PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS

ENGENHARIA DE SOFTWARE

23012069 - Jéssica Silva Kushida

23013238 - Marcela Franco

23008255 - Natália Naomi Sumida

23009486 – Nicole Silvestrini Garrio

ATIVIDADE FINAL

Padrões e Arquitetura de Software

CAMPINAS - SP

2025

SUMÁRIO

1. INTRODUÇÃO

O seguinte sistema descrito é uma aplicação dos conceitos aprendidos na matéria de Padrões e Arquitetura de Software sobre boas práticas de programação (princípios de SOLID e Clean Code), assim como sobre padrões de projeto GoF.

1.1. Objetivo do sistema

O sistema desenvolvido é um Monitor de Previsão do Tempo, cujo objetivo é coletar, processar e exibir informações meteorológicas atualizadas a partir de diversas fontes confiáveis (como OpenWeatherMap e WeatherBit), centralizando esses dados em um único local.

As informações exibidas pelo sistema incluem: provedor de dados, cidade, estado, país, temperatura mínima, temperatura máxima e a data/hora da última atualização.

O projeto foi desenvolvido em .NET, utilizando a linguagem C#, com o auxílio da IDE Microsoft Visual Studio 2022.

2. ARQUITETURA UTILIZADA (CLEAN ARCHITECTURE)

Neste projeto, adotamos a Clean Architecture, proposta por Robert C. Martin (Uncle Bob), que visa organizar o sistema em camadas independentes e bem definidas, separando claramente as regras de negócio, a lógica de aplicação, os detalhes de infraestrutura e os mecanismos de interface com o usuário.

Essa abordagem favorece a separação de responsabilidades, facilita testes automatizados, promove reutilização de código e aumenta a flexibilidade para substituição de provedores de dados, fontes externas e canais de exibição (CLI, Web, etc.).

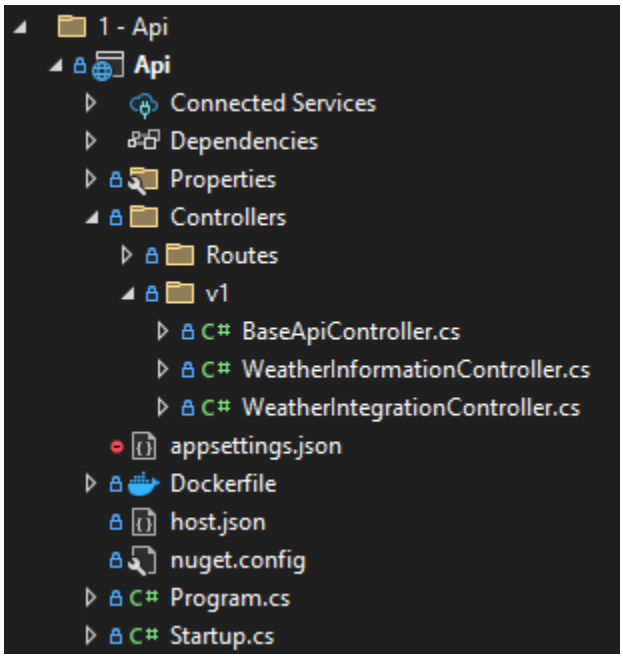
A estrutura geral do sistema foi dividida em quatro camadas principais:

2.1 API:

Essa é a camada de apresentação do sistema, responsável por exibir os dados ao usuário final. Atualmente, a principal interface implementada é uma aplicação de linha de comando (CLI), porém a arquitetura adotada facilita a adição de outras interfaces, como uma aplicação web ou uma API REST. Dentro do projeto da API, temos componentes fundamentais como:

- Program.cs e Startup.cs, que inicializam a aplicação e configuram os serviços e middlewares necessários.
- appsettings.json, que armazena configurações gerais, como strings de conexão e chaves de acesso.
- Controllers, responsáveis por receber as requisições, repassar aos serviços da aplicação e retornar as respostas ao usuário.

