INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA

DO RIO GRANDE DO NORTE

CAMPUS CURRAIS NOVOS

LUIZ PAULO DE LIMA ARAÚJO

TITULO DO TRABALHO: SUBTÍTULO

CURRAIS NOVOS - RN

2026

LUIZ PAULO DE LIMA ARAÚJO

TITULO DO TRABALHO: SUBTÍTULO:

Trabalho de conclusão de curso apresentado ao curso de graduação em Tecnologia em Sistemas para Internet, como parte dos requisitos para obtenção do título de Tecnólogo em Sistemas para Internet pelo Instituto Federal do Rio Grande do Norte.

Orientador(a): Orientador do Trabalho.
Co-orientador(a): .

CURRAIS NOVOS - RN
2026

LUIZ PAULO DE LIMA ARAÚJO

TITULO DO TRABALHO: SUBTÍTULO

Trabalho de conclusão de curso apresentado ao curso de graduação em Tecnologia em Sistemas para Internet, como parte dos requisitos para obtenção do título de Tecnólogo em Sistemas para Internet pelo Instituto Federal do Rio Grande do Norte.

CURRAIS NOVOS - RN, xx de mmm de 2026

Orientador do Trabalho
Orientador

Professor
Examinador(a) 1

Professor
Examinador(a) 2

CURRAIS NOVOS - RN
2026

Dedico este trabalho aos meus tias e tios e aos meus pais que sempre apoiaram minha formaçã

“Arquitecti est scientia
pluribus disciplinis et variis
eruditionibus ornata, quae ab ceteris artibus
perficiuntur. Opera ea nascitur et fabrica
et ratiocinatione.
(DE ARCHITECTURA, Liber primus, Caput Primus, signum paragraphi I)

RESUMO

O resumo tem a função de resumir os pontos-chave da monografia, apresentando sucintamente a introdução, metodologia, resultados, discussão e conclusões, além de destacar a relevância do estudo. Deve ser conciso, informativo e atrativo, com uma extensão usualmente entre 150 e 300 palavras, dependendo das diretrizes da instituição ou da revista acadêmica.

Palavras-chave:

ABSTRACT

The abstract serves the purpose of summarizing the key points of the thesis, briefly presenting the introduction, methodology, results, discussion, and conclusions, while highlighting the study's relevance. It should be concise, informative, and engaging, typically ranging from 150 to 300 words, depending on the guidelines of the institution or academic journal.

Keywords:

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

TI Tecnologia da Informação

TSI Tecnologia em Sistemas para Internet

IEEE Institute of Electrical and Eletctronics Engineers

IFRN Instituto Federal do Rio Grande do Norte

OOP Programação Orientada a Objetos

UI Interface de Usuário

DDD Domain Driven Design by Erick Evans

LISTA DE SÍMBOLOS

Γ	Letra Grega Gamma

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

Os computadores eletrônicos e o *software* emergiram como ferramenta de importância notória durante os eventos finais da segunda grande guerra. Seu emprego possibilitara computação de trajetórias balisticas, propriedades das detonações atômicas e quebra de de criptografia inimiga (Dodig-Crnkovic, 2001, p.4).

Poucos anos após o fim do conflito, na década de 1950, houve o desenvolvimento contínuo não somente da tecnologia física como também da lógica "*software*". Programas, antes escritos em binário, agora podiam ser criados em linguagens cada vez mais próximas da linguagem humana (Dodig-Crnkovic, 2001, p.5).

Na década 1960 a Ciência da Computação consolidara-se, de fato, como uma ciência formal e de importância notória para as décadas futuras. Sua abragência utilitária não mais retringia-se acadêmico-militar mas, agora, colaborava adjuntamente aos grandes negócios, instituições públicas e universidades (Newell; Perlis; Simon, 1967).

Via-se um linear, quase exponencial, aumento do poder computacional ao mesmo tempo em que diminuia-se as dimensões físicas dos computadores. De acordo com Almeida (1967) o fundador da Intel, Gordon Moore, publicara na revista *Electronics Magazine*, um artigo, onde previra aumento em dobro do poder computacional a cada 18 meses fato esse que possibilitara o desenvolvimento de aplicações mais complexas.

