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 parentes que sempre apoiaram minha formação tecnológica.

“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

www WWWWorld Wide Web

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

Em meio a vários problemas e desafios enfrentados pela ciência da computação e engenharia de software, encontramos 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).

Neste sentido, 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, onde mistura-se com as soluções por eles providos, implicando-se assim em um acoplamento potêncialmente perigoso e tóxico às ações de manutenção, extensão e testes (Evans, 2004; MARTIN, 2017).

Diante desses fatos, vários autores já analisaram a problemática em prol de extender o estado da arte no âmbito do *software* e seus alicerces indispensáveis. Onde se buscou metodologias eficazes que equilibraram os antônimos omnipresentes na tecnologia da informação $dependncia/independncia$.

Partindo dessa premissa, vê-se que diferentes autores, buscaram um ambiente propicio para experimentos, usando de variáveis diferentes na forma de tecnologia e contexto.

1.1 Contextualização e Problema

A engenharia de software,bem como a ciência da computação, reforçam a importância do emprego de boas práticas na criação de sistemas no contexto moderno, onde eles coordenam e aprimoram processos informacionais da sociedade humana.

Entretanto, a tendência dos negócios de tecnologia a adotar metodologias ágeis como solução ótima em todos contextos adjunta ao surgimento recente da inteligência artificial nesta terceira década estarão abrindo caminho para o re-aumento em aplitude da crise do software sob novas características.

Dentre os problemas nascidos desta nova crise, vê-se o uso incauteloso das mais diversas ferramentas existentes nas formas de bibliotecas e *frameworks*. A pricípio, tais ferramentas úteis agregação rápida de funcionalidades nas aplicações agora consolidam-se como fator propulsor da queda de qualidade e obsolescência em sistemas com anos de manutenção.

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

1.2.1 Objetivos gerais

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 em métodos e técnicas.

1.2.2 Objetivos especificos

- 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 intrísecos.

1.3 Justificativa

Justifica-se o trabalho, pro ser 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.

No tocante ao âmbito profissional, justifica-se por ser uma oportunidade pessoal de aprimorar os conjuntos de conhecimentos técnico-arquiteturais do docente, permitindo-lhe um acesso mais seguro ao mercado de trabalho de TI.

Diante dessa realização, 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 ora contexutalizada, brevemente levará o leitor à história da ciência da computação direcionando-o a um fato trágico dessa ciência: as dependências e seus problemas correlatos.

No tocante à fundamentação teórica, ela induzirá 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.

No que se refere a metodologia apresentada, buscaremos o procedimento padrão, no qual foram adotados ao 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.

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

Diante do exposto esperamos que os resultados apontem fatos e descobertas ocorridas durante as atividades abrindo espaço para elaboração das conclusões.

Neste sentido a concluimos que o respectivo estudo tragam reflexão a luz de obras do referencial teórico.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção, apresenta uma breve compilação das obras mais importantes de autores, nos quais direcionam acerca dos métodos capazes de mitigar os efeitos adversos, presentes nas más práticas, que mostraram-se danosas à qualidade do *software* sob o paradigma de programação orientação a objetos Programação Orientada a Objetos (OOP).

Neste sentido, 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). No tocante a esse paradigma, ele teve sua origem com síntese de outros autores presente em Kasture e Jaiswal (2019), nos 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

Notoriamente, ao definirmos esses pilares, percebemos que eles possibilitaram o desenvolvimento de princípios e padrões de projeto ao longo do tempo de aprimoramento técnico do paradigma. Logo, como padrão de projeto entende-se uma espécie

de peça metodológica de característica regular, flexível e aplicável em múltiplos projetos em prol de solucionar problemas recorrentes da OOP (Enrich HELM Richard, 1995, p. 19-20).

Por sua vez, os princípios, mais abragentes e abstratos, influênciam um contexto mais amplo. Embora não constituam obrigação, eles são recomendações cuja credibilidade funcional, fundamenta-se na experiência de profissionais que trabalham por décadas sobre esse paradigma. Nesse sentido a aplicabilidade dos princípios se baseia na: universalidade, idependência de tecnologia implementacional e previsibilidade de consequências.

