do código. Além disso, foram aplicados os princípios SOLID (MARTIN, 2017) para garantir que o sistema fosse coeso e com baixo acoplamento, facilitando futuras manutenções e expansões.

A implementação seguiu a metodologia de Test Driven Development (TDD), que permitiu a criação de módulos testáveis e realizados de forma isolada. A automação de builds e testes foi viabilizada por meio de um ambiente de integração contínua, garantindo a consistência e a qualidade do código durante o processo de desenvolvimento. Para manter a independência dos testes em relação ao hardware, foram adotadas técnicas de inversão de dependências e a utilização de flags de compilação, permitindo que os testes fossem executados em um ambiente Linux, sem a necessidade do hardware físico.

A criação da interface gráfica passou por um processo de prototipação utilizando o Figma, permitindo explorar diferentes alternativas de design antes da implementação efetiva. Para contornar as limitações das bibliotecas gráficas embarcadas, foi desenvolvido um framework para facilitar a componentização e o gerenciamento de eventos.

Ao longo do desenvolvimento, foram criadas ferramentas que otimizaram o processo de testes e integração, como um gerador automatizado de Mocks para substituir dependências em testes, e um transpilador que converte código SQL em C++ compatível com o SQLite para o ambiente do ESP32. Além disso, a comunicação entre o sistema embarcado e a aplicação desktop foi estruturada por meio de um protocolo baseado em comandos definidos, garantindo eficiência e clareza na troca de informações entre os dois sistemas.

3.1 CAMADA DE ABSTRAÇÃO DO HARDWARE (HAL)

Para construir o software embarcado, é necessário acessar os periféricos do microcontrolador, como interface I2C, comunicação serial, entradas e saídas digitais e analógicas, timers e acesso a memória flash. Contudo, utilizar as bibliotecas padrões fornecidas pela Espressif ou outras fontes leva a um problema de portabilidade do código. Por exemplo, se o projeto evoluir para outra arquitetura de microcontrolador, seria necessário reescrever todo o código específico de ESP-32 (SYSTEMS, 2025) para a nova arquitetura de hardware. Outra problemática é executar testes automatizados em unidades que dependem desse tipo de biblioteca, já que as dependências não podem ser compiladas para Linux ou outro sistema operacional.

Para resolver esses problemas, foi criado um módulo de abstração do hardware (DOUGLASS, 2010), que consiste em fornecer uma interface padrão para as funcionalidades de

hardware e, através de inversão de dependências, diferentes implementações das funcionalidades podem ser facilmente criadas, como um caso específico do padrão Adapter. Como a implementação na arquitetura da ESP-32, a implementação em outra arquitetura ou de um mock, que torna possível realizar testes automatizados durante o desenvolvimento sem a necessidade de ter o microcontrolador em mãos.

Para garantir que o código compilado dos testes da aplicação não inclua as dependências de bibliotecas externas, é utilizada a diretiva de compilação "#ifndef". Assim, esses arquivos são inclusos apenas na build para o microcontrolador.

Como pode ser visto no código 7, a interface para comunicação I2C é definida de forma genérica de acordo com as necessidades do projeto. A partir dela, podem ser criadas classes que implementam o contrato estabelecido, como no framework arduino em 9 e 10 ou em um mock para que seja possível criar qualquer caso de teste desejado em 8.

Código Fonte 7 – Código do HAL da Interface para comunicação I2C

```
#pragma once

namespace machine{
    class I2C{
    public:
        virtual void beginTransmission(int address) = 0;
        virtual void write(int data) = 0;
        virtual void endTransmission() = 0;
        virtual void requestFrom(int address, int quantity) = 0;
        virtual int available() = 0;
        virtual int read() = 0;
    };
}
```

Fonte: Elaborado pelo autor (2025)

Código Fonte 8 – Código do HAL da Implementação do Mock para comunicação I2C

```
/* This code was generated. DO NOT EDIT */

#include <machine/i2c/i2c.h>
#include <gmock/gmock.h>

namespace machine {
    class MockI2C : public I2C {
    public:
        MOCK_METHOD(void, beginTransmission, (int address), (override));
        MOCK_METHOD(void, write, (int data), (override));
        MOCK_METHOD(void, endTransmission, (), (override));
        MOCK_METHOD(void, requestFrom, (int address, int quantity), (override));
        MOCK_METHOD(int, available, (), (override));
        MOCK_METHOD(int, read, (), (override));

    };
}
```

Fonte: Elaborado pelo autor (2025)

Código Fonte 9 – Código do HAL da Interface do Framework Arduino para comunicação I2C

```
#ifndef TESTING

#pragma once

#include "machine/i2c/i2c.h"

namespace machine{
    class ArduinoI2C : public I2C{
    public:
        ArduinoI2C(int sda, int scl);
        void beginTransmission(int address);
        void write(int data);
        void endTransmission();
        void requestFrom(int address, int quantity);
        int available();
        int read();
    };
}

#endif
```

Fonte: Elaborado pelo autor (2025)

Código Fonte 10 – Código do HAL da Implementação do Framework Arduino para comunicação I2C

```
#ifndef TESTING

#include <machine/i2c/arduinoi2c.h>
#include <Arduino.h>
#include <Wire.h>

namespace machine {
    ArduinoI2C::ArduinoI2C(int sda, int scl){
        Wire.setPins(sda, scl);
        Wire.begin();
    }

    void ArduinoI2C::beginTransmission(int address){
        Wire.beginTransmission(address);
    }

    void ArduinoI2C::write(int data){
        Wire.write(data);
    }

    void ArduinoI2C::endTransmission(){
        Wire.endTransmission();
    }

    void ArduinoI2C::requestFrom(int address, int quantity){
        Wire.requestFrom(address, quantity);
    }

```

```
    int ArduinoI2C::available(){
30      return Wire.available();
    }
32
    int ArduinoI2C::read(){
34      return Wire.read();
    }
36  }

38  #endif
```

Fonte: Elaborado pelo autor (2025)

Para utilizar esse módulo corretamente, é necessário usar injeção de dependência, como no exemplo em 11. Ao instanciar a classe RealFlowSensor, basta fornecer uma instância da implementação de I2C desejada como argumento do construtor.

Código Fonte 11 – Código do HAL da Implementação do Framework Arduino para comunicação I2C

```
  #pragma once
2
  #include <measurement/measurement.h>
4 #include <machine/concurrency/concurrency.h>
  #include <machine/i2c/i2c.h>
6
  namespace measurement {
8
  class RealFlowSensor: public FlowSensorProxy {
10   private:
      ...
12    machine::I2C& i2c;
      machine::Concurrency& concurrency;
14
     public:
16    RealFlowSensor(
          machine::I2C& i2c,
18        machine::Concurrency& concurrency
      ):
20        i2c(i2c),
          concurrency(concurrency)
22    {}
      ...
```

Fonte: Elaborado pelo autor (2025)

3.2 DESENVOLVIMENTO DO SISTEMA DE CALIBRAÇÃO

A Calibração do sensor é uma funcionalidade essencial para garantir que a coleta seja armazenada corretamente e as análises sejam precisas, baseadas em dados reais.

O protocolo de calibração adotado no sistema envolve a utilização de uma seringa de calibração com volume conhecido, preenchendo e liberando todo o ar de sua câmara por meio da movimentação do êmbolo. O fluxo de ar gerado pela seringa é conectado ao sensor

do sistema, permitindo ajustar a leitura do dispositivo para garantir que a amplitude do sinal esteja correta. Este procedimento é semelhante ao utilizado na calibração de ventiladores pulmonares, conforme descrito por (MARTELO, 2015), que propõe um método técnico baseado em uma seringa de calibração para garantir a precisão em dispositivos de ventilação mecânica em ambientes clínicos.