Entretanto, o avanço não foi insento de problemas e crises. Durante a décadas de 1960 e 1970 houvera o aumento significativo da demanda por *software* pelas organizações comerciais, públicas e de ensino. A situação impeliu o agravamento da chamada "Crise do Software". Um cenário que teve como característica principal a ineficácia dos processos e técnicas empregadas na implementação de sistemas criticos, tudo isso em um cenário onde demandava-se cada vez mais softwares de média e alta complexidade.

O cenário problemático da época trouxera efeitos indesejáveis em todos os âmbitos do empreendimento de se construir software. A complexidade crescia ao mesmo tempo que a imaturidade técnico-metodológica firmava-se como fator propulsor da crise (Jr, 1971, p.14). Apresentavam-se como consequências dos maus processos O(A):

- Estouro de prazos previamente estipulados.
- Aumento de custo muito além do planejado.
- Agravamento da inconsistência entre o que era exigido e o que era provido.
- Expansão da ingerenciabilidade de projetos em pouco tempo de manutenção.
- Queda de qualidade rápida da base de código fonte.

Apesar da crise em si nunca ter sido absolutamente resolvida de fato, esforços oriundos da experiência de vários profissionais e pesquisadores competentes resultaram na consolidação de obras técnicas com orientações fundamentais que propõem

soluções eficazes, como: *The Mytical Man-Month, Design Patterns: Elements of Reusable Object-Oriented Software, Structure Programming - Dijkstra, Software Engineering Conference NATO Science Committee* dentre outras obras.

Em meio a vários problemas e desafios enfrentados pela ciência da computação e engenharia de software há o da dependência como ocasionadora do alto acoplamento em aplicações modernas. O software, como solução, surge, da mesma maneira que a pesquisa científica, alicerçando-se sobre outros artefatos de *software* desenvolvidos anteriormente por outros programadores. A tais alicerces comumente nomina-se: o módulos, as bibliotecas e os *frameworks* (IEEE; all, 2010).

Os artefatos alicerçantes do *software* nascem com objetivo de prover funcionalidades genéricas que acabam, por vezes, provocando situações em que lógica de negócio, principal parte da nova aplicação em desenvolvimento, mistura-se com as soluções por eles providos implicando, assim, em um acoplamento potêncialmente perigoso e tóxico às ações de manutenção, extensão e testes (Evans, 2004; MARTIN, 2017).

Vários autores já analisaram a problemática em prol de extender o estado da arte no âmbito do *software* e seus alicerces indispensáveis. Buscam, eles, metodologias ótimas que equilibrem os antônimos omnipresentes na tecnologia da informação: dependência/independência.

Tais análises partem das experiências de seus autores e devem, no esforço acadêmico, serem experimentadas concomitantemente com ambientes possuidores de variáveis diferentes na forma de tecnlogia e contexto.

1.1 Contextualização e Problema

A engenharia de software e ciência da computação reforçam a importância do emprego de boas práticas na criação de *software* no contexto atual onde sistemas coordenam e aprimoram processos informacionais da sociedade humana moderna.

Entretanto, o uso do ágil como bala de prata para geração de valor em tempo recorde em conjunto com florecimento de modelos de inteligência artificial generativa, supostamente capazes de substituir o raciocínio humano sem problemas e com baixo custo, nesta terceira década estão abrindo caminho para o renascimento da crise do software sobre uma nova face.

Vê-se o uso incauteloso das mais diversas ferramentas nas formas de bibliotecas e *frameworks* que fornecem meios rápidos de agregar funcionalidade às aplicações. Todavia, sem mínimo compromisso de atenção com os problemas futuros que a dependência pode infligir sobre projetos cujo tempo de vida útil pode ser importante.

Com vistas ao contexto e seu problema, formula-se duas questões:

Sistemas independentes e desacoplados de suas dependências são possíveis ?

Quais são os beneficios e problemas que recaem sobre projetos que assumem um caminho de independência estrita ?

1.2 Objetivos

O objetivo geral deste trabalho é determinar os efeitos causados pelo emprego de técnicas, padrões e ou filosofias que permitem que o *software* seja mais independente de objetos, pacotes, frameworks e tecnologias externas necessárias a sua utilidade existencial. Tudo isso em observância do que o estado da arte provê como métodos ótimos.

