Comparação entre Portas na Arquitetura Hexagonal e Interfaces na Arquitetura Limpa: Uma Análise Conceitual e Prática

Nagib Sabbag Filho

Leaders.Tec.Br, 1(8), ISSN: 2966-263X, 2024.

e-mail: profnagib.filho@fiap.com.br

DOI: https://doi.org/10.5281/zenodo.13380201

PermaLink: https://leaders.tec.br/artigo/comparacao-entre-portas-na-arquitetura-hexagonal-e-interfaces-na-arquitetura-limpa-uma-analise-conceitual-e-pratica

Received: 22 Aug 2024 / Accepted: 24 Aug 2024 / Published online: 26 Aug 2024

Abstract:

O artigo explora as semelhanças e diferenças entre dois conceitos centrais em arquiteturas de software populares: as Portas na Arquitetura Hexagonal e as Interfaces na Arquitetura Limpa. A análise aborda os fundamentos teóricos de cada abordagem, destacando como esses conceitos moldam a estrutura e a interação dos componentes de software.

Key words:

arquitetura hexagonal, portas, interfaces, arquitetura limpa, diferenciação, design de software, dependências, comunicação, adaptação, testes, abstração, componentes, sistemas, flexibilidade, desacoplamento, camadas, interações, implementações, padrões de projeto.

Conceitos Fundamentais da Arquitetura de Software

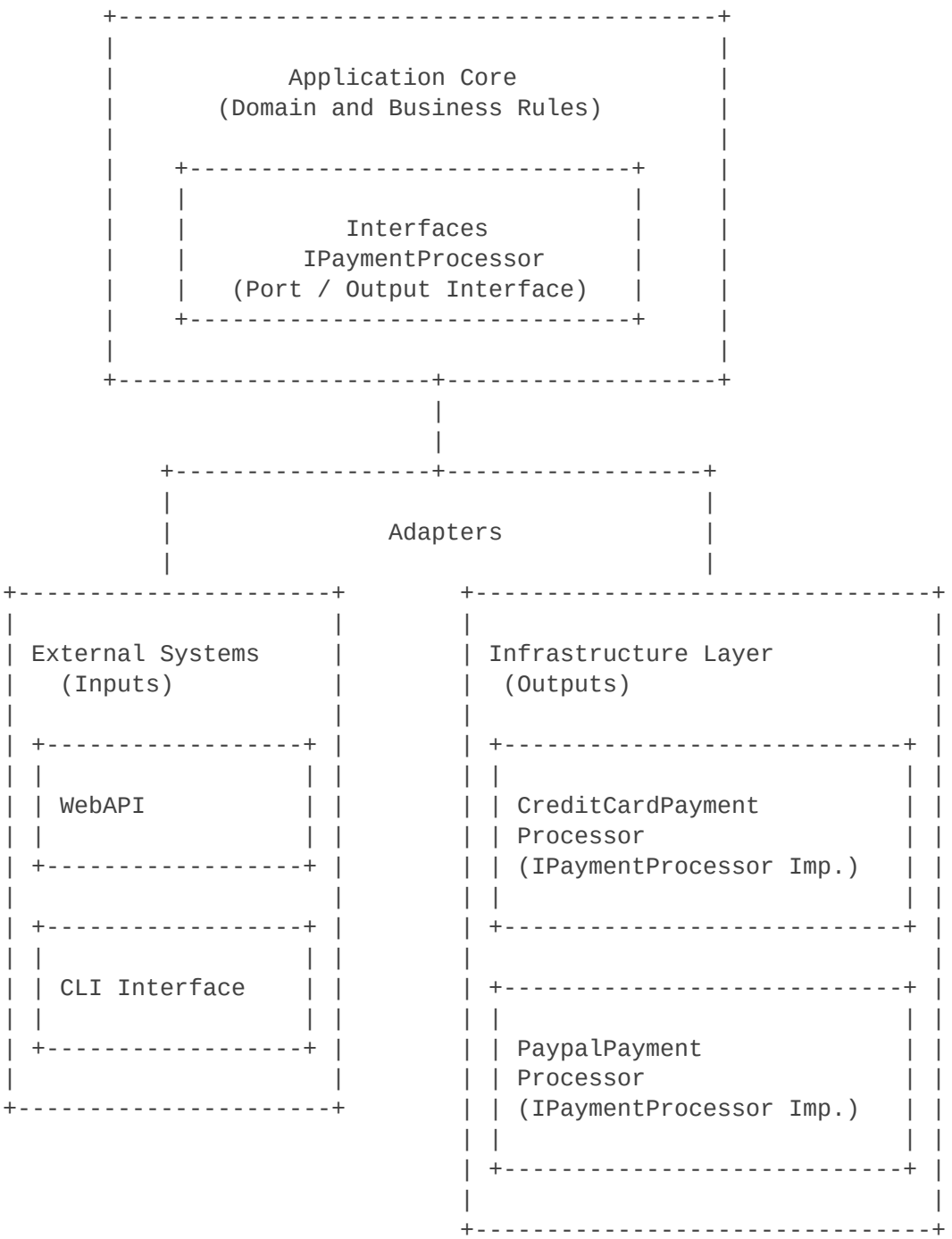
A arquitetura de software desempenha um papel essencial na construção de sistemas complexos e duradouros. Ela define as diretrizes para a organização do código e a interação entre os componentes do sistema, influenciando diretamente na sua escalabilidade, flexibilidade e manutenção. Com o avanço das tecnologias e o crescimento das demandas por sistemas mais robustos, surgiram diversas abordagens arquitetônicas que buscam resolver problemas comuns na engenharia de software. Dentre essas abordagens, destacam-se a Arquitetura Hexagonal e a Arquitetura Limpa, ambas com o objetivo de criar sistemas que sejam independentes de detalhes técnicos, permitindo uma fácil adaptação às mudanças tecnológicas. No entanto, cada uma dessas arquiteturas adota princípios e padrões específicos que as diferenciam em termos de organização e tratamento de dependências externas (MARTIN, 2017).

