INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA

DO RIO GRANDE DO NORTE

CAMPUS CURRAIS NOVOS

LUIZ PAULO DE LIMA ARAÚJO

RELATÓRIO SOBRE INDEPENDÊNCIA BÁSICA DE SISTEMA EXPERIMENTAL PARA COM SUAS DEPENDÊNCIAS: ABORDAGEM PRÁTICA SOBRE O PHP

CURRAIS NOVOS - RN

2026

LUIZ PAULO DE LIMA ARAÚJO

RELATÓRIO SOBRE INDEPENDÊNCIA BÁSICA DE SISTEMA EXPERIMENTAL PARA COM SUAS DEPENDÊNCIAS: ABORDAGEM PRÁTICA SOBRE O PHP:

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): Merciosvaldo da Silva Exemplo Merciosvaldo da Silva Exemplo.
Co-orientador(a): .

CURRAIS NOVOS - RN
2026

LUIZ PAULO DE LIMA ARAÚJO

Relatório Sobre Independência Básica de Sistema Experimental para Com Suas Dependências: Abordagem prática sobre o PHP

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

Merciosvaldo da Silva Exemplo
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

Os computadores emergem no século XX como máquinas que possibilitaram a evolução da forma como se lida com a informação gerada nas sociedades humanas modernas. O aumento vertiginoso do poder computacional permitiu a consolidação de sistemas cada vez mais complexos o que implicou na mudança na forma como o software é escrito. De programas inteiramente concebidos por um programador para grandes sistemas mantidos por vários técnicos cujo os alicerçes nascem externamente na forma de módulos e pacotes. Entretanto, a agilidade propiciada pela modularidade trouxe consigo o possível uso incauteloso das dependências que o fazem ser funcional implicando e problemas que acomulam-se ao longo do tempo. Partindo disso , criou-se uma aplicação experimental orientada a objetos voltada para ateliês de costura utilizando a arqutetura hexagonal com classes adaptadoras como forma de inverter dependências nas partes de comunicação e dados. Por fim, concluiu-se que os esforços em prol da independência entre sistema e dependências é possível como também é dispendioso em tempo e recursos intelectuais ao mesmo tempo que e é capaz de causar o *overengeneering*.

Orientação a Objetos, Gestão de Dependência, Padrões, Arquitetura:

ABSTRACT

Keywords:

LISTA DE FIGURAS

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 World Wide Web

SQL Structured Query Language

API Aplication Programing Interface

HTTP Hipertext Transfer Protocol

UDP User Datagram Protocol

LAN Local Area Network

WAN Wide Area Network

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, encontra-se 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.

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 definir-se esses pilares, percebe-se 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, aponta-se 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, cita-se 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, aponta-se 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):

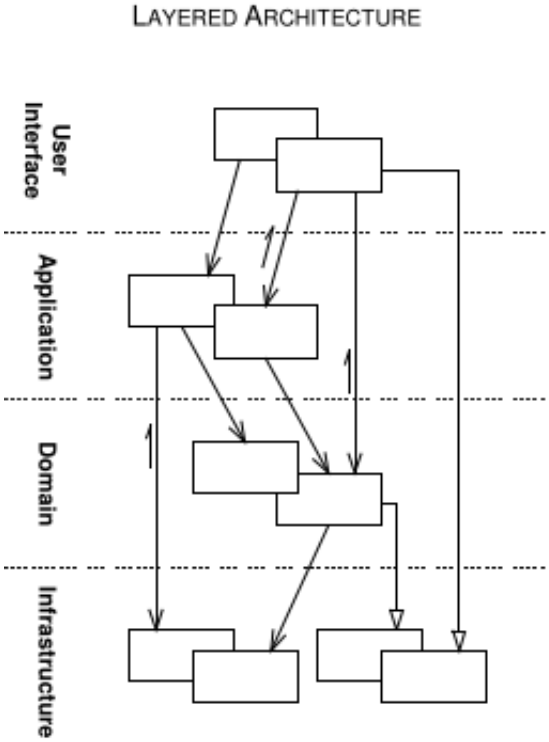
- Arquitetura que independe do frameworks para funcionar.
- Regras de negócio podem ser testadas sem interferência de camadas superiores ou inferiores.
- Idependência de Interface de Usuário onde ela pode mudar sem interferir o resto do sistema.
- Idependência de banco de dados onde sua mudança não interfere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