Seus objetivos especificos são:

- Consultar trechos de obras que expoem métodos que reduzem acoplamento e dependência em sistemas.
- Compreender modelagem de sistemas com regras de negócios independentes.
- Compreender benefícios e adversidades mais seus custos.

1.3 Justificativa

Este trabalho justifica-se como sendo, no âmbito acadêmico, um esforço experimental para incremento no número de obras que apresentam o emprego prático-experimental de metodologias, técnicas e ou filosofias desenvolvidas para solucionar ou mitigar problemas que assolam o desenvolvimento software.

Quanto ao âmbito pessoal/profissional, este trabalho é uma oportunidade de aprimorar os conjuntos de conhecimentos técnico-arquiteturais do docente, permitindo-lhe um acesso mais seguro ao mercado de trabalho de TI em implementações comercialmente viáveis.

Trata-se do emprego real do conhecimento pré-existente em prol de obter conclusões sobre as limitações e possibilidades proporcionados pela prática.

1.4 Apresentação do Trabalho

A introdução contextualiza brevemente o leitor à história da ciência da computação direcionando-o a um fato trágico dessa ciência: as dependências e seus problemas correlatos.

A fundamentação teórica introduz o leitor aos fundamentos do paradigma de programação utilizado no projeto experimental e, em seguida, a seções indiretas de obras técnicas populares entre programadores introdutoriamente compilando métodos conhecidos e padronizados com potencial para resolver ou mitigar os desafios correlatos às dependências.

A metodologia apresenta quais procedimentos padrões foram adotados no projeto. Em outras palavras, detalhar-se-á: recursos empregados, uso do tempo, execução do processo, arquiteturas, padrões e princípios filosóficos. Todos aqueles de fato utilizados.

O Desenvolvimento entrará em detalhes do domínio de negócio presente no cerne da aplicação objeto ao mesmo tempo em que descreverá o emprego real de métodos que propoem resolver ou mitigar a alta dependência externa.

Os resultados apontarão ocorrências e usos do que fora apresentado no referencial teórico. Foca-se na descrição do processo.

A conclusão ou considerações finais, advindos da experiência obtida pela execução do projeto, apresentar-se-ão na forma de apontamentos dos beneficios e problemas oriundos da prática construtiva descrita em metodologia.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta uma breve revisão das obras mais importantes de autores que tratam acerca dos métodos capazes de mitigar os efeitos adversos presentes em práticas, em padrões e arquiteturas, que mostraram-se danosas à qualidade do *software* sob o paradigma de programação orientada a objetos Programação Orientada a Objetos (OOP).

Antes de avançar sobre conceitos arquiteturais e de padrões de projetos, deve-se, primeiro, observar aos fundamentos intrínsecos do paradigma sobre o qual dispõe-se a trabalhar sobre.

O conceito de paradigma é introduzido por Floyd (2007) como sendo: "um padrão, um exemplo com o qual as coisas são feitas". O mesmo autor deixara claro, ao discutir acerca dos paradigmas de sua época, que o conceito, no âmbito do desenvolvimento de software, consiste na forma como programas são feitos.

Os paradigmas surgiram concomitantemente com o desenvolvimento de linguagens de programação, em especial nas de alto nível que abstraiam a implementação binária direta de instruções, possibilitando uma implementação mais humana e menos complexa (Sammet, 1969, p. 8-).

Inicialmente houve o surgimento, com o desenvolvimento da arquitetura de Von Neumann, do primeiro paradigma, o Imperativo que trazia os conceitos fundamentais de estado e ação modificante do estado. Sua influência é, até hoje, enorme e serviu de base para paradigmas posteriores (Jungthon; Goulart, 2009, p. 1).

Com o aumento de complexidade dos sistemas, surge o paradigma estruturado que definia a sequência, iteração e decisão como sendo as partes fundamentais de qualquer programa implementável (Dahl; Dijkstra; Hoare, 1972).

Por fim, a orientação a objetos, inicialmente implantadas nas linguagens Simula 1962 e Smalltalk 1972, traz os conceitos de objeto como uma abstração de qualquer elemento da vida real que interage separadamente com outros por meio de "mensagens"(Rentsch, 1982, p. 52-55).