Arquitetura Hexagonal: Estrutura e Função

A Arquitetura Hexagonal, também conhecida como Arquitetura de Ports and Adapters (Portas e Adaptadores), foi introduzida por Alistair Cockburn como uma solução para isolar o núcleo da aplicação das interações externas. Seu principal objetivo é proteger a lógica de negócio das mudanças nas tecnologias externas, promovendo assim uma arquitetura que seja tanto flexível quanto robusta. Na Arquitetura Hexagonal, o núcleo da aplicação interage com o mundo exterior por meio de "portas", que são interfaces bem definidas. Essas portas permitem que diferentes adaptadores sejam conectados ao núcleo, de forma que a lógica de negócio possa ser executada independentemente das tecnologias externas, como bancos de dados, interfaces de usuário, ou serviços de terceiros (RICHARDS, 2015).

Um exemplo ilustrativo dessa arquitetura pode ser encontrado em sistemas de e-commerce. Nesse contexto, uma aplicação pode definir uma porta que representa a interface para processamento de pagamentos. Essa porta pode ser implementada por vários adaptadores, como um adaptador para pagamentos via cartão de crédito e outro para pagamentos via PayPal. Essa estrutura modular possibilita que a lógica central do sistema permaneça inalterada, mesmo quando novos métodos de pagamento são adicionados. Assim, a Arquitetura Hexagonal promove a evolução contínua do sistema, sem comprometer a estabilidade e a integridade da aplicação.

Design Exemplo - Arquitetura Hexagonal



Application Core - Interfaces

A interface IPaymentProcessor define o contrato que ambos os processadores de pagamento devem seguir:

```
namespace ApplicationCore.Interfaces
{
    public interface IPaymentProcessor
    {
        bool ProcessPayment(decimal amount, string paymentDetails);
    }
}
```