Nesse sentido, além da calibração de amplitude so sinal, foi implementado um protocolo de calibração de nível, no qual o sensor é posicionado em um local sem fluxo de ar. Dessa forma, primeiro calibra-se o deslocamento vertical do sinal e, após isso, há o procedimento prévio de calibração de amplitude com a seringa de calibração.

Esse protocolo foi implementado diretamente do dispositivo embarcado, através da interface gráfica, com os passos que o usuário necessita realizar escritos de forma sequencial na tela. Assim, não há a necessidade de um computador, como na versão anterior. Para que a calibração persista, foi criada uma tabela de configurações do dispositivo no banco de dados local, que é checada ao ligar o aparelho e alterada quando o usuário realiza calibração.

3.3 PERSISTÊNCIA DE DADOS

Visando a independência do sistema da internet, é necessário realizar armazenamento local. Portanto, um o banco de dados adequado para essa aplicação é o SQLite , que permite realizar o armazenamento de dados relacionais em um arquivo no computador do usuário, podendo ser acessado através de comandos SQL.

Além dos dados finais serem armazenados em um banco de dados SQL, também é necessário armazenar os dados dentro do software embarcado. Nesse sentido, a equipe de hardware projetou uma memória externa via cartão sd, que pode ser utilizada para esse fim.

3.3.1 Geração de Código de Repositórios com SQLC

Inicialmente foi realizada a modelagem do esquemático do banco de dados, incluindo tabelas como pacientes, exames e análises, com seus respectivos atributos. Após isso, surge a necessidade de escolher como trabalhar com o banco de dados na aplicação desktop. Em Go, é possível utilizar um ORM (Object-Relational Mapping), query builder ou trabalhar com SQL puro. Na visão da arquitetura limpa ou hexagonal, qualquer opção é válida.

Os ORMs oferecem uma interface de alto nível para manipulação de dados, abstraindo a complexidade das queries SQL e facilitando a conversão entre tabelas e objetos (FOWLER, 2002). Embora essa abordagem simplifique o desenvolvimento em sistemas com consultas mais simples, ela pode se tornar um obstáculo na medida em que as queries se tornam mais complexas, podendo gerar comandos ineficientes ou de difícil otimização.

Por outro lado, trabalhar com SQL puro permite o total controle acerca das operações realizadas no banco de dados, garantindo que todas as consultas sejam escritas de maneira eficiente, com a linguagem SQL. Contudo, é necessário escrever código repetitivo para realizar consultas e conversões dos dados retornados.

Diante dessas considerações, foi optado pelo uso do SQLC (The sqlc Team, 2025), uma ferramenta que permite a escrita de queries SQL em um arquivo à parte e realiza a geração de código em Go de classes do padrão Repository tipado automaticamente para interagir com o banco de dados. Dessa forma, é possível ter o benefício do controle total sobre as queries sem sacrificar facilidade de uso, aumentando a manutenibilidade do projeto.

Como exemplo, o código sql da query de alterar os dados de um exame 16 é utilizado pelo SQLC para gerar um método em Go que realiza essa operação e a estrutura de dados que corresponde aos parâmetros da query 17.

Código Fonte 12 – Exemplo de SQL de entrada para ferramenta SQLC

```sql
-- name: UpdateExam :exec
UPDATE exams
SET
    patient_cpf = ?,
    name = ?,
    date = ?,
    duration = ?,
    flow_data = ?,
    volume_data = ?
WHERE
    id = ?;
```

Fonte: Elaborado pelo autor (2025)

Código Fonte 13 – Exemplo de código gerado automaticamente pela ferramenta SQLC

```go
...
const updateExam = `-- name: UpdateExam :exec
UPDATE exams
SET patient_cpf = ?,
    name = ?,
    date = ?,
    duration = ?,
    flow_data = ?,
    volume_data = ?
WHERE id = ?
`

type UpdateExamParams struct {
    PatientCpf string
    Name       string
    Date       time.Time
    Duration   int64
    FlowData   []byte
    VolumeData []byte
    ID         string
}
```

```
23 func (q *Queries) UpdateExam(ctx context.Context, arg UpdateExamParams) error {
       _, err := q.db.ExecContext(ctx, updateExam,
25         arg.PatientCpf,
           arg.Name,
27         arg.Date,
           arg.Duration,
29         arg.FlowData,
           arg.VolumeData,
31         arg.ID,
       )
33     return err
   }
```

Fonte: Elaborado pelo autor (2025)

3.3.2 Migrações com Goose

Como o projeto é de desenvolvimento contínuo, é provável que ocorram mudanças na estrutura do banco de dados ao longo do tempo. Dessa forma, é necessário utilizar migrações para estabelecer um versionamento da estrutura do banco de dados. Dessa forma, torna-se possível reverter mudanças para uma versão anterior e não corromper os dados de um cliente que atualize seu sistema, realizando as migrações gradualmente, por exemplo (AYEESHA; BHATIA, 2009). Existem diversas ferramentas que permitem isso. Foi utilizado o projeto Goose (PRESSLY, 2025) porque é facilmente integrável com o SQLC.

Com isso, foi criado um único módulo que contém as migrações, queries e repositórios gerados pelo SQLC, concentrando todo o código de persistência de dados em um lugar, promovendo a manutenibilidade do projeto.

3.3.3 Software Embarcado

Para armazenar os dados no software embarcado, a abordagem inicial foi utilizar de arquivos binários no sistema SPIFFS, que é o sistema de arquivos padrão da ESP-32, criando manualmente as operações com arquivos para salvar e extrair dados da memória flash. Por ser um procedimento manual, com código repetitivo propício a erros, foi buscada uma outra solução.

Foi encontrado um projeto que permitia o uso do banco de dados SQLite na ESP-32. Como o SQLite já estava sendo utilizado no software desktop, foi considerado o uso também no embarcado. Contudo, seria necessário escrever manualmente o código no padrão Repository para utilizar no projeto.

Para superar esse obstáculo, o autor desenvolveu suporte do projeto SQLC para C++ embarcado, gerando código de repositório específico para a biblioteca de SQLite para ESP-32. Isso foi possível realizando um fork de um plugin para SQLC que permite utilizar templates da linguagem Go para gerar código de forma arbitrária (DIETZE, 2025). Dessa forma, dezenas de linhas de queries em SQL, baseadas nas queries do projeto desktop

puderam gerar mais de mil linhas de código de repositório em C++. A partir do mesmo código SQL anterior 16, é gerado o código correspondente em C++ 18.

Ocorreram problemas de compatibilidade com o sistema de arquivos SPIFFS, então foi alterado para o LittleFS, que não apresentou conflitos com o SQLite embarcado. Com essas ferramentas, se tornou simples e produtivo trabalhar com persistência de dados na ESP-32.