O paradigma orientado a objetos originou, de acordo com a síntese de outros autores presente em Kasture e Jaiswal (2019), os quatros pilares fundamentais sobre da implementação OOP definidos como:

- Abstração capacidade de representar um subconjunto de atributos e comportamentos de uma entidade real
- Encapsulamento controle de acesso externo a atributos e comportamentos privados de um objeto
- Herança capacidade de extensão por superconjunto de atributos e comportamentos
- Polimorfismo capacidade de mutação de comportamentos por sobrescrita

O paradigma OOP adjunto de seus fundamentais pilares possibilitou o desenvolvimento de princípios e padrões de projeto. Ambos sendo peças metodológicas

possuidoras de características relativamente periódicas que extendem os quatro pilares fundamentais possibilitando solução ou mitigação de problemas existentes no desenho OOP (Enrich HELM Richard, 1995, p. 19-20).

Por sua vez, os princípios, mais abragentes e abstratos, influênciam um contexto mais fundamental e abstrato. Embora não constituam obrigação são, de fato, recomendações cuja credibilidade fundamenta-se na experiência de profissionais de longa data, aplicabilidade ampla em inúmeras tecnologias implementacionais, consequências relativamente previsíveis.

O SOLID, produto da análise de MARTIN (2017, et all.) trata, em cinco pontos interconexos, acerca da forma de se modelar um abstrato conjunto de classes. A responsabilidade única da classe com entidade, priorização da ação de extender sobre a ação de modificar, preservação da relação de substituição entre classe base e derivada em herança e a inversão de dependências por contratos abstratos.

O *Object Calisthenics* de Bay (2008) trás sete caminhos pelos quais um sistema conforma-se-á mais com os quatro pilares fundamentais do paradigma OOP.

Os padrões, responsáveis por tratarem a forma como as classes/objetos interagem, possuem quatro propriedades fundamentais que os definem: Um nome padronizado, um problema comum a resolver, a especificação geral da solução que resolve ou mitiga o problema alvo e as consequências positivas e negativas de adotá-lo (Enrich HELM Richard, 1995, p. 19 et al.).

Como exemplo, o padrão *Adapter* constitui-se como um objeto cujos os métodos são capazes de traduzir chamadas entre código cliente e o código servidor.

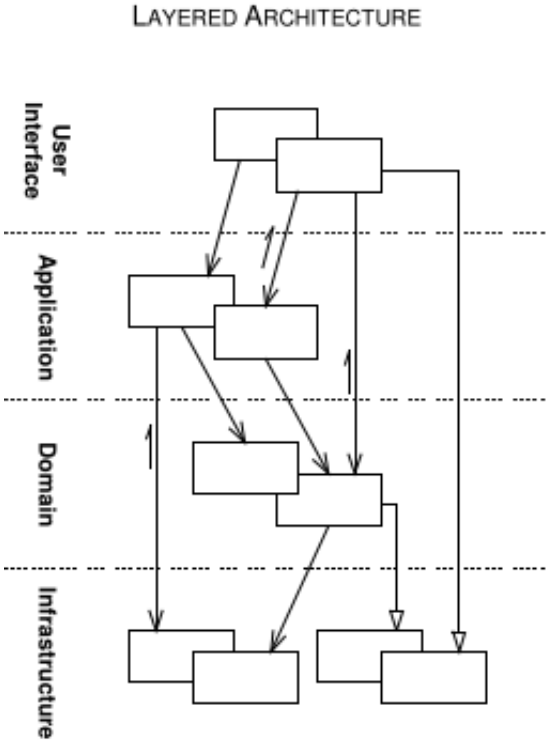
No âmbito das arquitetura de sistemas houve o aprimoramento contínuo de conceitos que promovem desacoplamento por meio da separação de responsabilidades. O objetivo geral constituiu-se das seguintes propriedades ideais desejadas, como apontado por (MARTIN, 2017, p. 202):

- Arquitetura que independe do frameworks para funcionar.
- Regras de negócio podem ser testadas sem interferência de camadas superiores ou inferiores.
- Idependência de Interface de Usuário onde ela pode mudar sem interferir o resto do sistema.
- Idependência de banco de dados onde sua mudança não interfere nas regras de negócio.
- Independência total das regras de negócio.