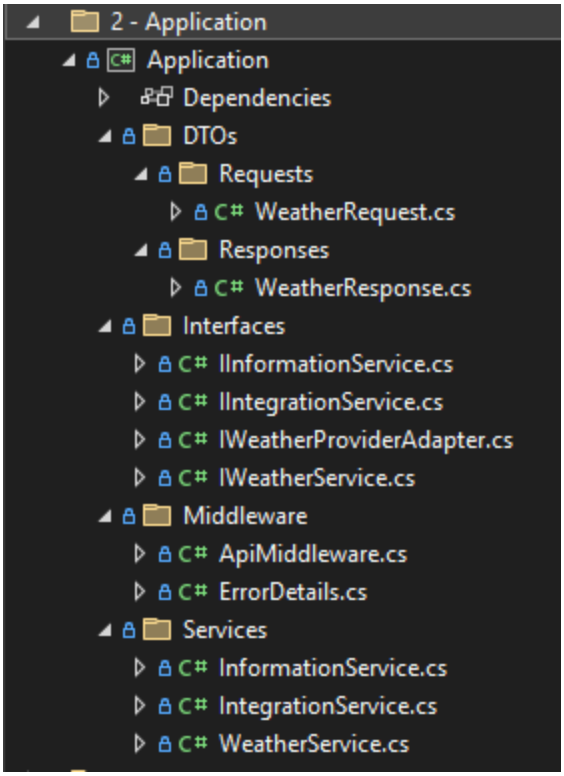
Além disso, utilizamos o Swagger, uma ferramenta integrada à API para facilitar a documentação, visualização e teste dos endpoints, permitindo que qualquer pessoa consiga explorar os dados do sistema de forma prática e intuitiva.



2.2 **Application**:

Essa camada implementa os casos de uso do sistema e faz a ponte entre o domínio (domain) e a infraestrutura. É responsável por orquestrar o fluxo de dados entre as demais camadas, mantendo-se desacoplada de detalhes técnicos externos. Nela estão contidos:

- DTOs (Requests e Responses): responsáveis por transportar os dados entre a API e os serviços internos, facilitando a entrada e saída de informações no formato esperado.
- Interfaces (Service e Adapter): definem o comportamento esperado dos serviços e provedores de clima, permitindo fácil substituição ou adição de novas implementações.
- Services: classes como IntegrationService, InformationService e WeatherService que contêm a lógica de aplicação, como a coordenação da coleta de dados meteorológicos e o envio de informações para o domain.
- Middleware: componentes intermediários usados para lidar com aspectos transversais, como tratamento de erros ou autenticação, se aplicável.

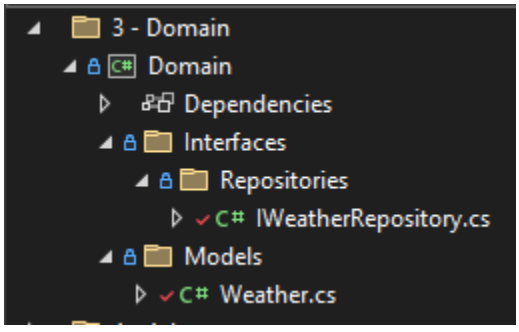


2.3 Domain (Models):

Essa camada contém as entidades e interfaces principais do sistema.

Em Models, temos a classe Weather, que representa os dados meteorológicos padronizados no formato em que serão enviados ao banco.