Adapters - External Systems - WebAPI - Controllers

A API Web PaymentController utiliza as implementações de IPaymentProcessor para processar pagamentos tanto com cartão de crédito quanto PayPal:

```
using ApplicationCore.Interfaces;
using Microsoft.AspNetCore.Mvc;

namespace Adapters.ExternalSystems.WebAPI.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class PaymentController : ControllerBase
    {
        private readonly IPaymentProcessor _creditCardPaymentProcessor;
        private readonly IPaymentProcessor _paypalPaymentProcessor;

        public PaymentController(IPaymentProcessor creditCardPaymentProcessor, IPayment
Processor paypalPaymentProcessor)
        {
            _creditCardPaymentProcessor = creditCardPaymentProcessor;
            _paypalPaymentProcessor = paypalPaymentProcessor;
        }

        [HttpPost("credit-card")]
        public IActionResult ProcessCreditCardPayment([FromBody] PaymentRequestModel re
quest)
        {
            var result = _creditCardPaymentProcessor.ProcessPayment(request.Amount, req
uest.PaymentDetails);

            if (result)
                return Ok("Credit card payment processed successfully.");

            return BadRequest("Credit card payment processing failed.");
        }

        [HttpPost("paypal")]
        public IActionResult ProcessPayPalPayment([FromBody] PaymentRequestModel reques
t)
        {
            var result = _paypalPaymentProcessor.ProcessPayment(request.Amount, request
.PaymentDetails);

            if (result)
                return Ok("PayPal payment processed successfully.");
```

```
            return BadRequest("PayPal payment processing failed.");
        }
    }
}
```

Adapters - External Systems - WebAPI - Models

```
namespace Adapters.ExternalSystems.WebAPI.Models
{
    public class PaymentRequestModel
    {
        public decimal Amount { get; set; }
        public string PaymentDetails { get; set; }
    }
}
```

Adapters - Infrastructure Layer - CreditCardPaymentProcessor

A implementação CreditCardPaymentProcessor é responsável por realizar o processamento real dos pagamentos, simulando integrações com sistemas de pagamento externos:

```
using ApplicationCore.Interfaces;
using System;

namespace Adapters.InfrastructureLayer.PaymentProcessors
{
    public class CreditCardPaymentProcessor : IPaymentProcessor
    {
        public bool ProcessPayment(decimal amount, string paymentDetails)
        {
            // Lógica para processar o pagamento com cartão de crédito
            // Exemplo: Integração com um gateway de pagamento externo

            Console.WriteLine($"Processing credit card payment of {amount} with details
 {paymentDetails}.");
            return true; // Simulação de sucesso
        }
    }
}
```

Adapters - Infrastructure Layer - PayPalPaymentProcessor

A implementação PayPalPaymentProcessor é responsável por realizar o processamento do pagamento através do PayPal:

```
using ApplicationCore.Interfaces;
using System;

namespace Adapters.InfrastructureLayer.PaymentProcessors
{
    public class PayPalPaymentProcessor : IPaymentProcessor
    {
        public bool ProcessPayment(decimal amount, string paymentDetails)
        {
```

```
            // Lógica para processar o pagamento via PayPal
            // Exemplo: Integração com a API do PayPal

            Console.WriteLine($"Processing PayPal payment of {amount} with details {pay
mentDetails}.");
            return true; // Simulação de sucesso
        }
    }
}
```

Adapters - External Systems - WebAPI - Services.AddScoped

Configuração da Injeção de Dependência:

```
services.AddControllers();

// Injeção de dependência para os processadores de pagamento
services.AddScoped < IPaymentProcessor, CreditCardPaymentProcessor > (provider =>
            new CreditCardPaymentProcessor());

services.AddScoped < IPaymentProcessor, PayPalPaymentProcessor > (provider =>
            new PayPalPaymentProcessor());
```