Arquiteturalmente houve o desenvolvimento initerrupto de conceitos promovedores do separação de responsabilidades principalmente em arquiteturas baseadas em camadas. Chegou-se à conclusão universal de que todo o software possui, em seu código fonte, dois elementos distinguíveis categoricamente. Cada autor o definiu de formas levemente diferentes.

Evans (2004, p. 51) refere-se ao domínio em separação das demais camadas satélites que permitem a funcionalidade de tal domínio ao usuário final.

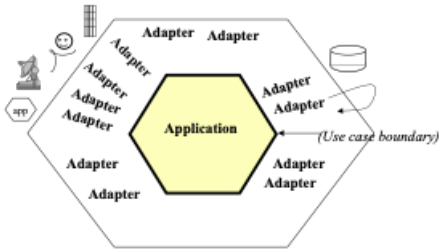
Figura 1 – Modelo de Camadas Notáveis em Sistemas de Acordo com DDD.



Fonte: Erick Evans 2004 2026

Semelhantemente, Cockburn (2005), em seu artigo original, define um padrão majoritariamente comoposto de classes adaptadoras que possibilitam a separação do que ele chamou de "aplicação"do *outside world* "mundo externo".

Figura 2 – Modelo Hexagonal com Classes Adaptadoras Interligando Domínio e Dependências.



Fonte: Alistair Cockburn 2005 2026

Mais a frente MARTIN (2017, p. 189) define categorias separadas em *business rules* "regras de negócio"e demais partes satélites que possibilitam que tais regras sejam úteis na forma de uma aplicação utilizável.

Em conclusão sintética, tais autores trataram acerca de um mesmo problema que possui um conjunto de caracteristicas semelhantes e satélites do conceito de

dependência. O *software* constroi-se envolto em um meio que o permite ser útil ao usuário final. O meio, de forma genérica, possuidor das seguintes caracteristicas:

- Interfaces gráficas.
- Persistência da informação (bancos de dados).
- Comunicação entre computadores em redes TCP/IP.
- Segurança da informação por criptografia e hashing.
- Especificidades da infraestrutura em que a aplicação executa.
- Manipulação de dispositivos computacionais de entrada ou saída.

Evans (2004, p. 52) aponta que o *software* pode, em processos de desenvolvimento descuidados, espalhar as regras do domínio de negócios sobre algumas ou todas partes sobrelistadas.

Apesar todas as visões apresentadas dos parágrafos superiores aparentarem ser ótimas, há, de fato, um criticismo na forma pela qual tais orientações evoluem em complexidade nos projetos acarretando em problemas de complexidade e excesso de engenharia como causa e o desenvolvimento menos ágil em função da maior massividade de código fonte como consequência (RedGreenwp, 2015).

3 METODOLOGIA

detalhar processo que desenvolvimento executará

Este trabalho, em sua essência experimental, desenvolver-se-á ao entorno de um projeto real, aqui definido como objeto prático. Sua construção se orienta pela metodologia é aqui prescrita em função dos conhecimentos úteis da engenharia e arquitetura de *software* fundamental.

Todo software surgirá, de acordo com Roger (2021, p. 244), em conformidade com o processo fundamental de levantamento de requisitos. Ele define esse processo como sendo o início fundamental de todo esforço prático de engenharia de software.

As partes envolvidas nesta etapa são, no geral: clientes, funcionários, gerentes, administradores e, nos casos em que o sistema já é usado no estado em que se encontra, usuários. Todos eles contribuem na consolidação estável de um conjunto de caracteristicas que o sistema deve possuir para ser informacionalmente útil ao empreendimento.

O conjunto de requisitos feito a partir da comunicação constante guia todas as etapas posteriores.

Seguindo a etapa prima metodológica comunicativa, Roger (2021) define mais outras quatro atividades fundamentais a qualquer projeto ativamente desenvolvido, são elas:

- Planejamento - organiza-se como usar recursos durante tempo eficientemente.
- Modelagem modela-se/arquitetur-se como o sistema final estruturar-se-á.
- Construção implementa-se/escreve-se o sistema.
- Entrega Testes, implantação, monitoramento, administração e preparação para proxima iteração.