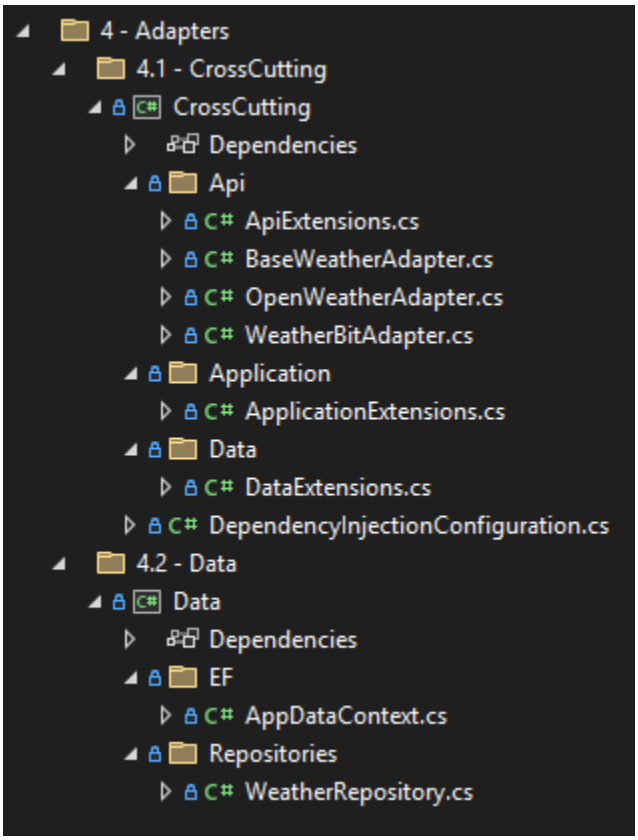
Em Interfaces, temos IWeatherRepository, que define as operações necessárias para acessar e salvar esses dados, sem se preocupar com como isso será feito. Dessa forma, a camada permanece independente de detalhes técnicos, focada apenas nas regras e estrutura do sistema.



2.4 Infraestrutura (Adapters):

Essa camada é responsável por lidar com os detalhes técnicos da aplicação, como a comunicação com APIs externas (ex: OpenWeather, WeatherBit) e a persistência de dados no banco. Ela contém os adapters que convertem os dados recebidos de fontes externas para o formato padronizado da entidade Weather no domain. Dentro da infraestrutura, temos os seguintes componentes organizados em projetos auxiliares:

- CrossCutting: responsável por configurações gerais e utilitários compartilhados, como DependencyInjection.
- Data: implementa o acesso ao banco de dados, utilizando o Entity Framework (EF) e os repositórios definidos na camada de domain.



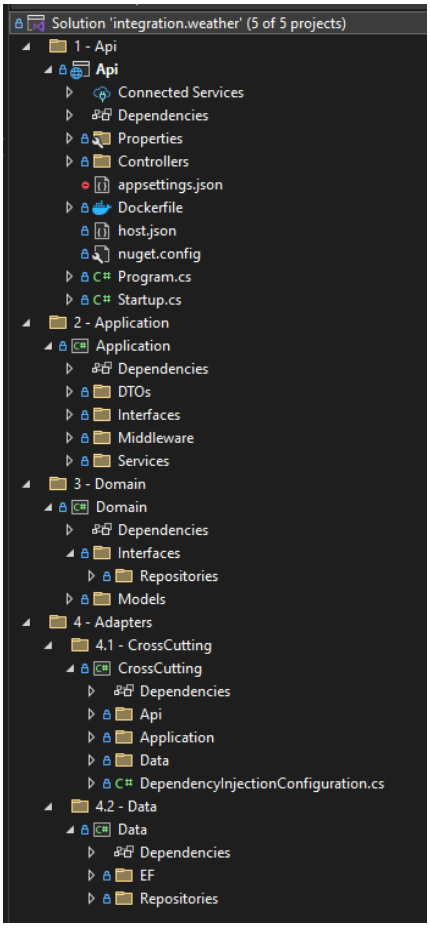
3. APLICAÇÃO DOS PRINCÍPIOS SOLID

A aplicação dos princípios SOLID na solution integration.weather garante uma arquitetura limpa, coesa e de fácil manutenção. Cada princípio foi aplicado na prática da seguinte forma:

3.1 Single Responsibility Principle (SRP)

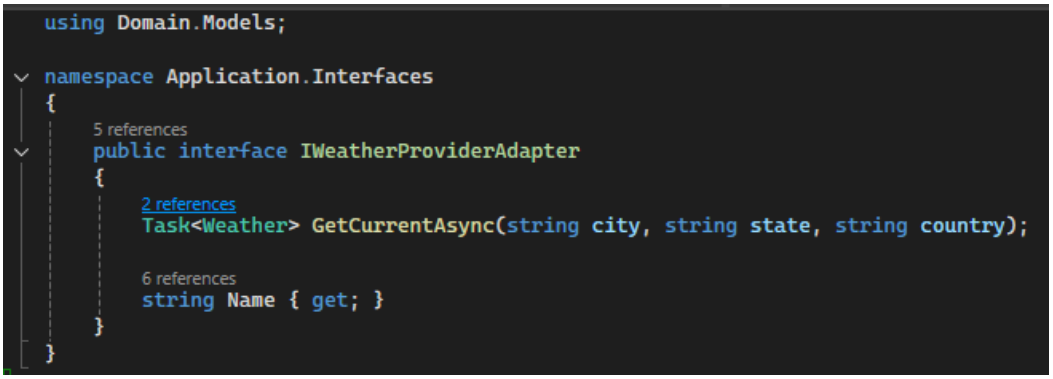
Cada classe possui uma única responsabilidade, o que é evidente na organização dos projetos:

- Api: contém apenas os Controllers, responsáveis pelos endpoints REST.
- Application: concentra os Services e DTOs, que orquestram os casos de uso.
- Domain: define entidades e interfaces, sem dependências externas.
- Adapters: lida com integrações externas, como as chamadas das APIs OpenWeatherMap e WeatherBit.