Código Fonte 14 – Exemplo de Código gerado automaticamente pelo plugin da ferramenta SQLC desenvolvido para SQLite na ESP-32

```
...
void SqliteRepository::UpdateExam(
    int patient_cpf,
    std::string name,
    std::string date,
    int duration,
    std::vector<uint8_t> flow_data,
    std::vector<uint8_t> volume_data,
    int id
){
    std::string sql = "UPDATE exams SET patient_cpf = ?, name = ?, date = ?,
        duration = ?, flow_data = ?, volume_data = ? WHERE id = ?";
    sqlite3_stmt* stmt = nullptr;
    err = sqlite3_prepare_v2(toSqlite3(this->database), sql.c_str(), -1, &stmt,
        nullptr);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));

        return ;
    }
    err = sqlite3_bind_int(stmt, 1, patient_cpf);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_bind_text(stmt, 2, name.c_str(), -1, SQLITE_STATIC);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_bind_text(stmt, 3, date.c_str(), -1, SQLITE_STATIC);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_bind_int(stmt, 4, duration);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
```

```
    err = sqlite3_bind_blob(stmt, 5, flow_data.data(), flow_data.size(),
        SQLITE_STATIC);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_bind_blob(stmt, 6, volume_data.data(), volume_data.size(),
        SQLITE_STATIC);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_bind_int(stmt, 7, id);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_step(stmt);
    if (err != SQLITE_DONE) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return;
    }
    sqlite3_finalize(stmt);
    err = 0;
}
...
```

Fonte: Elaborado pelo autor (2025)

3.4 COMUNICAÇÃO ENTRE DISPOSITIVO EMBARCADO E SOFTWARE DESKTOP

Pelas restrições orçamentárias do projeto com relação a regulamentação inicial do dispositivo na Anvisa, não haverá comunicação sem fio do microcontrolador no momento, seja por Bluetooth ou chip Wi-Fi. Dado isso, a melhor opção de interação com o computador é comunicação serial via USB-C.

Para que seja possível a comunicação, é criada uma tarefa no RTOS do sistema embarcado que é dedicada para escutar as mensagens enviadas pelo software desktop e encaminhar para um controlador, que irá destinar a requisição para o serviço apropriado.

Além disso, é criada uma co-rotina no software desktop que irá buscar pelo dispositivo embarcado entre as portas seriais do computador e estabelecer a comunicação, escutando comandos do embarcado na porta encontrada.

Existem diversos protocolos de comunicação que poderiam ser implementados na comunicação serial, contudo, foi optado por criar um protocolo simples, no qual mensagens são enviadas com um título e corpo, no formato estabelecido no código 15. Dessa forma, o título pode ser utilizado como o identificador de uma funcionalidade e seus argumentos

podem ser passados no corpo da mensagem.

Por exemplo, o dispositivo embarcado envia constantemente a medição do sensor de fluxo para que o software desktop mostre o gráfico em tempo real da coleta ou um paciente criado pelo software desktop pode ser enviado para o embarcado, de modo a sincronizar a base de dados.

Código Fonte 15 – Protocolo de comunicação serial

```
#include "com/com.h"
#include <machine/serial/serial.h>
#include <string>

namespace com {
    void Communication::sendMessage(std::string title, std::string message){
        serialA.printf("%s;%s;\n", title.c_str(), message.c_str());
    }
}
```

Fonte: Elaborado pelo autor (2025)

3.5 DESIGN E IMPLEMENTAÇÃO DAS INTERFACES GRÁFICAS

3.5.1 Prototipação de Telas com Figma

A prototipação das telas do software desktop e embarcado é uma atividade essecial, visto que é mais ágil testar estilos e modificar a estrutura de um protótipo visual. Para isso, foi utilizada a platforma Figma (Figma, Inc., 2016), que proporciona ferramentas de prototipação de telas, com fácil manipulação de seus componentes. Também é possível importar componentes e estilos desenvolvidos pela comunidade, tornando prático transformar designs do Figma em páginas web.

3.5.2 Escolha de Bibliotecas Gráficas para Aplicação Desktop

Devido ao estilo de design minimalista da startup Add Health, a aplicação foi baseada na biblioteca de estilos Shadcn/UI (Shadcn, 2023), desenvolvida para o framework React (Meta Open Source, 2013). Dessa forma, o React foi o framework escolhido para otimizar o tempo de desenvolvimento do frontend.

Para estilização, foi utilizada a biblioteca de CSS Tailwind (Tailwind Labs, 2017), que proporciona um sistema de design padronizado por toda a aplicação, estilizando elementos através de classes de HTML. Dessa forma, foi minimizada a quantidade de arquivos de frontend, concentrando as informações de componentes e páginas em seus respectivos arquivos.

3.5.3 Desenvolvimento de Interface Embarcada

O maior diferencial com relação à versão anterior do produto é a construção da interfaca gráfica embarcada. Foi desenvolvida toda a infraestrutura necessária, pois não foi utilizado um framework que gerencie a parte gráfica da tela TFT embarcada. Nesse sentido, foi utilizada uma biblioteca que permite desenhar figuras geométricas, imagens e texto na tela. A partir disso, foi necessário criar uma fundação para o desenvolvimento.

3.5.3.1 Escolha de Biblioteca Gráfica para o Dispositivo Embarcado

Para a construção da interface gráfica, foi utilizada a biblioteca de C++ TFT_eSPI (Bodmer, 2017) como base para desenhar retas, textos e imagens na tela embarcada. Há alguns recursos avançados, como personalização de fontes e antialiasing de algumas figuras geométricas, como círculos, tornando o visual mais suave para o usuário.

3.5.3.2 Desenvolvimento de Componentização

O primeiro passo para tornar robusta a interface gráfica é a componentização , ou seja, encapsular o comportamento e visual de um elemento gráfico em uma classe, por exemplo . Isso permite a reutilização de componentes na aplicação de forma paramétrica, como botões que possuem o mesmo comportamento característico e diferem apenas de bordas, cores ou símbolos (GAMMA et al., 1994).

Não basta apenas agrupar o código de um determinado objeto gráfico, deve ser possível ter uma hierarquia de componentes que podem ser compostos por outros elementos , elevando o grau de reutilização e separação de preocupações dos elementos . Através da composição de widgets, é possível gerenciar o ciclo de vida da interface gráfica de forma mais eficiente, delegando gerenciamento de eventos e como os objetos são renderizados na tela.

3.5.3.3 Implementação de Gerenciamento de Eventos

O usuário interage com a tela através de toques ou gestos. Para gerenciar essas ações, pode ser utilizado da composição de elementos para realizar o gerenciamento de eventos. Um evento de toque pode ser emitido para uma tela que contém diversos elementos. Fica a cargo da tela responder esse evento e/ou delegar para os componentes filhos e assim por diante. A estratégia abordada foi criar contextos delimitados para os componentes, de forma que se um toque for performado em uma coordenada que faça parte da área desse componente, o evento possa ser redirecionado para ele e seus componentes filhos de forma recursiva. Com essa abstração sendo lidada dentro da classe de componentes e telas, facilita a construção de interfaces gráficas com camadas de componentes. Contudo, ainda há um problema. Esse tipo de comunicação é unidirecional, do componente de maior nível para os de menor nível.

Para resolver isso, foi criado um sistema de handlers, onde uma função anônima pode ser passada como argumento para a criação de certos tipos de componentes, que pode ser invocada pelo componente de menor nível. Por exemplo, uma tela contém um componente de botão, que recebeu um evento de clique e pode invocar a função anônima para acionar um comportamento delegado pela tela. Esse padrão foi utilizado no menu da aplicação, por exemplo. Há o componente de lista de itens onde o usuário pode selecionar a seção desejada para navegar, mas o estado de qual item foi selecionado está contido nesse componente. Então, um handler pode ser passado para toda vez que o usuário selecionar um item, acionar uma função e enviar o estado para a tela. Assim como há um botão que ao ser pressionado, redireciona para a tela destino.

3.5.3.4 Desenvolvimento de Arquitetura Orientada a Eventos

No exemplo anterior, o botão da página de menu teria que se comunicar com o componente de mais alto nível para redirecionar as telas. Não seria prático criar handlers aninhados que transmitam mensagens de componentes de baixo nível. Para isso, foi necessário criar uma arquitetura orientada a eventos, de forma que componentes topologicamente distantes possam comunicar-se de forma simplificada. Um padrão interessante para isso é o Mediator, que fornece um meio para que elementos possam se comunicar de maneira desacoplada através de eventos.