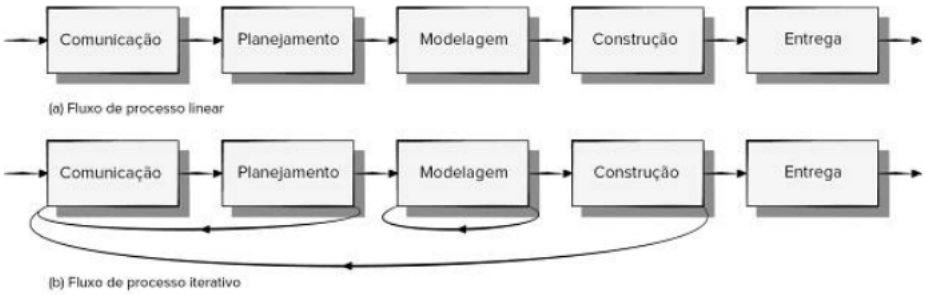
Essas etapas foram, de modo a não tornar complexo o trabalho, incorporadas, em sua forma inalterada, às atividades fundamentais do objeto prático deste trabalho.

Além de um conjunto bem definido e aparentemente linear de tarefas, uma organização temporal própria também deve ser adotada. Destacou-se, para a finalidade deste trabalho, os seguintes ordenamentos temporais:

O objeto prático temporalmente assume, na execução de suas etapas, seguimento semi-linear com capacidade para iteração entre etapas internas ou iteração de todas etapas. Não define-se limites temporais $prazos$ nem estabelecimento de necessidade entrega iterativa de segmentos usáveis a usuários finais. Em outras palavras o projeto desenvolve-se de forma a possuir total conformidade com os requisitos extraidos do ambiente de negócios consultado.

Determinada as etapas e a própria forma como elas são executadas, procede-se para como a estrutura do sistema é implementada. Aqui reside o cerne das questões levantadas em problemática.

Figura 3 – Fluxos de Processos Adotados.



Fonte: Pressman 2021 2026

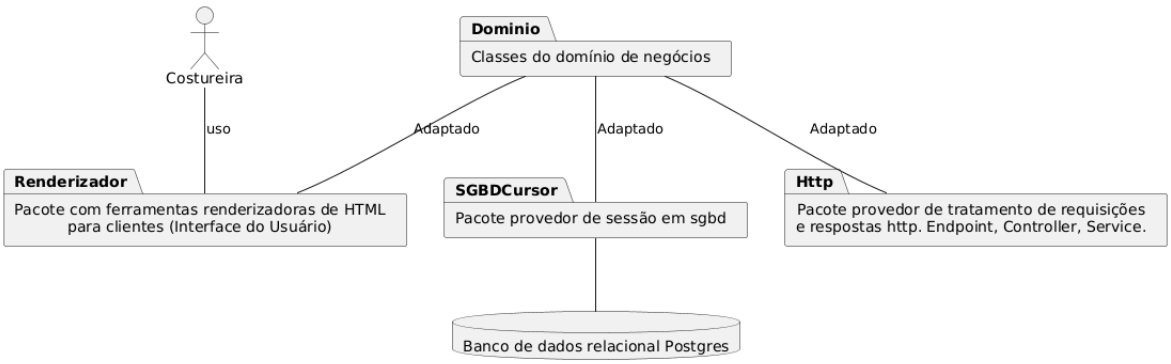
Arquiteturalmente a aplicação adota um modelo tradicional de camadas onde cada camada é responsável por uma responsabilidade definida ainda no levantamento de requisitos.

detalhar

A garantia de separação entre domínio e funcionalidades satélites dar-se por orientação da *Hexagonal Architecture* que faz uso massivo e simples de classes adaptadoras que fornecem uma ponte entre o *software* desenvolvido e seu meio necessário à utilidade.

O modelo acamadado adotado orienta-se, com relação a classificação de objetos, sobre o trabalho de Evans (2004)

Figura 4 – Diagrama Deployment Para Componentes Possuidores de Várias Classes.



Fonte: O Autor 2026

4 DESENVOLVIMENTO DA PESQUISA

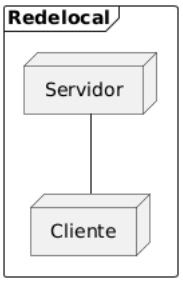
Detalhar prática da execução.

E é a partir do levantamento de requisitos que inicia-se as etapas de desenvolvimento de uma aplicação voltada para micro empreendedoras costureiras.