3.2 Open-Closed Principle (OCP)

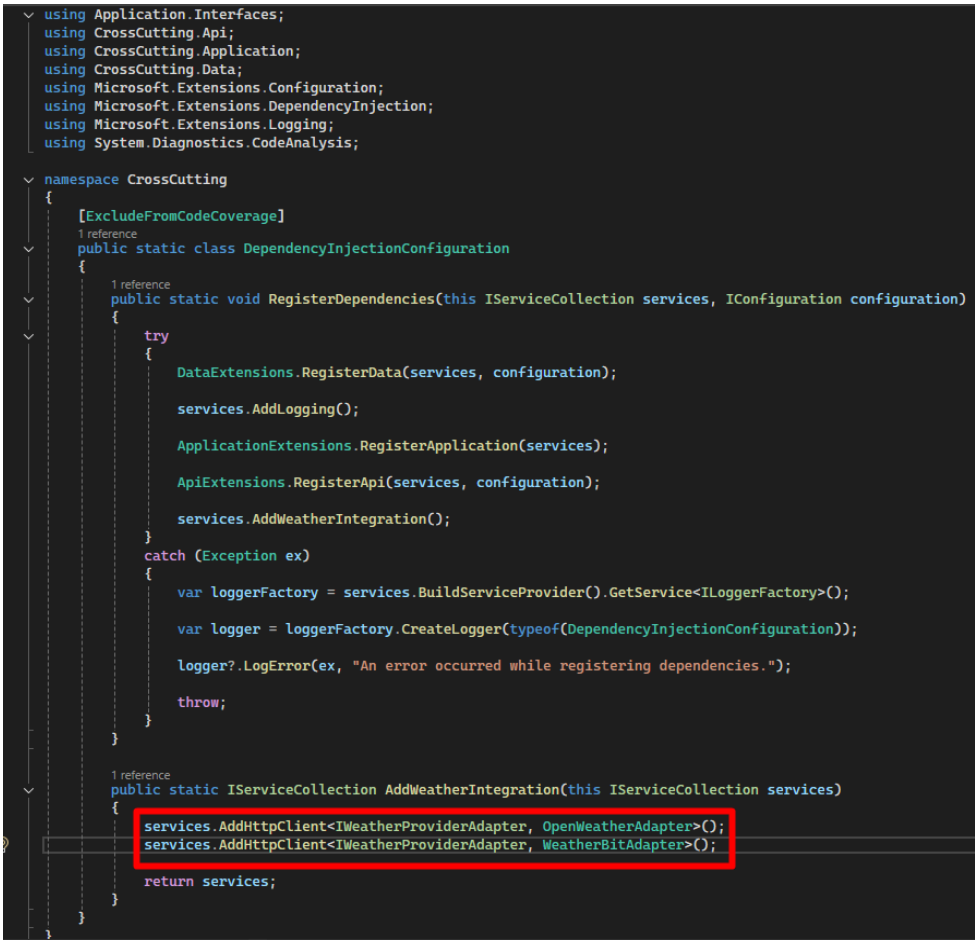
Componentes são abertos para extensão, mas fechados para modificação. As Interfaces como *IWeatherProviderAdapter* e *IWeatherService* permitem adicionar novos provedores por meio de novas classes que implementam essas interfaces, sem alterar código existente. A injeção de dependência facilita o uso dessas implementações sem impactar a lógica dos serviços.



```
using Domain.Models;

namespace Application.Interfaces
{
    5 references
    public interface IWeatherProviderAdapter
    {
        2 references
        Task<Weather> GetCurrentAsync(string city, string state, string country);

        6 references
        string Name { get; }
    }
}
```

```
using Domain.Models;
using Microsoft.Extensions.Configuration;
using System.Text.Json;

2 references
public class WeatherBitAdapter : BaseWeatherAdapter
{
    5 references
    public override string Name => "WeatherBit";

    0 references
    public WeatherBitAdapter(HttpClient http, IConfiguration config)
        : base(http, config, "WeatherBit:ApiKey")
    { }

    2 references
    protected override string BuildUrl(string city, string state, string country, string apiKey)
        => "https://api.weatherbit.io/v2.0/forecast/hourly" +
           $"?city={Uri.EscapeDataString(city)}" +
           (string.IsNullOrEmpty(state)
               ? ""
               : $"&state={Uri.EscapeDataString(state)}") +
           $"&country={Uri.EscapeDataString(country)}" +
           $"&hours=24" +
           $"&key={apiKey}";

    2 references
    protected override Weather ParseWeather(JsonDocument doc, string city, string state, string country)
    {
        var temps = doc.RootElement
                       .GetProperty("data")
                       .EnumerateArray()
                       .Select(el => el.GetProperty("temp").GetDouble())
                       .ToList();

        double minTemp = temps.Min();
        double maxTemp = temps.Max();

        return new Weather
        {
            City = city,
            State = state,
            Country = country,
            CelsiusTemperatureMin = (long)Math.Round(minTemp),
            CelsiusTemperatureMax = (long)Math.Round(maxTemp),
            LastUpdate = DateTime.UtcNow
        };
    }
}
```