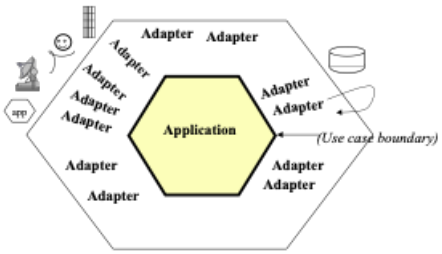
Figura 1 – Modelo de Camadas Notáveis em Sistemas de Acordo com DDD.



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, aponta-se 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 objetivo de apresentar quais métodos presentes em obras técnicas serão selecionados para possibilitar a consolidação de um processo funcional na construção do objeto experimental deste trabalho.

Assim, ao se posicionar 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, conforme citado anteriormente no referencial teórico, 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 estas etapas como sendo 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), Iam (2011) pontuam mais outras quatro atividades fundamentais a qualquer projeto ativamente desenvolvido, onde lista:

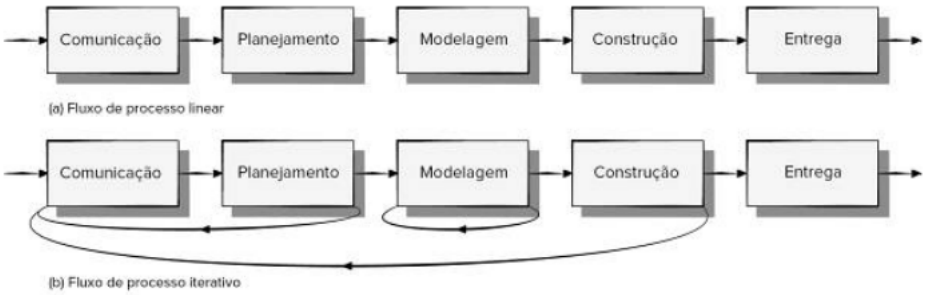
- Planejamento - Organização de tempo e recursos;
- Modelagem - Arquitetação e modelagem da estrutura do sistema;
- Construção - Escrita/Implementação do sistema;
- Entrega - Testes, implantação, monitoramento, administração e preparação para proxima iteração;

Neste sentido, partindo de um conjunto bem definido e aparentemente linear de tarefas, encontramos que elas não são sequencialmente ótimas, pois apresentam uma organização temporal muito simplista. Logo, tendo em vista que todo esforço técnico pode se desenrolar em finitas organizações temporalmente válidas, os fluxos de processos apontados pelo mesmo autor dinamizar-se-ão como demonstrado na próxima figura.

Diante dessa premissa, admitiu-se uma flexibilização na execução de etapas mais próxima de uma iteração entre elas e ou de todas elas (evolutivo). Em relação aos prazos formais não foram admitidos pelo fato de que a aplicação provavelente não será utilizada de fato.

Consequentemente além das etapas iniciais de comunicação e planejamento, dentro do modelo de etapas e fluxo flexível ora apresentado, segue-se iteradamente para modelagem e construção que determinam, sob a orientação de arquiteturas e

Figura 3 – Fluxos de Processos.