Partiu-se do pressuposto de que todo empreendimento de costura informal poderiar possuir problemas comuns de ingerência de suas informações. A partir deste pressuposto estabeleceu-se um canal de comunicação coloquial com profissional com anos de experiência de atuação autônoma próxima ao autor.

A comunicação inicial revela situações problemáticas onde: os prazos são esquecidos, o levantamento de custos com insumos usados em confecções e concertos é ignorado e armazenamento irregular de medidas corporais por vezes dificulta consultas rápidas e confunde quando em consultado dados obsoletos.

Figura 5 – Diagrama da Infraestrutura Local de Rede.



Fonte: O Autor 2026

Quanto aos requisitos não funcionais, concluiu-se que sistema atuante em redes locais seria, em materia de confiabilidade, custo e legalidade, mais adequado para estes ambientes de processos simples.

5 RESULTADOS

6 CONCLUSÃO

6.1 Discussão

6.2 Contribuições

6.3 Limitações

6.4 Trabalhos Futuros

REFERÊNCIAS

ALMEIDA, R. B. Evolução dos processadores. *Evolução dos processadores*, unicamp, 1967.

BAY, J. Object calisthenics. In: PROGRAMMERS, T. P. (Ed.). *The Thoughtworks Anthology*. USA: Thoughtworks, Inc, 2008. p. 70–80. ISBN 1-934356-14-X.

COCKBURN, A. *The Hexagonal Ports and Adapters Architecture*. 2005. Disponível em: <https://alistair.cockburn.us/hexagonal-architecture>. Acesso em: 10/11/2025.

DAHL, O. J.; DIJKSTRA, E. W.; HOARE, C. A. R. (Ed.). *Structured programming*. GBR: Academic Press Ltd., 1972. ISBN 0122005503.

DODIG-CRNKOVIC, G. History of computer science. *Västerås: Mälardalen University*, 2001.

ENRICH HELM RICHARD, J. R. V. J. G. *Design patterns - elements of reusable object-oriented software*. [S.l.: s.n.], 1995. ISBN 0201633612.

EVANS, E. *Domain-driven design: tackling complexity in the heart of software*. [S.l.]: Addison-Wesley Professional, 2004.

FLOYD, R. W. The paradigms of programming. In: *ACM Turing award lectures*. [S.l.: s.n.], 2007. p. 1978.

IEEE, S. G.; ALL. *IEEE Standard Glosary of Software Engineering Terminology*. Std96038. New York, NY, USA: Secretary, IEEE Standard Board, 2010. ISBN 978-0-7381-6205-8.

JR, F. P. B. *The mythical man-month: essays on software engineering*. 1st. ed. USA: ADDISON-WESLEY, 1971.

JUNGTHON, G.; GOULART, C. M. Paradigmas de programação. *Monografia (Monografia)—Faculdade de Informática de Taquara, Rio Grande do Sul*, v. 57, 2009.

KASTURE, D.; JAISWAL, R. C. Pillars of object oriented system. *Int. J. Res. Appl. Sci. Eng. Technol.*, v. 7, n. 12, p. 589–590, 2019.

MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. 1st. ed. USA: Prentice Hall Press, 2017. ISBN 0134494164.

NEWELL, A.; PERLIS, A. J.; SIMON, H. A. What is computer science? *Science*, American Association for the Advancement of Science, v. 157, n. 3795, p. 1373–1374, 1967.

REDGREENWP. *Introduction to LaTeX*. 2015. Disponível em: <https://thegreenbar.wordpress.com/2015/12/12/observations-about-clean-hexagonal-architecture/comment-page-1/?utm_source=chatgpt.com>. Acesso em: 7 de janeiro de 2026.

RENTSCH, T. Object oriented programming. *ACM Sigplan Notices*, ACM New York, NY, USA, v. 17, n. 9, p. 51–57, 1982.

ROGER, B. R. M. P. S. *Engenharia de Software uma Abordagem Profissional*. 9. ed. Porto Alegre: AMGH Editora Ltda., 2021. ISBN 9781259872976.

SAMMET, J. E. *Programming Languages: History and Fundamentals*. 1st. ed. Englewood Cliffs, NJ: Prentice-Hall, 1969. ISBN 0137300051.