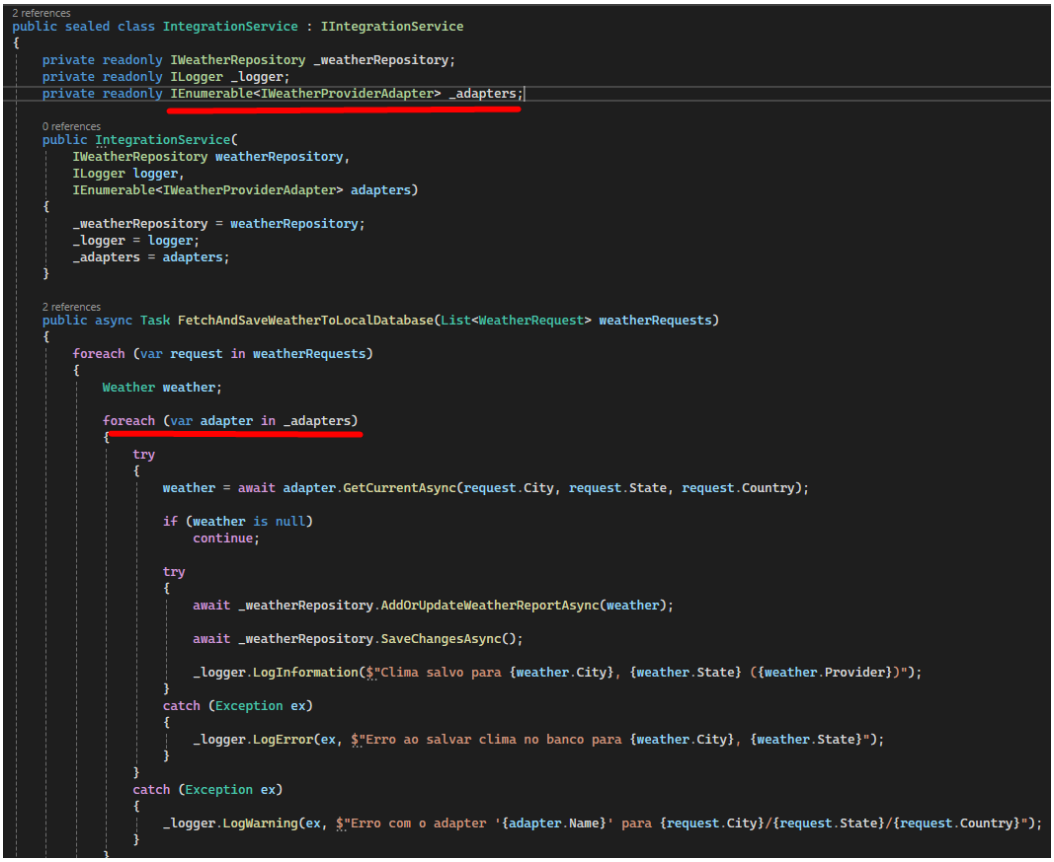
```
using Domain.Models;
using Microsoft.Extensions.Configuration;
using System.Text.Json;

2 references
public class OpenWeatherAdapter : BaseWeatherAdapter
{
    5 references
    public override string Name => "OpenWeatherMap";

    0 references
    public OpenWeatherAdapter(HttpClient http, IConfiguration config)
        : base(http, config, "OpenWeather:ApiKey")
    { }

    2 references
    protected override string BuildUrl(string city, string state, string country, string apiKey)
        => $"https://api.openweathermap.org/data/2.5/weather" +
           $"?q={Uri.EscapeDataString(city)},{Uri.EscapeDataString(country)}" +
           $"&appid={apiKey}" +
           $"&units=metric";

    2 references
    protected override Weather ParseWeather(JsonDocument doc, string city, string state, string country)
    {
        var main = doc.RootElement.GetProperty("main");
        return new Weather
        {
            City = city,
            State = state,
            Country = country,
            CelsiusTemperatureMin = (long)main.GetProperty("temp_min").GetDouble(),
            CelsiusTemperatureMax = (long)main.GetProperty("temp_max").GetDouble(),
            LastUpdate = DateTime.UtcNow
        };
    }
}
```



```
using Application.Interfaces;
using CrossCutting.Api;
using CrossCutting.Application;
using CrossCutting.Data;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using System.Diagnostics.CodeAnalysis;

namespace CrossCutting
{
    [ExcludeFromCodeCoverage]
    1 reference
    public static class DependencyInjectionConfiguration
    {
        1 reference
        public static void RegisterDependencies(this IServiceCollection services, IConfiguration configuration)
        {
            try
            {
                DataExtensions.RegisterData(services, configuration);

                services.AddLogging();

                ApplicationExtensions.RegisterApplication(services);

                ApiExtensions.RegisterApi(services, configuration);

                services.AddWeatherIntegration();
            }
            catch (Exception ex)
            {
                var loggerFactory = services.BuildServiceProvider().GetService<ILoggerFactory>();

                var logger = loggerFactory.CreateLogger(typeof(DependencyInjectionConfiguration));

                logger?.LogError(ex, "An error occurred while registering dependencies.");

                throw;
            }
        }

        1 reference
        public static IServiceCollection AddWeatherIntegration(this IServiceCollection services)
        {
            services.AddHttpClient<IWeatherProviderAdapter, OpenWeatherAdapter>();
            services.AddHttpClient<IWeatherProviderAdapter, WeatherBitAdapter>();

            return services;
        }
    }
}
```