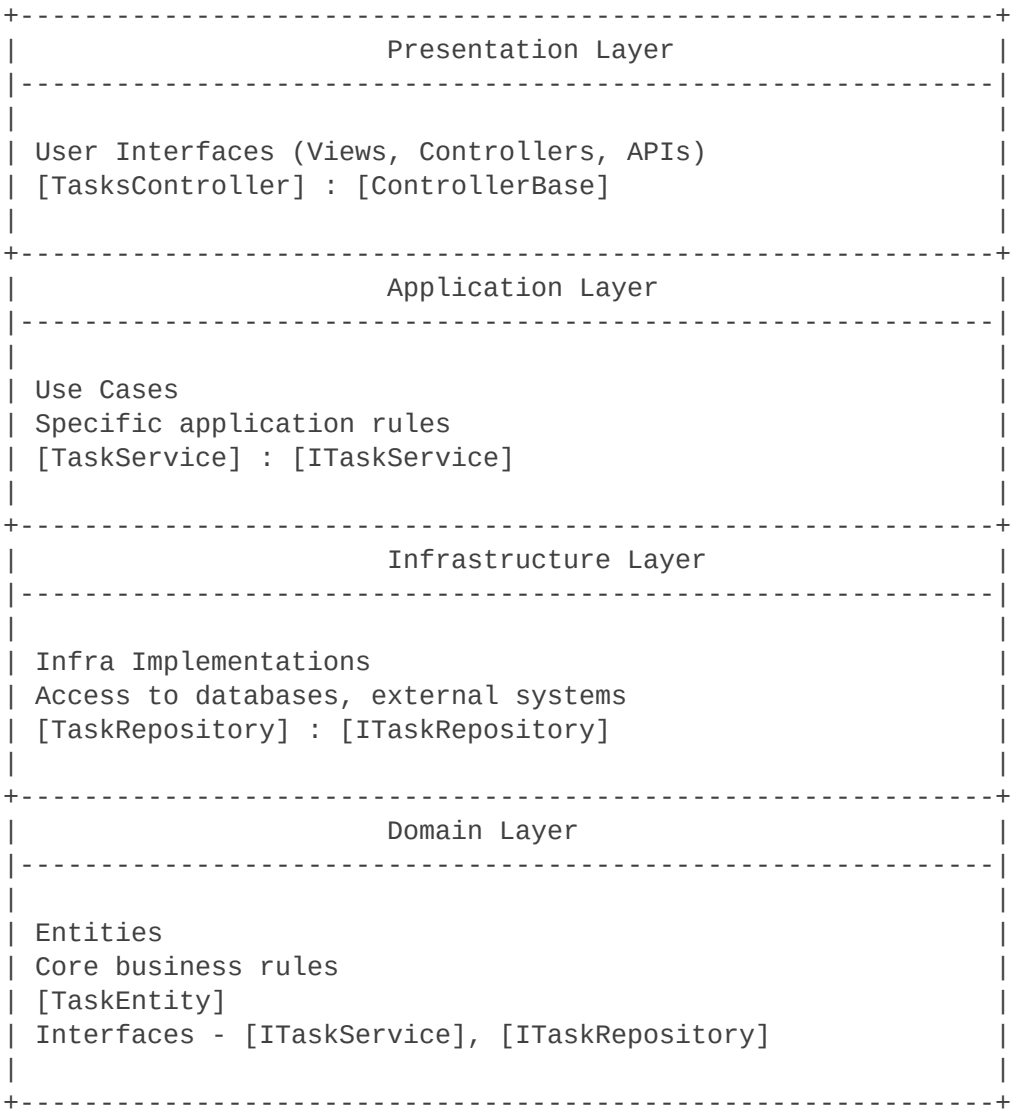
Esse padrão de portas e adaptadores também promove a testabilidade da aplicação, uma vez que a lógica de negócio pode ser testada isoladamente, utilizando implementações mock das portas. Além disso, o uso de portas como ponto de entrada e saída permite uma clara definição de responsabilidades dentro do sistema, resultando em uma arquitetura mais coesa e modular. Assim, a Arquitetura Hexagonal não só facilita a manutenção e a evolução do sistema, mas também promove um design mais limpo e organizado, onde cada componente tem uma responsabilidade bem definida e é facilmente substituível.

Arquitetura Limpa: Interfaces e Organizações

A Arquitetura Limpa, proposta por Robert C. Martin (também conhecido como Uncle Bob), segue princípios semelhantes, mas com uma abordagem diferente em termos de organização. A Arquitetura Limpa enfatiza a separação de preocupações e a independência de frameworks e bibliotecas, de modo que o núcleo da aplicação possa evoluir independentemente das mudanças externas. Essa abordagem organiza o sistema em camadas concêntricas, onde o núcleo de negócio (domínio) está no centro, cercado por camadas de casos de uso, e, por fim, pela camada de interface e infraestrutura. Dessa forma, a lógica de negócio é totalmente isolada das preocupações externas, garantindo que alterações em tecnologias ou frameworks não afetem o núcleo da aplicação (MARTIN, 2017).