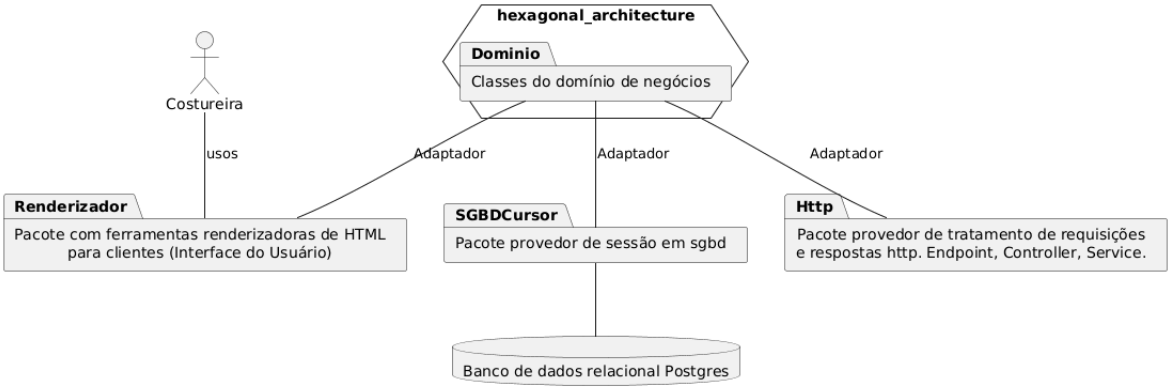


Fonte: Pressman 2021 2026

padrões comuns, características significativas em impacto por todo tempo de vida útil do *software*.

No que diz respeito à arquitetura, assume-se, de modo a evitar a complexidade de abordagens menos intuitivas, o modelo proposto por Cockburn (2005) que aponta o uso do padrão de projeto *Adapter* como forma de isolar os objetos do domínio de negócios dos demais objetos do ambiente referido com *Outside World* "Mundo Externo". A tal arquitetura atribui-se o nome de *Hexagonal Architecture*.

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



Fonte: O Autor 2026

No entanto, exclui-se, de modo a seguir a linha do experimental, a parte desenvolvida que provê interface ao usuário final. Logo resta apenas a escrita do Domínio de negócios, do adaptador de banco de dados e do adaptador para comunicação, onde usar-se-á protocolo da camada de transporte em redes de computadores TCP/IP.

No tocante, ao campo das classes em sistemas OOP, há interelações entre elas na forma de herança ou composição. Entretanto, ocorre que, em meios técnicos, discute-se acerca da necessidade de se priorizar a primeira sobre a segunda (lenon, 2023). 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 ao contexto de implementação.

Quanto a objetos que agregam ou são compostos de N objetos, adotou-se o padrão inicialmente definido por Goldberg (1984). Padrão este que consiste em separar

responsabilidade de gerenciar uma coleção de sub objetos de seu objeto manipulador principal assim evitando violações do princípio da responsabilidade única.

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

4 DESENVOLVIMENTO

O domínio de negócios do objeto surge como uma solução para gerência de processos em ateliês de costura. Em síntese, ele possui características: ser local e causar impacto limitado em ambiente informal.

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

A principiar-se o levantamento de requisitos, se obteve um conjunto de caracteristicas que definiu, em confirmação do que fora presumido, um *software* que deveria ser voltado para o domínio de costura e conserto de roupas.

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

Em seguida, partiu-se para escolha das melhores ferramentas para o contexto desafiador que delineou-se a problemática. Comessou, aqui, a etapa que avalia requisitos/aspectos não funcionais da aplicação.

Neste sentido, Iam (2011, p.85) define os requisitos não funcionais como sendo aqueles que não atribuem funcionalidades à aplicação, que são fundamentais para existência dela. Eles são: o ambiente físico, as questões de segurança da informação, a capacidade física da máquina servidora dentre outros que as circunstâncias possam trazer.

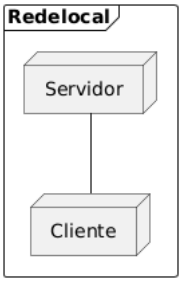
Assim, o ambiente do sistema, de modo a garantir acessibilidade por qualquer rede com acesso a internet, inicialmente fora desenvolvido para ser acessível por redes de internet de amplo alcançe, conhecidas como Wide Area Network (WAN). Entretanto, viu-se que essa caracteristica de disponibilidade ampla não seria viável devido ao alto custo de infraestrutura de computadores em núvem e possíveis problemas legais decorrentes de vazamentos de dados (BRASIL, 2028).