3.3 Liskov Substitution Principle (LSP)

Qualquer implementação de *IWeatherProviderAdapter* pode substituir outra sem alterar o comportamento do sistema. Por exemplo, *OpenWeatherAdapter* e *WeatherBitAdapter* seguem o mesmo contrato, podendo ser injetadas no *IntegrationService* de forma intercambiável, com retorno padronizado.



```
2 references
public sealed class IntegrationService : IIntegrationService
{
    private readonly IWeatherRepository _weatherRepository;
    private readonly ILogger _logger;
    private readonly IEnumerable<IWeatherProviderAdapter> _adapters;

    0 references
    public IntegrationService(
        IWeatherRepository weatherRepository,
        ILogger logger,
        IEnumerable<IWeatherProviderAdapter> adapters)
    {
        _weatherRepository = weatherRepository;
        _logger = logger;
        _adapters = adapters;
    }

    2 references
    public async Task FetchAndSaveWeatherToLocalDatabase(List<WeatherRequest> weatherRequests)
    {
        foreach (var request in weatherRequests)
        {
            Weather weather;

            foreach (var adapter in _adapters)
            {
                try
                {
                    weather = await adapter.GetCurrentAsync(request.City, request.State, request.Country);

                    if (weather is null)
                        continue;

                    try
                    {
                        await _weatherRepository.AddOrUpdateWeatherReportAsync(weather);

                        await _weatherRepository.SaveChangesAsync();

                        _logger.LogInformation($"Clima salvo para {weather.City}, {weather.State} ({weather.Provider})");
                    }
                    catch (Exception ex)
                    {
                        _logger.LogError(ex, $"Erro ao salvar clima no banco para {weather.City}, {weather.State}");
                    }
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, $"Erro com o adapter '{adapter.Name}' para {request.City}/{request.State}/{request.Country}");
                }
            }
```

3.4 Interface Segregation Principle (ISP)

Interfaces sao especificas e focadas, evitando que implementações tenham métodos que não utilizam, por exemplo:

- *IWeatherProviderAdapter*: define apenas o método para buscar dados de clima.
- *IIntegrationService*: orquestra múltiplos adapters.
- *IInformationService*: trata da formatação e entrega das informações climáticas.

Interfaces são específicas e focadas, evitando que implementações tenham métodos que não utilizam, por exemplo:

- IWeatherProviderAdapter: define apenas o método para buscar dados de clima.
- IIntegrationService: orquestra múltiplos adapters.

- IInformationService: trata da formatação e entrega das informações climáticas.

```
using Domain.Models;

namespace Application.Interfaces
{
    5 references
    public interface IWeatherProviderAdapter
    {
        2 references
        Task<Weather> GetCurrentAsync(string city, string state, string country);

        6 references
        string Name { get; }
    }
}
```

```
namespace Application.Interfaces
{
    4 references
    public interface IIntegrationService : IDisposable
    {
        2 references
        Task FetchAndSaveWeatherToLocalDatabase(List<WeatherRequest> weatherRequest);
    }
}
```

```
namespace Application.Interfaces
{

    public interface IInformationService : IDisposable
    {

        Task<List<WeatherResponse>> GetAllWeather();

        2 references
        Task<List<WeatherResponse>> GetAllWeatherFromProvider(string provider);

        2 references
        Task<List<WeatherResponse>> GetWeatherFromCity(string city, string state, string country);
    }
}
```

3.5 Dependency Inversion Principle (DIP)

Módulos de alto nível (Api, Application) não dependem de implementações concretas, mas sim de abstrações como IIntegrationService e IWeatherProviderAdapter. As classes concretas (OpenWeatherAdapter, WeatherBitAdapter) estão isoladas em Adapters e são conectadas à aplicação via injeção de dependência, configurada centralmente em