Um exemplo típico de aplicação da Arquitetura Limpa pode ser encontrado em sistemas de gerenciamento de tarefas. Neste cenário, o núcleo do sistema é composto por entidades e casos de uso que definem as regras e fluxos de negócio. Por exemplo, uma entidade Tarefa pode conter propriedades como Id, Nome e Estado de Conclusão. Um caso de uso, como Gerenciamento de Tarefas, pode ser responsável por marcar uma tarefa como concluída. As interfaces para interação com o usuário e persistência de dados são colocadas nas camadas exteriores, de modo que a lógica de negócio possa ser testada e desenvolvida de forma independente da interface de usuário ou tecnologia de banco de dados utilizada.

Design Exemplo - Arquitetura Limpa



Presentation Layer

A camada de apresentação lida com a interação do usuário. Pode ser um controlador MVC ou API em ASP.NET Core:

```
// Controller in ASP.NET Core
[ApiController]
[Route("api/[controller]")]
public class TasksController : ControllerBase
{
    private readonly ITaskService _taskService;

    public TasksController(ITaskService taskService)
    {
        _taskService = taskService;
    }

    [HttpPost("create")]
    public IActionResult CreateTask([FromBody] CreateTaskRequest request)
    {
        var result = _taskService.CreateTask(request);
        return Ok(result);
```

```
    }

    [HttpGet("list")]
    public IActionResult ListTasks()
    {
        var tasks = _taskService.GetTasks();
        return Ok(tasks);
    }
}
```

Application Layer

A camada de aplicação lida com a lógica específica da aplicação e as regras de uso:

```
// Application Service
public class TaskService : ITaskService
{
    private readonly ITaskRepository _taskRepository;

    public TaskService(ITaskRepository taskRepository)
    {
        _taskRepository = taskRepository;
    }

    public TaskResult CreateTask(CreateTaskRequest request)
    {
        // Aplicar regras de aplicação específicas
        if (string.IsNullOrEmpty(request.Title))
        {
            throw new ArgumentException("Title cannot be empty.");
        }

        // Criar uma nova tarefa
        var task = new TaskEntity
        {
            Title = request.Title,
            Description = request.Description,
            CreatedAt = DateTime.UtcNow
        };

        _taskRepository.AddTask(task);

        return new TaskResult
        {
            Success = true,
            TaskId = task.Id
        };
    }

    public IEnumerable < TaskDto > GetTasks()
    {
        var tasks = _taskRepository.GetAllTasks();
        return tasks.Select(task => new TaskDto
        {
            Id = task.Id,
            Title = task.Title,
            Description = task.Description
```

```
        });
    }
}
```

Domain Layer

A camada de domínio contém as entidades e as regras principais do negócio:

```
// Domain Entity
public class TaskEntity
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public DateTime CreatedAt { get; set; }
}

// Domain Repository Interface
public interface ITaskRepository
{
    void AddTask(TaskEntity task);
    IEnumerable < TaskEntity > GetAllTasks();
}

// Domain Service Interface
public interface ITaskService
{
    TaskResult CreateTask(CreateTaskRequest request);
    IEnumerable < TaskDto > GetTasks();
}
```

Infrastructure Layer

A camada de infraestrutura fornece implementações concretas e acesso a sistemas externos, como bancos de dados:

```
// Infrastructure Implementation
public class TaskRepository : ITaskRepository
{
    private readonly ApplicationDbContext _context;

    public TaskRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public void AddTask(TaskEntity task)
    {
        _context.Tasks.Add(task);
        _context.SaveChanges();
    }

    public IEnumerable < TaskEntity > GetAllTasks()
    {
        return _context.Tasks.ToList();
    }
```

```
}

// ApplicationDbContext
public class ApplicationDbContext : DbContext
{
    public DbSet < TaskEntity > Tasks { get; set; }

    public ApplicationDbContext(DbContextOptions < ApplicationDbContext > options) : ba
se(options) { }
}
```