Além disso, considerando o foco experimental assumido, estabeleceu-se o ambiente de funcionamento estrito à redes locais Local Area Network (LAN) cuja disponibilidade fica espacialmente restrita ao estabelecimento. Abordagem esta, semelhante a encontrada em aplicações anteriores à difusão da acessibilidade da internet.

Conseguinte, prosseguiu-se no emprego de uma linguagem de programação interpretada cuja caracteristica mais importante fosse a capacidade de integrar-se bem com o servidor. Logo figurou-se opções como: Javascript; PHP; Python; Ruby; Rust; Go dentre outras. Entretanto, devido ao já largo emprego de Javascript, assumiu-se uma alternativa mais tradicional na linguagem de programação PHP.

O PHP, que nascera como um conjunto de binários escritos na linguagem C, tornou-se, na década de 2000 e 2010, uma das mais utilizadas ferramentas do desenvolvimento *backend* (ThePHPDocumentation,). Apesar de seu constante declínio

Figura 5 – Diagrama da Infraestrutura Local de Rede Simples.



Fonte: O Autor 2026

atual em face de opções mais modernas, sua influência ainda não é negligenciável pois ainda se encontra presente no cerne de inúmeras aplicações legadas e *software* **web!** (**web!**) (Tim, 2025).

Neste sentido, assumimos uma linguagem de programação para a implementação do projeto, adiantando-se para uma modelagem do esquema de classes do domínio de negócios, onde o domínio minunciosamente estudado trás as seguintes classes fundamentais para solução do problema de negócio.

Figura 6 – Diagrama Textual Indentado das Classes Principais.

```
ServicoCostureira
|_Cliente
| |_Contato
| |_Medida
|_ColecaoPecas
  |_Pecas
    |_ColecaoInsumos
      |_Insumos
```

Fonte: O Autor 2026

Complementando o diagrama sobreposto, determina-se que "ServiçoCostureira", elemento pivô, possui relação de composição de clientes que por sua vez agregam classes representantes de seus dados. Por outro lado há classe de Coleção de peças, também parte composta de serviço que, por conseguinte, possui de sua parte uma coleção de insumos e seus respectivos dados.

Em questão de organização de diretórios e sub diretórios, convencionou-se a criação da seguinte hierarquia:

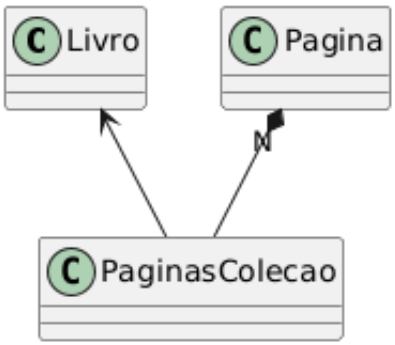
- Collection - Classes que gerenciam plúrimos objetos.
- Entity - Classes que representam Entidades do domínio de negócios.
- Service - Classes que implementam casos de uso do sistema instanciadas em main.
- Repository - Classes que manipulam banco de dados através de interface padronizada.

- Comunication - Classes que usam interface udp/ip para comunicação em redes "*project's specific*".
- Enum - Enumerações que representam estados evitando strings mágicas.
- ValueObject - Objetos de valor que não são omnirelevantes ao domínio

Tal organização orientou-se através de Goldberg (1984), Evans (2004), **??**)

No tocante às classes coleções, a obra Goldberg (1984) foi de extrema importância na orientação cautelosa de se evitar que objetos como peças e insumos possuissem excesso de responsabilidades em manipulando matrizes de subobjetos agregados.

Figura 7 – Classes Coleção.