Conseguinte aos princípios, apontamos o SOLID, onde MARTIN (2017) classifica em cinco subprincípios interconectados, tais como: A responsabilidade única da classe como entidade; priorização da ação de extender sobre a ação de modificar; preservação da relação de substituição entre classe base e derivada na herança; A segregação de interfaces como forma de evitar construções desnecessárias e, por último, a inversão de dependências por contratos abstratos pré-implementados.

Por outro lado, Bay (2008), em *Object Calisthenics* enumera sete pontos pelos quais um sistema conformar-se-á mais com os quatro pilares fundamentais do paradigma, onde para o autor surgirá como uma manifestação em resposta à má qualidade do código OOP pré-existente.

No tocante aos padrões de projeto, eles são responsáveis por tratarem a forma como as classes/objetos interagem da mesma forma que possuem quatro propriedades fundamentais: 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.).

Neste sentido, citamos o padrão *Adapter* que, a princípio, constitui-se como um objeto cujos os métodos são capazes de traduzir chamadas entre código cliente e o código servidor.

No que tange as arquiteturas de sistemas, apontamos que houve o aprimoramento contínuo de conceitos que promovem desacoplamento por meio da separação de responsabilidades. Neste sentido, o objetivo geral se forma das seguintes propriedades ideais desejadas, como apontado por (MARTIN, 2017, p. 202):

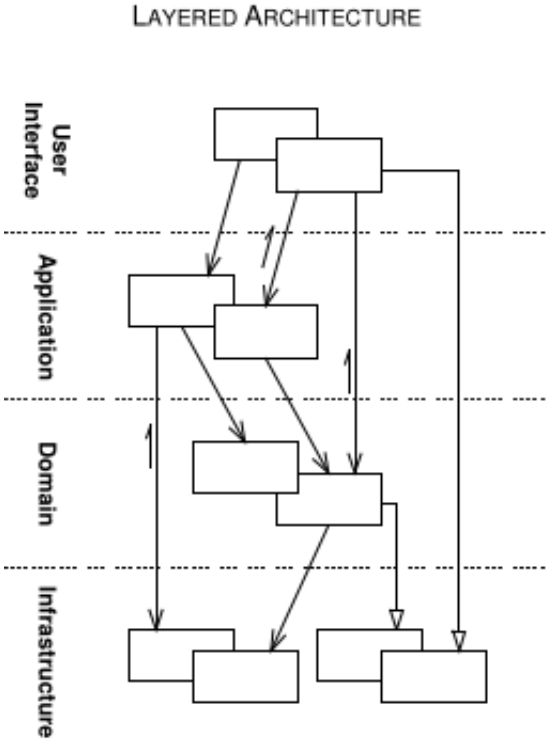
- Arquitetura que independe do frameworks para funcionar.
- Regras de negócio podem ser testadas sem interferência de camadas superiores ou inferiores.
- Idependência de Interface de Usuário onde ela pode mudar sem interferir o resto do sistema.
- Idependência de banco de dados onde sua mudança não interfere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

Notadamente o somatório dessas propriedades,levam ao desenvolvimento initerrupto de conceitos, que são promovedores da separação de responsabilidades,

principalmente em arquiteturas baseadas em camadas. Diante do exposto, chegou-se à conclusão universal de que todo o software possui, em seu código fonte, dois elementos distinguíveis categoricamente, onde cada autor o definiu de formas levemente diferentes.

Evans (2004, p. 51) enuncia que o domínio da aplicação separa-se das demais camadas que provêem sua funcionalidade..

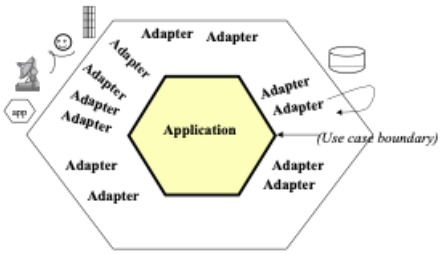
Figura 1 – Modelo de Camadas Notáveis em Sistemas de Acordo com DDD.



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.

Neste sentido, seguindo a lógica desses autores, apontamos que eles tratam acerca de um mesmo problema, que possui um conjunto de caracteristicas semelhantes e próximas do conceito de dependência. O *software* constroi-se envolto em um meio que o permite ser útil ao usuário final possuidor das seguintes propriedades

- 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.