Conectando as camadas

Configuração da injeção de dependência:

```
// Startup.cs or Program.cs
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext < ApplicationDbContext > (options =>
            options.UseSqlServer("YourConnectionString"));

        services.AddScoped < ITaskRepository, TaskRepository > ();
        services.AddScoped < ITaskService, TaskService > ();
        services.AddControllers();
    }
}
```

A Arquitetura Limpa, ao separar rigorosamente as preocupações em camadas, facilita a manutenção e a evolução do sistema, garantindo que as mudanças em uma camada não afetem as demais. Isso também promove a reutilização de código, uma vez que a lógica de negócio pode ser utilizada em diferentes contextos, como uma aplicação web, uma API REST, ou uma aplicação de console, sem necessidade de alterações. Além disso, essa arquitetura permite que a lógica de negócio seja testada de maneira isolada, sem a necessidade de dependências externas, o que resulta em testes mais rápidos e confiáveis.

Outro aspecto importante da Arquitetura Limpa é sua capacidade de lidar com mudanças. Em um ambiente de desenvolvimento ágil, onde as mudanças são constantes, essa arquitetura permite que o sistema evolua de forma incremental, sem que seja necessário reescrever grandes partes do código. Ao manter a lógica de negócio isolada das demais camadas, a Arquitetura Limpa promove uma maior flexibilidade e adaptabilidade do sistema, permitindo que ele acompanhe as mudanças nas necessidades do negócio e nas tecnologias utilizadas.

Diferenças entre Portas e Interfaces

A principal diferença entre as portas da Arquitetura Hexagonal e as interfaces da Arquitetura Limpa reside no contexto e no propósito de cada uma. Na Arquitetura Hexagonal, as portas são os pontos de entrada e saída do núcleo da aplicação, permitindo que diferentes adaptadores sejam conectados de acordo com a necessidade. Por outro lado, na Arquitetura Limpa, as interfaces servem como contratos que definem as interações entre as camadas da aplicação. Embora ambas as abordagens utilizem interfaces para promover a flexibilidade e a modularidade do código, a forma como essas interfaces são empregadas e a estrutura da aplicação como um todo diferem entre as duas arquiteturas.

Enquanto a Arquitetura Hexagonal foca na comunicação bidirecional entre o núcleo da aplicação e o mundo externo,

a Arquitetura Limpa enfatiza uma estrutura em camadas que isola completamente a lógica de negócio de qualquer dependência externa. Essa diferença fundamental influencia a forma como os sistemas são projetados e mantidos ao longo do tempo. Na Arquitetura Hexagonal, a ênfase está em permitir que o núcleo da aplicação se comunique com o mundo externo de maneira flexível, enquanto na Arquitetura Limpa, o foco é garantir que o núcleo da aplicação seja completamente independente de detalhes externos.

Considerações Finais

Tanto a Arquitetura Hexagonal quanto a Arquitetura Limpa oferecem abordagens valiosas para a construção de sistemas flexíveis, modulares e de fácil manutenção. A escolha entre uma ou outra depende das necessidades específicas do projeto e das preferências da equipe de desenvolvimento. Em alguns casos, pode ser interessante combinar aspectos das duas abordagens para tirar proveito de suas respectivas vantagens. Independentemente da escolha, é essencial que a arquitetura de software seja pensada de forma a suportar a evolução do sistema, permitindo que ele se adapte às mudanças tecnológicas e às novas demandas do negócio de forma contínua e eficiente.

Referências

Para obter mais informações sobre arquiteturas, consulte a documentação oficial e outros recursos disponíveis:

• Richard, M. Hexagonal Architecture. In: Richards, Mark. Software Architecture Patterns. Sebastopol: O'Reilly Media, 2015. Disponível em: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437. Acesso em: 19 ago. 2024.

• Martin, R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall. Acesso em: 21 ago. 2024.

Nagib é Professor Universitário e Tech Manager. Possui uma trajetória de conquistas em certificações técnicas e ágeis, incluindo MCSD, MCSA e PSM1. PG em Gestão de TI pelo SENAC e MBA em Tecnologia de Software pela USP, Nagib cursou programas de extensão do MIT e Universidade de Chicago. Outras conquistas incluem a autoria de um artigo sobre chatbots, revisado por pares e apresentado na Universidade de Barcelona.