Fonte: O Autor 2026

Com relação às dependências, que são foco desse experimento, o PHP possui extensões padrões ativáveis na sua configuração, onde cada módulo extende sua capacidade de modo a permitir que a aplicação possa performar ações críticas às funcionalidades instituidas nos requisitos funcionais.

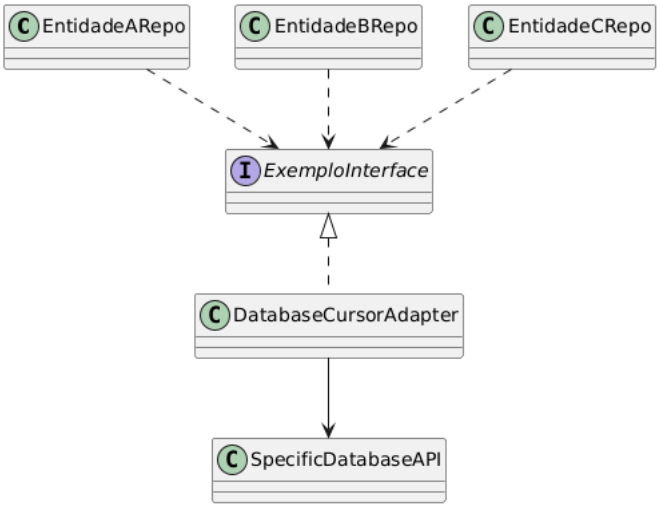
Duas extensões cuja finalidades garatem as capacidades de execução de transações Structured Query Language (SQL) e atendimento do ciclo requisição/resposta foram ativadas, respectivamente sendo elas: pgsql e sockets.

No âmbito dos dados, a ação tomada objetivou a independência da ferramenta que manipula o banco de dados através da criação de uma interface, também conhecida como contrato abstrato, onde definiria que classes forasteiras estaria em conformidade com tal contrato que, a princípio, possuiria quatro métodos fundamentais escritos, sendo eles: ler, escrever, redefinir e encerrar.

Em seguida, escreveu-se uma classe do tipo adaptador que implementa o sobrecitado contrato, onde esse código seria capaz de acessar diretamente a Aplication Programing Interface (API) da extensão pgsl. Ademais, esta abordagem também visou alcançar a simplicidade através de um adaptador que pudesse ser universalmente utilizado por classes do tipo repositório possuidoras de código específico de consultas para instânciação de classes de domínio SQL.

A posteriori, viu-se as ações assumidas, no contexto dos dados, criaram um cenário com características próximas mas não idênticas às expostas por Evans (2004, p.106) na seção que trata acerca de repositórios, conforme o diagrama:

Figura 8 – Adaptador de Banco de Dados é, Por Contrato, Substituível.



Fonte: O Autor 2026

Quanto ao âmbito da comunicação, não utilizou-se protocolo de aplicação em redes conhecido como Hipertext Transfer Protocol (HTTP), como consequência, elaborou-se um protocolo próprio construido sobre o User Datagram Protocol (UDP). No que se refere ao protocolo ora apresentado, trata-se de comandos na forma de substantivo, verbo e dados da ação, conforme exemplificado no protocolo desenvolvido:

Figura 9 – Exemplos de Comandos do Protocolo Atelie.

```
cliente|criar|{"nome":"Maria","medida":{"cintura":70cm}}\n
cliente|remover|{"id":56}\n
cliente|ler|{"nome":"Maria"}\n
cliente|atualizar|{"id":"98","medida":56.3}\n
```

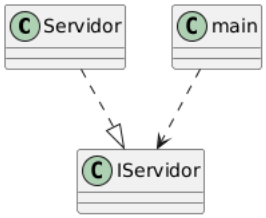
Fonte: O Autor 2026

Partindo desse modelo, vê-se que para cada evento de requisição do cliente, há ação por parte de um servidor, do direcionamento para o tratamento do comando, bem como, para o roteamento em casos de uso e resposta.