Diante do exposto, como forma de fortalecer estes pontos, 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 dos conceitos apresentados dos parágrafos superiores aparentarem ser ótimas, existe, de fato, um criticismo na forma pela qual tais orientações podem evoluir 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

A presente seção tem o intuito de apresentar quais métodos, dentre os mais diversos presentes em obras técnicas, foram selecionados para possibilitar a definição de um processo funcional consolidador do objeto experimental deste trabalho.

Assim, ao direcionar-se como objetivo nuclear a independência estrita do software para com suas dependências, em observância do que se foi definido como meio para atingir tal estado nas obras de grandes autores, parte-se, então, para uma proposta metodológica simples porem efetiva.

A principio todo *software* surgirá, de acordo com Roger (2021, p.244), em conformidade com o processo de levantamento dos requisitos funcionais. Dessa maneira, ele aponta esta etapa como sendo o marco inicial de todo esforço de engenharia de sistemas que decorre-se sobre uma comunicação constante de partes interessadas.

Indo além, caracteriza-se como partes interessadas: clientes; funcionários; gerentes; administradores e, nos casos em que o sistema já é usado no estado em que se encontra, usuários. Todos eles contribuintes de um processo que resultará, em situações ideais, num sistema possuidor de total utilidade ao empreendimento para o qual fora desenvolvido.

Dessa forma, partindo desta primeira etapa metodológica comunicativa Roger (2021) pontua mais outras quatro atividades fundamentais a qualquer projeto ativamente desenvolvido.

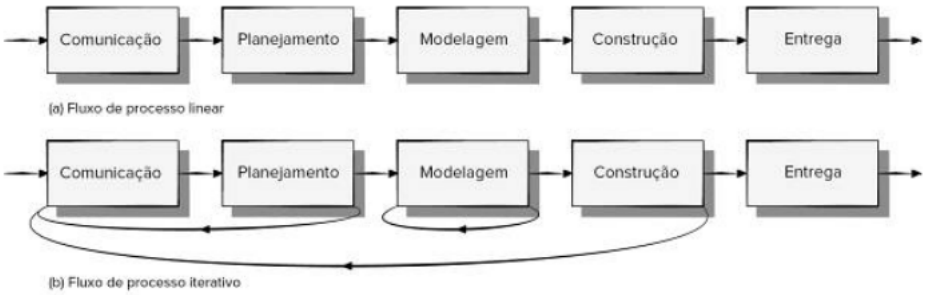
- Planejamento - Organiza-se como usar recursos durante tempo eficientemente.
- Modelagem - Modela-se/arquitetur-se como o sistema final estruturar-se-á.
- Construção - Implementa-se/escreve-se o sistema.
- Entrega - Testes, implantação, monitoramento, administração e preparação para proxima iteração.

Ademais, partir de um conjunto bem definido e aparentemente linear de tarefas não é suficiente pois uma organização temporal própria também deve ser admitida, tendo em visto que todo esforço técnico pode se desenrolar em finitas organizações temporalmente válidas. Destaca-se, de modo a evitar a complexidade, os fluxos de processos apontados pelo mesmo autor.

Em conclusão, admitiu-se uma flexibilização na execução de etapas mais próxima de uma iteração entre etapas e ou de todas etapas. Logo, prazos formais não foram admitidos pelo fato de que a aplicação provavelente não será empregada para uso concreto.

Com um processo de etapas e fluxo bem definido estabelecido segue-se, dentro deste modelo, para a etapa de planejamento e modelagem onde macro características significativas em impacto por todo o tempo de vida útil do *software* serão inreversivelmente assumidas.

Figura 3 – Fluxos de Processos Apropriadamente Simples.