No mediator, classes podem publicar eventos em um determinado tópico enquanto outros assinam e reagem a esses eventos através de handlers em uma comunicação n para n. Contudo, há um problema de repetição de código caso o mecanismo de injeção de dependência seja a passagem pelo construtor dos objetos, pois em uma grande hierarquia de componentes, haveria a necessidade de criar um mediator na raiz e enviar para todos os elementos.

Para solucionar esse problema, foi utilizado o design pattern Registry (FOWLER, 2002), que fornece um mecanismo de injeção de dependências independente do construtor. Ele é formado através do design pattern singleton, que é um mecanismo de obtenção de uma instância única globalmente na aplicação ao criar uma classe. Com um mediator registrado no registry, as classes bases para componentes e telas podem obter a instância global do mediator de forma oculta, facilitando o desenvolvimento posterior e simplificando o código.

3.5.3.5 Criação de Roteamento de Telas

Uma aplicação para o sistema de eventos global criado foi o roteamento de telas. Várias telas podem ser instanciadas e relacionadas com um identificador, similar a uma url de navegador. Com uma simples chamada de função para um endpoint “/registration”, por exemplo, pode ser enviado um evento global, que está sendo escutado pela classe responsável pelo roteamento das telas e trocar a tela a ser processada e renderizada. Isso torna trivial a implementação de novas telas, simplificando o programa.

3.5.3.6 Implementação de RTOS e Multitaskting

Com a infraestrutura descrita anteriormente, o desenvolvimento de funcionalidades gráficas é simplificado. Contudo, já havia sido desenvolvida uma robusta camada de aplicação onde use cases como cadastro de usuários e análise e gerenciamento de exames estão presentes. Seguindo a arquitetura hexagonal, o frontend embarcado atua como driver dessa aplicação, responsável por chamar os casos de uso, conduzindo a aplicação.

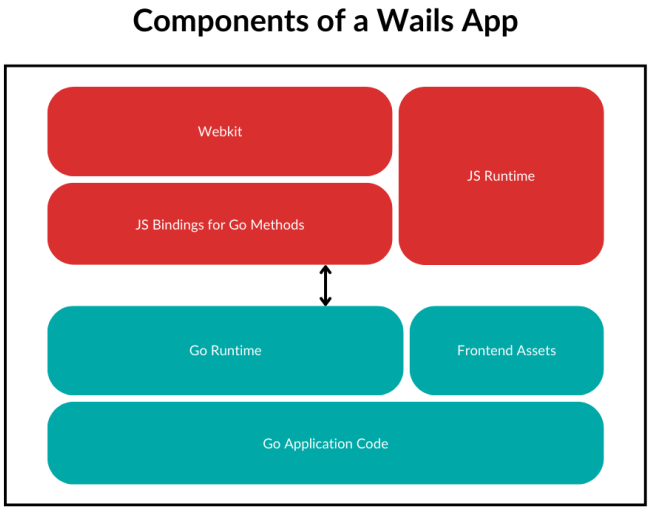
Porém, se a aplicação inteira for desenvolvida sem utilizar conceitos de programação concorrente, é possível que a aplicação sofra problemas de desempenho. Um exemplo é a lentidão na interface gráfica decorrente de um longo processamento e utilização de recursos externos, danificando a experiência de usuário. Logo, o frontend foi posto em uma Tarefa diferente do processamento de dados e da aquisição de dados. Dessa forma, a experiência de usuário se torna mais fluida.

3.6 APLICAÇÃO DESKTOP

Para o desenvolvimento da aplicação desktop, foram considerados alguns frameworks. O Flutter foi inicialmente analisado por sua capacidade de compilar para desktop e dispositivos móveis, o que poderia ser vantajoso para futuras expansões. Também foi cogitado o uso do Electron, que permite a criação de aplicações multiplataforma com tecnologias web (HTML, CSS, JavaScript), oferecendo flexibilidade ao desenvolver o frontend. Por fim, o Wails foi outra opção, que integra um backend em Go com o frontend web em um único executável de aplicaçãoo desktop.

Após avaliar essas opções, o projeto Wails foi escolhido devido à sua proposta de permitir o desenvolvimento de aplicações desktop com a performance e a simplicidade do Go no backend, e a flexibilidade do frontend com HTML, CSS e JavaScript. Nesse contexto, o framework proporciona uma integração nativa entre o Go e o frontend, permitindo um desenvolvimento mais direto e ágil, como pode ser visto em detalhes na figura 9. Isso, aliado ao fato de o Go ser uma linguagem que proporciona robustez e escalabilidade, tornou o Wails a opção mais vantajosa para o projeto.

Figura 9 – Componentes de uma aplicação Wails



Fonte: (WAILS, 2025).

3.7 PERSISTÊNCIA DE DADOS

Visando a independência do sistema da internet, é necessário realizar armazenamento local. Portanto, um o banco de dados adequado para essa aplicação é o SQLite , que permite realizar o armazenamento de dados relacionais em um arquivo no computador do usuário, podendo ser acessado através de comandos SQL.

Além dos dados finais serem armazenados em um banco de dados SQL, também é necessário armazenar os dados dentro do software embarcado. Nesse sentido, a equipe de hardware projetou uma memória externa via cartão sd, que pode ser utilizada para esse fim.

3.7.1 Geração de Código de Repositórios com SQLC

Inicialmente foi realizada a modelagem do esquemático do banco de dados, incluindo tabelas como pacientes, exames e análises, com seus respectivos atributos. Após isso, surge a necessidade de escolher como trabalhar com o banco de dados na aplicação desktop. Em Go, é possível utilizar um ORM (Object-Relational Mapping), query builder ou trabalhar com SQL puro. Na visão da arquitetura limpa ou hexagonal, qualquer opção é válida.

Os ORMs oferecem uma interface de alto nível para manipulação de dados, abstraindo a complexidade das queries SQL e facilitando a conversão entre tabelas e objetos (FOWLER, 2002). Embora essa abordagem simplifique o desenvolvimento em sistemas com consultas mais simples, ela pode se tornar um obstáculo na medida em que as queries se tornam mais complexas, podendo gerar comandos ineficientes ou de difícil otimização.

Por outro lado, trabalhar com SQL puro permite o total controle acerca das operações realizadas no banco de dados, garantindo que todas as consultas sejam escritas de maneira

eficiente, com a linguagem SQL. Contudo, é necessário escrever código repetitivo para realizar consultas e conversões dos dados retornados.

Diante dessas considerações, foi optado pelo uso do SQLC (The sqlc Team, 2025), uma ferramenta que permite a escrita de queries SQL em um arquivo à parte e realiza a geração de código em Go de classes do padrão Repository tipado automaticamente para interagir com o banco de dados. Dessa forma, é possível ter o benefício do controle total sobre as queries sem sacrificar facilidade de uso, aumentando a manutenibilidade do projeto.

Como exemplo, o código sql da query de alterar os dados de um exame 16 é utilizado pelo SQLC para gerar um método em Go que realiza essa operação e a estrutura de dados que corresponde aos parâmetros da query 17.