Quanto ao módulo dependido para esta funcionalidade, a questão da independência foi garantida através da implementação de uma classe chamada servidor. Sendo ela implementante de uma interface de mesmo nome que define métodos padrões reconhecidos pelo ponto de execução principal "main.php", conforme a diagramação que se segue:

Por fim, conclui-se a etapa de construção, onde a comunicação e os dados são funcionais e relativamente mais independentes entre eles, onde podemos extrair resultados e conclusões sobre os mesmos.

Figura 10 – Relação Servidor, IServidor e Main.



Fonte: O Autor 2026

5 RESULTADOS

A presente seção aponta os resultados práticos da implementação do sistema na forma de facilidades e dificuldades.

Dente estas sub partes, apontamos que as interfaces presentes em *Comunication* e *Repository* permitem que o programador crie classes que as implementam. Tal abordagem, que vai de encontro com o princípio de inversão de dependência presente dos princípios SOLID, que garante que há alguma base para substituição de componentes externos ao domínio sem alterações profundas nos internos.

A abordagem mista e entre herança e composição resultou em um modelo de classes de granularidade facilemente compreensível e gerenciável ao programador, não havendo rigidez excessiva do uso abusivo de herança e nem o caus de inúmeras interfaces sendo criadas para cada relação de composição.

6 CONCLUSÃO

O sistema Atelie constituiu-se como reforço na compreensão e de técnicas, filosofias e arquiteturas por parte do autor. Apesar da sua natureza experimental, suas nuances trouxeram luz às questões levantadas nos objetivos deste trabalho.

Diante dessa premissa é possível implementar software que independe de partes satelites conferintes de funcionalidade. Entretanto, percebeu-se que abordagens que visam tal objetivo consomem mais recursos intelectuais e temporais por parte do técnico responsável, ao mesmo tempo que abrem espaço para o emprego desenfreado de complexidade desnecessária, problema este conhecido como *overengeneering* "excessos de engenharia".

Ademais observou-se também que classes e funções embutidas na linguagem adotada surgem como ponto de inflexão, remedia-las é um esforço desnecessariamente árduo e com capacidade de implicar em gargalos de desempenho, atrasos na entrega de valor e falhas e erros em problemas difíceis de detectar-las.

Por fim, conclui-se que a observância das características do domínio em conjunto com consulta crítica de obras técnicas, abre caminho para consolidação de um projeto tecnicamente equilibrado e capaz de perpassar ao teste do tempo.

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.

BRASIL. *Lei geral de proteção de dados*. 2028. Disponível em: <https://www.planalto.gov.br>. Acesso em: 16 de janeiro de 2026.

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.

GOLDBERG, A. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984. (Addison-Wesley Smalltalk-80 series). ISBN 9780201113723. Disponível em: <https://books.google.com.br/books?id=u7ZQAAAAMAAJ>.

IAM, S. *Software Engineering*. 9. ed. New Jersey: Pearson, 2011. ISBN 9780137035151.

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.

LENON, R. *Composition Over Inheritance: Uma Abordagem "Contra-Terrorista"para Orientação a Objetos*. 2023. Disponível em: <https://medium.com/@lenonrodrigues/composition-over-inheritance-uma-abordagem-contra-terrorista-para-orienta%C3%A7%C3%A3o-a-objetos-9e520eefbfe5>. Acesso em: 16 de janeiro de 2026.

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. *Observations about "cleanhexagonal"architecture*. 2015. Disponível em: <https://thegreenbar.wordpress.com/2015/12/12/observations-about-clean-hexagonal-architecture/comment-page-1/?utm_source=chatgpt.com>. Acesso em: 16 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.

THEPHPDOCUMENTATION. *História do php*. Disponível em: <https://www.php.net/manual/pt_BR/history.php.php>.

TIM, A. *Is PHP declining? JetBrains says yes. And no*. 2025. Disponível em: <https://www.theregister.com/2025/10/21/massive_jetbrains_dev_survey/>. Acesso em: 16 de janeiro de 2026.