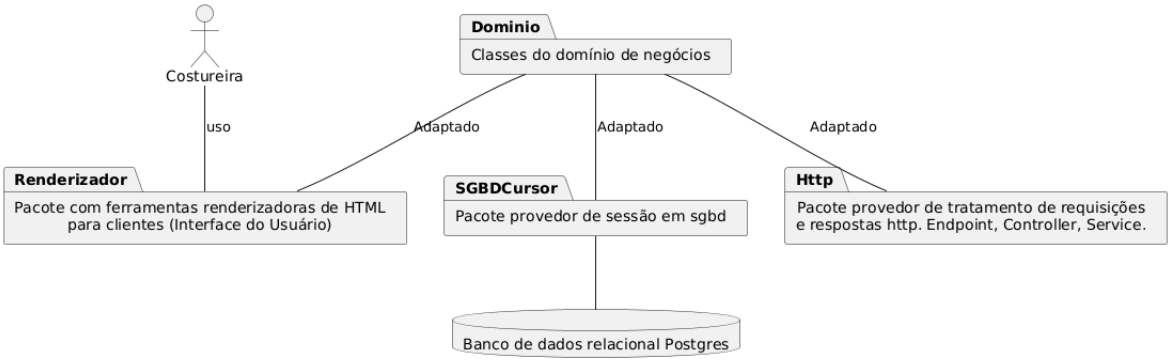
Fonte: Pressman 2021 2026

Por conseguinte, o sistema, arquiteturalmente adota uma estruturação usualmente conhecida sob o título de *Hexagonal Architecture* onde os objetos mais importantes da aplicação, denominados de objetos de domínio de negócio, encontram-se isolados dos demais objetos do ambiente referidos como *outside world* "mundo externo"(Cockburn, 2005).

Por fim, a garantia de separação entre domínio e funcionalidades utilitárias dar-se-á por orientação da arquitetura sobrecitada que sinaliza o uso de classes pertecentes ao padrão de projeto *Adapter* como meio de isolação.

Em seguida, o sistema, de modo a atender os objetivos, possui classes que intermediam várias características indispensáveis ao funcionamento da aplicação.

Figura 4 – Interação entre Classes Nucleares do Domínio o *Outside World*.



Fonte: O Autor 2026

Uma vez que classes, em sistemas OOP, sempre possuem interelações com suas semelhantes na forma de herança ou composição ocorre que, em meios técnicos, discute-se acerca da necessidade de se priorizar uma sobre a outra. Não obstante, o objeto deste trabalho assume uma abordagem mista onde uma e outra são empregadas simultâneamente de acordo com o que se julga mais adequado e conveniente no contexto.

Em suma, de modo a prosseguir mais proximamente ao domínio de negócios caberá, mais apropiadamente, ao desenvolvimento, tratar acerca dos detalhes do domínio que inspirou a implementação deste experimento.

4 DESENVOLVIMENTO DA PESQUISA

O sistema objeto surge como uma solução para gerência de processos em ateliês de costura. Em síntese, o domínio de negócios estudado é: local; de impacto limitado; informal e executado por uma pessoa.

Em virtude disto, partiu-se do pressuposto de que todo empreendimento de costura informal possui problemas comuns de ingerência de suas informações. Sendo assim, a partir deste pressuposto estabeleceu-se um canal de comunicação coloquial com profissional com anos de experiência de atuação autônoma próxima ao programador.

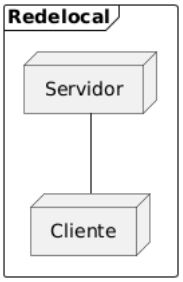
Ao principiar-se no levantamento de requisitos, obteve-se um conjunto de caracteristicas que, em confirmação do que fora supra exposto, um *software* voltado para o domínio de costura e conserto de roupas deveria ter.

Como consequência da comunicação inicial revelou-se situações problemáticas onde: os prazos de serviços são esquecidos senão o próprio serviço e suas características; o levantamento de custos com insumos usados em confecções e concertos não monitorado.

Por fim, ao determinar problemas, segue-se para a escolha de quais tecnologias devem ser empregadas em função do que o contexto apresentou como desafio. A estes atribui-se o título de requisitos não funcionais.

Contudo, em função do orçamento extremamente limitado não houve possibilidade de construção de um sistema seguro o suficiente para operar redes abertas WAN. Logo determinou-se que a aplicação objeto seria de natureza local à rede dos estabelecimentos, abrindo espaço para flexibilizações e simplificações não adequadas ou possíveis a sistemas presentes na WWW (www).

Figura 5 – Diagrama da Infraestrutura Local de Rede.



Fonte: O Autor 2026

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: 10 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.