Código Fonte 16 – Exemplo de SQL de entrada para ferramenta SQLC

```sql
-- name: UpdateExam :exec
UPDATE exams
SET
    patient_cpf = ?,
    name = ?,
    date = ?,
    duration = ?,
    flow_data = ?,
    volume_data = ?
WHERE
    id = ?;
```

Fonte: Elaborado pelo autor (2025)

Código Fonte 17 – Exemplo de código gerado automaticamente pela ferramenta SQLC

```go
...
const updateExam = `-- name: UpdateExam :exec
UPDATE exams
SET patient_cpf = ?,
    name = ?,
    date = ?,
    duration = ?,
    flow_data = ?,
    volume_data = ?
WHERE id = ?
`

type UpdateExamParams struct {
    PatientCpf string
    Name       string
    Date       time.Time
    Duration   int64
    FlowData   []byte
    VolumeData []byte
    ID         string
}

func (q *Queries) UpdateExam(ctx context.Context, arg UpdateExamParams) error {
    _, err := q.db.ExecContext(ctx, updateExam,
        arg.PatientCpf,
```

```
            arg.Name,
            arg.Date,
            arg.Duration,
            arg.FlowData,
            arg.VolumeData,
            arg.ID,
        )
        return err
    }
```

Fonte: Elaborado pelo autor (2025)

3.7.2 Migrações com Goose

Como o projeto é de desenvolvimento contínuo, é provável que ocorram mudanças na estrutura do banco de dados ao longo do tempo. Dessa forma, é necessário utilizar migrações para estabelecer um versionamento da estrutura do banco de dados. Dessa forma, torna-se possível reverter mudanças para uma versão anterior e não corromper os dados de um cliente que atualize seu sistema, realizando as migrações gradualmente, por exemplo (AYEESHA; BHATIA, 2009). Existem diversas ferramentas que permitem isso. Foi utilizado o projeto Goose (PRESSLY, 2025) porque é facilmente integrável com o SQLC.

Com isso, foi criado um único módulo que contém as migrações, queries e repositórios gerados pelo SQLC, concentrando todo o código de persistência de dados em um lugar, promovendo a manutenibilidade do projeto.

3.7.3 Software Embarcado

Para armazenar os dados no software embarcado, a abordagem inicial foi utilizar de arquivos binários no sistema SPIFFS, que é o sistema de arquivos padrão da ESP-32, criando manualmente as operações com arquivos para salvar e extrair dados da memória flash. Por ser um procedimento manual, com código repetitivo propício a erros, foi buscada uma outra solução.

Foi encontrado um projeto que permitia o uso do banco de dados SQLite na ESP-32. Como o SQLite já estava sendo utilizado no software desktop, foi considerado o uso também no embarcado. Contudo, seria necessário escrever manualmente o código no padrão Repository para utilizar no projeto.

Para superar esse obstáculo, o autor desenvolveu suporte do projeto SQLC para C++ embarcado, gerando código de repositório específico para a biblioteca de SQLite para ESP-32. Isso foi possível realizando um fork de um plugin para SQLC que permite utilizar templates da linguagem Go para gerar código de forma arbitrária (DIETZE, 2025). Dessa forma, dezenas de linhas de queries em SQL, baseadas nas queries do projeto desktop puderam gerar mais de mil linhas de código de repositório em C++. A partir do mesmo código SQL anterior 16, é gerado o código correspondente em C++ 18.

Ocorreram problemas de compatibilidade com o sistema de arquivos SPIFFS, então foi alterado para o LittleFS, que não apresentou conflitos com o SQLite embarcado. Com essas ferramentas, se tornou simples e produtivo trabalhar com persistência de dados na ESP-32.

Código Fonte 18 – Exemplo de Código gerado automaticamente pelo plugin da ferramenta SQLC desenvolvido para SQLite na ESP-32

```
...
void SqliteRepository::UpdateExam(
    int patient_cpf,
    std::string name,
    std::string date,
    int duration,
    std::vector<uint8_t> flow_data,
    std::vector<uint8_t> volume_data,
    int id
){
    std::string sql = "UPDATE exams SET patient_cpf = ?, name = ?, date = ?,
        duration = ?, flow_data = ?, volume_data = ? WHERE id = ?";
    sqlite3_stmt* stmt = nullptr;
    err = sqlite3_prepare_v2(toSqlite3(this->database), sql.c_str(), -1, &stmt,
        nullptr);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));

        return ;
    }
    err = sqlite3_bind_int(stmt, 1, patient_cpf);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_bind_text(stmt, 2, name.c_str(), -1, SQLITE_STATIC);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_bind_text(stmt, 3, date.c_str(), -1, SQLITE_STATIC);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_bind_int(stmt, 4, duration);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_bind_blob(stmt, 5, flow_data.data(), flow_data.size(),
        SQLITE_STATIC);
    if (err != SQLITE_OK) {
```

```
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_bind_blob(stmt, 6, volume_data.data(), volume_data.size(),
        SQLITE_STATIC);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_bind_int(stmt, 7, id);
    if (err != SQLITE_OK) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return ;
    }
    err = sqlite3_step(stmt);
    if (err != SQLITE_DONE) {
        Serial.printf("SQL error: %s\n", sqlite3_errmsg(toSqlite3(this->database)
            ));
        return;
    }
    sqlite3_finalize(stmt);
    err = 0;
}
...
```

Fonte: Elaborado pelo autor (2025)

4 RESULTADOS

Este trabalho teve como objetivo o desenvolvimento de um sistema para análise de padrões respiratórios, abordando tanto os aspectos teóricos quanto práticos relacionados ao projeto. Ao longo da pesquisa e implementação, foram discutidas e aplicadas técnicas de arquitetura de software, com foco na utilização da Arquitetura Hexagonal, buscando garantir a modularidade, testabilidade e flexibilidade do sistema.

O desenvolvimento seguiu um processo iterativo baseado na experimentação e refinamento contínuo, no qual cada funcionalidade foi projetada, implementada e testada de forma incremental. Esse método permitiu validar constantemente as decisões de design, garantindo que cada módulo fosse ajustado para atender aos requisitos do sistema de forma eficiente.

Nesse sentido, a Tabela 2 indica os desafios encontrados para atingir cada meta proposta e as soluções desenvolvidas para superar as dificuldades e alcançar os objetivos. A maioria dos objetivos individuais foi concluída em sua totalidade. No entanto, não foi possível finalizar em tempo hábil algumas funcionalidades, como componentes gráficos para tela de ajuda e tela de cadastro de pacientes do dispositivo embarcado. Além disso, houveram testes de software referentes aos algoritmos de calibração, contudo o equipamento padrão ouro de medição de fluxo e volume, para realizar uma análise comparativa com as medições da versão nova do RDA não estava disponível para teste de bancada, que deve ser realizado quando disponível.

Tabela 2 – Resultados das metas

Meta	Alcance	Desafios	Soluções
Revisão bibliográfica sobre arquitetura de software e análise de sistemas respiratórios.	100%	Encontrar exemplos práticos de padrões de projeto para C++ embarcado com testes independentes do hardware físico.	Exploração de métodos de inversão de dependência na etapa de compilação para desacoplar testes do hardware.
Persistência de dados no sistema embarcado.	100%	Ausência de ferramentas que facilitem a camada de persistência em SQL embarcado.	Desenvolvimento de infraestrutura para transpilar código SQL em C++ compatível com SQLite no ESP32.
Desenvolvimento de interface gráfica embarcada e desktop.	90%	Limitações da biblioteca gráfica embarcada quanto a funcionalidades e customização.	Criação de um framework para interfaces gráficas, com componentização, roteamento de telas e gerenciamento de eventos.
Desenvolvimento de sistema de calibração.	80%	Necessidade de testes manuais de bancada para garantir a funcionalidade do sistema.	Implementação de testes automatizados utilizando um Mock do sensor real.
Comunicação e sincronização de dados entre embarcado e desktop.	90%	Estruturar comandos para transmissão eficiente e compreensível entre dispositivos.	Criação de um protocolo de comunicação baseado em comandos estruturados com título, corpo em texto e suporte a JSON.
Desenvolvimento de arquitetura testável	100%	Criação manual de classes de Mocks para substituir dependências em testes no embarcado.	Desenvolvimento de uma ferramenta automatizada para geração de Mocks com base nas interfaces existentes.
Desenvolvimento de arquitetura concorrente	100%	Garantir a aquisição contínua e precisa de fluxo em tempo real.	Implementação de tarefas no FreeRTOS, priorizando a medição de fluxo com interrupções em alta precisão.
Desenvolvimento de arquitetura modular e escalável	100%	Importação de múltiplos headers poluindo a sugestão de nomes no ambiente de desenvolvimento.	Organização do código com namespaces específicos para cada módulo, reduzindo conflitos e facilitando a navegação.

Fonte: Elaborada pelo autor (2025)

4.1 TESTABILIDADE E MODULARIDADE DO SISTEMA

A organização do código em camadas, tanto no software embarcado quanto no software desktop, possibilitou a separação das funcionalidades em módulos desacoplados. A camada de serviço dos módulos adotou o modelo Transaction Script, com domínio reduzido a estruturas de dados e funções auxiliares. Dessa forma, as funcionalidades foram estruturadas de maneira procedural e localizada, promovendo simplicidade e clareza no código.

A reutilização de código entre módulos, como persistência de dados via repositório ou uso periférico de hardware, foi viabilizada pelo princípio de inversão de dependências da Arquitetura Hexagonal. Essa abordagem simplificou a realização de testes, permitindo a injeção de versões de teste das dependências de forma isolada, desde que implementassem a interface necessária para a funcionalidade testada.

Para garantir testabilidade no software embarcado, foi essencial a utilização de flags de compilação, assegurando que dependências específicas da arquitetura embarcada não fossem linkadas na etapa de testes. Assim, tornou-se possível executar testes sem necessidade do hardware físico, compilando o código de teste para Linux e integrando os testes em uma pipeline CI/CD.

4.2 MANUTENIBILIDADE E EVOLUÇÃO DO CÓDIGO

A estrutura do projeto foi desenvolvida com base nos princípios SOLID, garantindo que cada unidade de código possuísse um único propósito, facilitando modificações pontuais e extensibilidade futura. Com a Arquitetura Hexagonal, tornou-se viável substituir componentes de hardware e bibliotecas externas de software, limitando as alterações apenas às implementações dos adaptadores ou parâmetros de compilação. Isso proporcionou maior flexibilidade e resiliência, permitindo que o software evolua de forma ágil sem comprometer sua estrutura.

4.3 DESEMPENHO E EFICIÊNCIA

Por se tratar de um sistema de tempo real com requisitos de usabilidade de interface gráfica e processamento concorrente, a utilização de um RTOS embarcado foi fundamental. As tarefas foram alocadas nos núcleos do ESP32 com prioridades adequadas às suas funcionalidades. A medição de fluxo do sensor, por exemplo, foi executada em um núcleo isolado, com a maior prioridade, garantindo intervalos fixos de 20 ms e um erro de cálculo de intervalo de tempo inferior a 1 microssegundo.

4.4 INTERFACE GRÁFICA

A interface gráfica embarcada foi projetada com um design minimalista que, além de reforçar a identidade visual da startup Add Health, também otimizou o desempenho, reduzindo a carga de processamento gráfico. Dessa forma, a interface apresentou baixa latência, sem comprometer a experiência do usuário. A Figura 10 mostra a tela inicial, incluindo as funcionalidades principais do dispositivo: Cadastro de Paciente, Coleta, Análise, Calibração e Ajuda. O usuário seleciona a opção na lista da esquerda e pressiona o ícone correspondente para ser redirecionado.

Figura 10 – Tela Inicial do RDA Embarcado

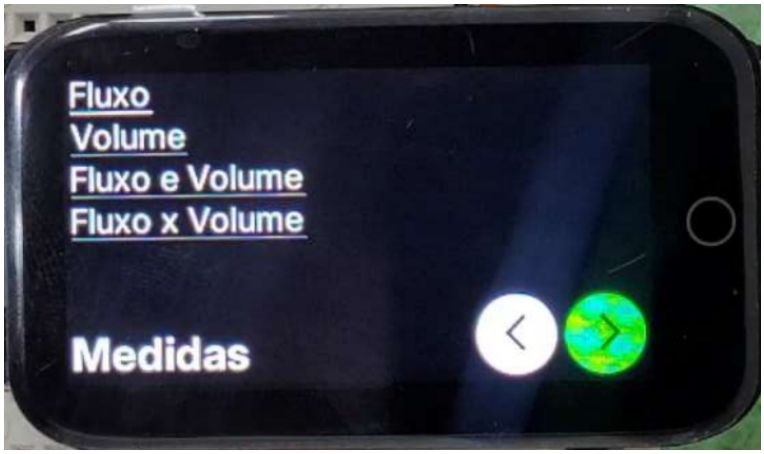
Fonte: Registrado pelo autor (2024).

A tela de coleta inicialmente dispõe o gráfico de fluxo respiratório (Figura 11). Ao tocar no botão de menu, com fundo branco, é aberta a tela de tela de alteração de gráfico (Figura 12), que tem como opções os gráficos de Fluxo, Volume, Fluxo e Volume (Figura 13) e Fluxo por Volume (Figura 14). Em todas as opções, há um gráfico associado à variáve, um botão para iniciar a coleta em verde, um timer no canto superior direito da tela para mostrar o tempo de gravação e informações da curva no canto inferior esquerdo.

Figura 11 – Tela de Gráfico de Fluxo do RDA Embarcado

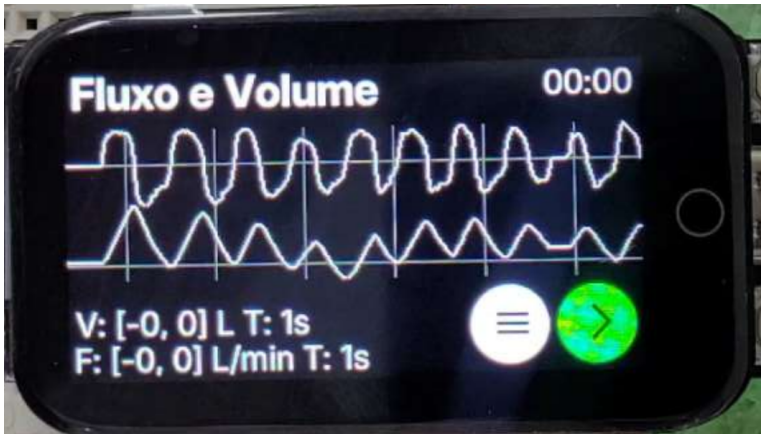
Fonte: Registrado pelo autor (2024).

Figura 12 – Tela de Alteração de Gráfico do RDA Embarcado



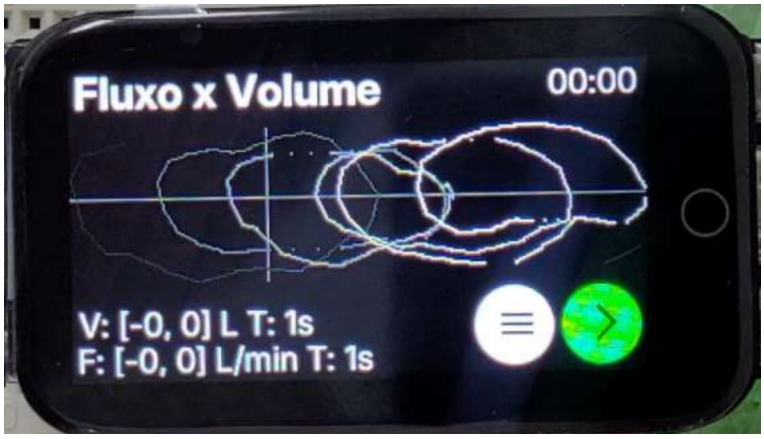
Fonte: Registrado pelo autor (2024).

Figura 13 – Tela de Gráfico de Fluxo e Volume do RDA Embarcado



Fonte: Registrado pelo autor (2024).

Figura 14 – Tela de Gráfico de Fluxo por Volume do RDA Embarcado

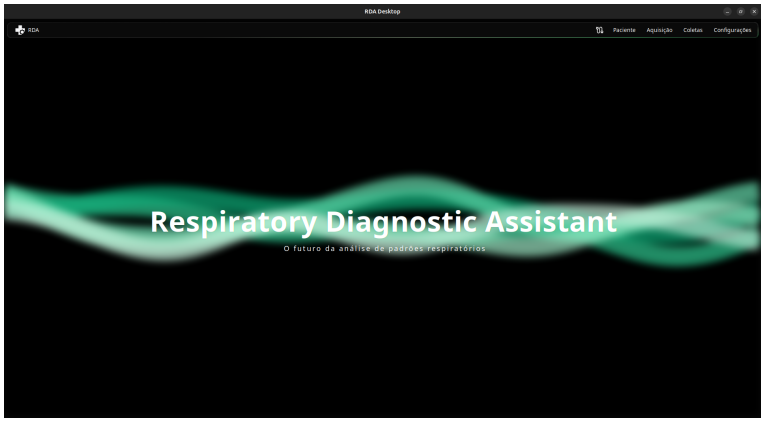


Fonte: Registrado pelo autor (2024).

A aplicação desktop tem uma animação de curvas na tela inicial (Figura 15), com botões no topo para navegar para as seções de Paciente, Aquisição de dados, Base de

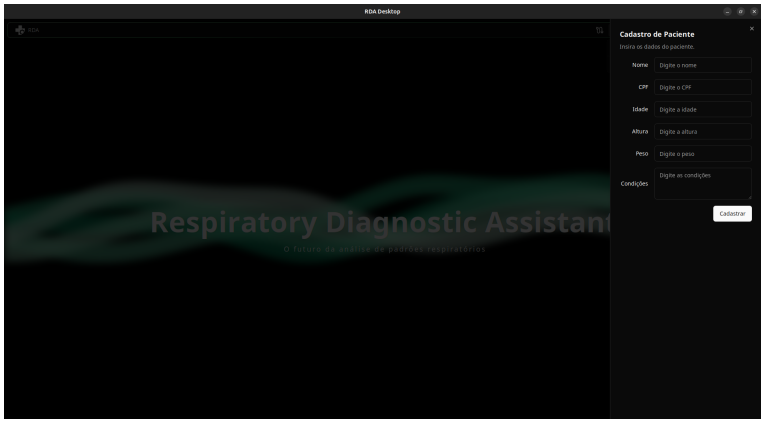
Coletas e Configurações. A Figura 16 mostra o formulário de cadastro para os pacientes, com informações relevantes para uso posterior.

Figura 15 – Tela Inicial do RDA Desktop



Fonte: Registrado pelo autor (2025).

Figura 16 – Tela de Cadastro de Pacientes do RDA Desktop



Fonte: Registrado pelo autor (2025).

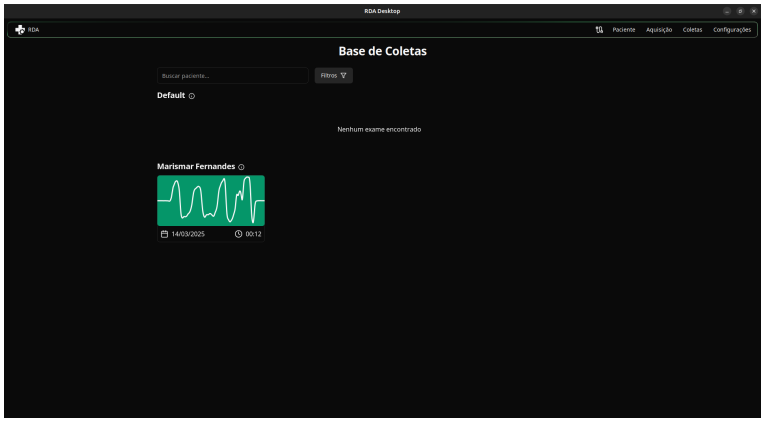
Com um paciente selecionado e o RDA conectado, o usuário pode visualizar a aquisição em tempo real (Figura 17), podendo controlar a coleta pelo dispositivo embarcado ou pelo desktop. Também pode ser alterado o tipo de gráfico no dropdown da seção inferior da página.

Figura 17 – Tela de Coleta em Tempo Real do RDA Desktop

Fonte: Registrado pelo autor (2025).

Ao finalizar a coleta, o exame é armazenado e pode ser descoberto na tela de base de dados (Figura 18), que contém a lista de todos os pacientes cadastrados e seus exames dispostos em carrosséis. O usuário pode procurar pacientes de forma facilitada com o campo de busca em conjunto com os filtros.

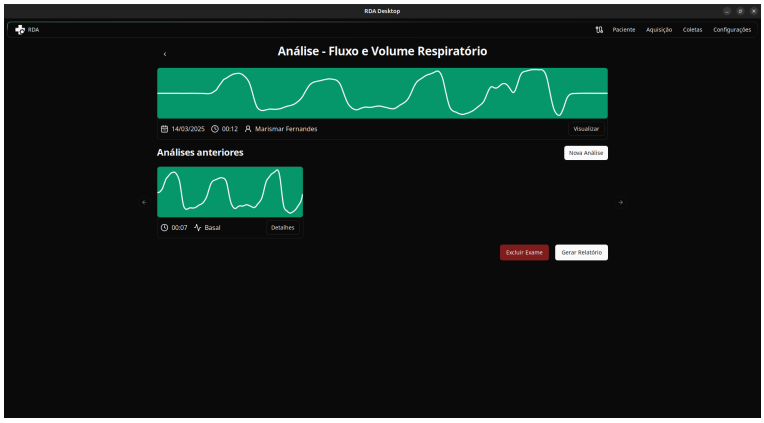
Figura 18 – Tela de Base de Pacientes e Exames do RDA Desktop



Fonte: Registrado pelo autor (2025).

Ao ao clicar em uma coleta na tela de base de dados, o usuário é redirecionado para a tela de análise do exame selecionado (Figura 19). No card do topo, está disposta a curva de fluxo respiratório e metadados, como nome do paciente, data e duração do exame. Abaixo está localizado um carrossel com as análises anteriores realizadas pelo usuário, que indicam a duração do intervalo analisado e o tipo de análise, além de dispor de forma visual a curva de fluxo. O botão de detalhes abre um pop-up com os resultados da análise.

Figura 19 – Tela de Análises de Exame do RDA Desktop



Fonte: Registrado pelo autor (2025).

Por fim, para criar uma análise no exame, basta clicar no botão de "Nova Análise"da tela de análise do exame. Dessa forma, o usuário é redirecionado para a tela de escolha de intervalo para análise (Figura 20), que permite o usuário selecionar o intervalo de análise clicando no ponto de início e fim, podendo usar recursos como zoom e mudar o gráfico visualizado. Quando satisfeito, o usuário escolhe o tipo de análise a ser performada no canto superior direito e pressiona o botão de calcular. Dessa forma, a análise é realizada e estará disponível na seção de análises anteriores do exame.

Figura 20 – Tela de Criação de Análise do RDA Desktop

Fonte: Registrado pelo autor (2025).

5 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresenta uma solução prática para a análise de padrões respiratórios e estabelece uma base robusta que pode ser aplicada a outros produtos, contribuindo para novas pesquisas e inovações no desenvolvimento de sistemas embarcados, análise de dados e tecnologias de monitoramento de padrões respiratórios.

A principal contribuição deste trabalho foi a criação de uma infraestrutura sólida para o desenvolvimento contínuo do RDA. A aplicação de boas práticas e uma arquitetura modular garantem a qualidade do software e facilitam sua evolução, permitindo a incorporação de novas funcionalidades sem comprometer a estrutura existente. Além disso, a abordagem proposta pode ser aplicada a outros produtos que necessitem de modularidade, testabilidade e facilidade de manutenção, beneficiando diferentes contextos no desenvolvimento de software embarcado.

Para trabalhos futuros, algumas melhorias podem ser exploradas. Uma alternativa viável é a adoção da biblioteca LVGL para a interface gráfica embarcada, visto que ela oferece suporte aprimorado para animações e ferramentas visuais que simplificam o design e reduzem a necessidade de testes manuais.

Além disso, a integração do sistema com uma plataforma na nuvem, possivelmente desenvolvida pela startup Add Health, poderia expandir suas funcionalidades. Essa integração permitiria o armazenamento remoto de dados, análise automatizada por inteligência artificial e interoperabilidade com outros sistemas, proporcionando maior valsor ao projeto e ampliando seu impacto.

REFERÊNCIAS

AYEESHA, D.; BHATIA, S. Refactoring of a database. *International Journal of Computer Science and Information Security*, 12 2009.

BARRETT, K. E. et al. *Ganong's Review of Medical Physiology*. [S.l.]: McGraw-Hill Education, 2020.

BECK, K. *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley, 2003. ISBN 978-0321146533.

Bodmer. *TFT_eSPI: A fast SPI TFT library for ESP8266 and ESP32*. 2017. Acesso em: 15 mar. 2025. Disponível em: <https://github.com/Bodmer/TFT_eSPI>.

CAMPOS, I. M.; CAMPOS, S. L.; LEITÃO, H. A. d. S. Patente: Programa de Computador, *RDA analysis*. 2022. Registro: 21/10/2022.

CAMPOS, I. M.; LEITÃO, H. A. d. S.; CAMPOS, S. L.; JUNIOR, G. L. M.; BRITO, M. E. d. C. Patente: Programa de Computador, *RDA Sync*. 2022. Registro: 20/12/2022.

CAMPOS, S. L.; CANHOTO, J. L. O.; JUNIOR, G. L. M.; LEITÃO, H. A. d. S.; BRITO, M. E. d. C.; ANDRADE, A. d. F. D. D. Patente: Privilégio de Inovação, *Dispositivo para Diagnóstico e Treinamento do Padrão da Respiração Humana*. 2016. Registro: 02/09/2016.

COCKBURN, A. *Hexagonal Architecture*. 2005. Acesso em: 24 fev. 2025. Disponível em: <https://alistair.cockburn.us/hexagonal-architecture/>.

COHN, M. *Succeeding with Agile: Software Development Using Scrum*. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 978-0-321-57936-2.

DIETZE, F. *sqlc-gen-from-template: Plugin para gerar código tipado e seguro para consultas SQL usando templates*. 2025. Acesso em: 15 mar. 2025. Disponível em: <https://github.com/fdietze/sqlc-gen-from-template>.

DOUGLASS, B. P. *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*. Oxford, UK: Newnes, 2010. ISBN 978-0-08-095971-9.

Figma, Inc. *Figma: The Collaborative Interface Design Tool*. 2016. Acesso em: 15 mar. 2025. Disponível em: <https://www.figma.com/>.

FOWLER, M. *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley, 2002. ISBN 978-0321127426.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1994. ISBN 978-0201633610.

GANONG, W. F. *Review of Medical Physiology*. [S.l.]: McGraw-Hill Education, 2019.

Google Test Authors. *GoogleTest - Google Testing and Mocking Framework*. [S.l.], 2025. Acesso em: 15 mar. 2025. Disponível em: <https://google.github.io/googletest/>.

GRAVENSTEIN, J. S.; COLLEAGUES. *Capnography*. [S.l.]: Cambridge University Press, 2004.

HANKINSON, J. L. et al. Spirometric reference values from a sample of the general u.s. population. *American Journal of Respiratory and Critical Care Medicine*, v. 159, p. 179–187, 1999.

LAGE, P. S.; SILVA, F. J. M. e; CAMPOS, F. A. de O. Padrões respiratórios patológicos: respiração de cheyne-stokes, respiração de biot, respiração de kussmaul e respiração suspirosa. *Revista Fisioterapia e Terapias Complementares*, v. 10, n. 2, p. 45–52, 2020. Disponível em: <https://revistaft.com.br/padroes-respiratorios-patologicos-respiracao-de-cheyne-stokes-respiracao-de-biot-respiracao-de-kussm
>.

MARTELO, P. *Ventiladores Pulmonares: Definição do Método de Calibração*. Dissertação (Dissertação (Mestrado em Engenharia e Instrumentação Médica)) — Instituto Politécnico do Porto, Porto, 2015. Disponível em: <https://recipp.ipp.pt/bitstream/10400.22/8062/1/DM_PauloMartelo_2015_MEIM.pdf>.

MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Upper Saddle River, NJ: Prentice Hall, 2017. ISBN 978-0134494166.

MEHTA, S.; DHOORIA, A. *Textbook of Pulmonary and Critical Care Medicine*. [S.l.]: Jaypee Brothers Medical Publishers, 2019.

MESZAROS, G. *xUnit Test Patterns: Refactoring Test Code*. Boston, MA: Addison-Wesley, 2007. ISBN 978-0131495050.

Meta Open Source. *React: A JavaScript library for building user interfaces*. 2013. Acesso em: 15 mar. 2025. Disponível em: <https://react.dev/>.

PORTO, C. C. *Porto & Porto Semiologia Médica*. 8. ed. Rio de Janeiro: Guanabara Koogan, 2019. 274–281 p.

PRESSLY. *Goose: Database migration tool for Go*. 2025. Acesso em: 15 mar. 2025. Disponível em: <https://github.com/pressly/goose>.

Selenium Project. *Selenium Documentation*. [S.l.], 2025. Acesso em: 15 mar. 2025. Disponível em: <https://www.selenium.dev/documentation/>.

Shadcn. *shadcn/ui: Beautifully designed components for React*. 2023. Acesso em: 15 mar. 2025. Disponível em: <https://ui.shadcn.com/>.

SILVA, J. H. *Rastreamento de disfunções respiratórias em indivíduos na condição pós-covid-19*. Dissertação (Dissertação (Mestrado em Fisioterapia)) — Universidade Federal de Pernambuco, Recife, 2023. Acesso em: 15 mar. 2025. Disponível em: <https://repositorio.ufpe.br/handle/123456789/51650>.

SYSTEMS, E. *ESP32-S3 Technical Documentation*. [S.l.], 2025. Acesso em: 15 mar. 2025. Disponível em: <https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/get-started/index.html>.

Tailwind Labs. *Tailwind CSS: A utility-first CSS framework*. 2017. Acesso em: 15 mar. 2025. Disponível em: <https://tailwindcss.com/>.

The sqlc Team. *sqlc: Generate Type-Safe Go Code from SQL Queries*. 2025. Acesso em: 15 mar. 2025. Disponível em: <https://sqlc.dev>.

VIDOTTO, L. S.; OUTROS. Disfunção respiratória: o que sabemos? *Jornal Brasileiro de Pneumologia*, v. 45, n. 1, p. e20170347, 2019. Disponível em: <https://www.scielo.br/j/jbpneu/a/d6LjQp4KnpQWmDdLSdWJjNh/?lang=pt>.

WAILS. *Componentes de uma Aplicação Wails*. 2025. Acesso em: 12 mar. 2025. Disponível em: <https://wails.io/docs/howdoesitwork>.

WEST, J. B. *Pulmonary Physiology and Pathophysiology: An Integrated, Case-Based Approach*. [S.l.]: Lippincott Williams & Wilkins, 2012.