DependencyInjectionConfiguration.cs o que promove um sistema desacoplado, alinhado à Clean Architecture.

```
namespace Api.Controllers.v1
{
    [Route($"api/v{{version:apiVersion}}/{ApiRoutes.Integration}")]
    1 reference
    public class WeatherIntegrationController : BaseApiController
    {
        private readonly IIntegrationService _integrationService;

        0 references
        public WeatherIntegrationController(
            IIntegrationService integrationService)
        {
            _integrationService = integrationService;
        }

        [HttpPost("FetchAndSaveWeatherToLocalDatabase")]
        [ProducesResponseType(StatusCodes.Status200OK)]
        [ProducesResponseType(StatusCodes.Status401Unauthorized)]
        [ProducesResponseType(StatusCodes.Status403Forbidden)]
        [ProducesResponseType(StatusCodes.Status405MethodNotAllowed)]
        [ProducesResponseType(StatusCodes.Status412PreconditionFailed)]
        [ProducesResponseType(StatusCodes.Status500InternalServerError)]
        0 references
        public async Task<IActionResult> FetchAndSaveWeatherToLocalDatabase([FromBody] List<WeatherRequest> weatherRequest)
        {
            await _integrationService.FetchAndSaveWeatherToLocalDatabase(weatherRequest);

            return Ok();
        }
    }
}
```

```
namespace CrossCutting
{
    [ExcludeFromCodeCoverage]
    1 reference
    public static class DependencyInjectionConfiguration
    {
        1 reference
        public static void RegisterDependencies(this IServiceCollection services, IConfiguration configuration)
        {
            try
            {
                DataExtensions.RegisterData(services, configuration);

                services.AddLogging();

                ApplicationExtensions.RegisterApplication(services);

                ApiExtensions.RegisterApi(services, configuration);

                services.AddWeatherIntegration();
            }
            catch (Exception ex)
            {
                var loggerFactory = services.BuildServiceProvider().GetService<ILoggerFactory>();

                var logger = loggerFactory.CreateLogger(typeof(DependencyInjectionConfiguration));

                logger?.LogError(ex, "An error occurred while registering dependencies.");

                throw;
            }
        }

        1 reference
        public static IServiceCollection AddWeatherIntegration(this IServiceCollection services)
        {
            services.AddHttpClient<IWeatherProviderAdapter, OpenWeatherAdapter>();
            services.AddHttpClient<IWeatherProviderAdapter, WeatherBitAdapter>();

            return services;
        }
    }
}
```

4. CLEAN CODE

4.1. Nomes claros e expressivos

- Classes como *WeatherInformationController*, *IntegrationService*, *IWeatherProviderAdapter* e *OpenWeatherAdapter* são autoexplicativas, revelando imediatamente seu propósito.
- Métodos como *GetAllWeatherFromProvider* e *FetchAndSaveWeatherToLocalDatabase* descrevem precisamente a ação que executam.

4.2. Métodos enxutos e objetivos

- Cada método faz apenas uma ação:
    - *GetAllWeatherFromProvider* chama *IInformationService* e retorna os dados.
    - O *GetCurrentAsync* no *BaseWeatherAdapter* executa um pipeline curto: montar URL → executar GET → tratar resposta → parsear JSON e retornar *Weather*.

4.3. Baixo acoplamento e alta coesão

- Camadas bem isoladas:do
    - **API** trata apenas de HTTP e rota.
    - **Application** orquestra lógica, delegando tratamentos.
    - **Adapters** lidam com APIs externas.
    - **Domain/Data** cuidam de entidades e persistência.
- Cada classe mantém sua responsabilidade única, evitando dependências desnecessárias entre camadas.

4.4. Tratamento de erros e logs

- Uso de try/catch refinado no *IntegrationService* com registros específicos de erros (*LogWarning*, *LogError*), facilitando diagnóstico sem interromper fluxos não críticos.

4.5. Dependência de abstrações

- Todas as dependências são injetadas via interfaces, evitando acoplamento direto a implementações e facilitando testes unitários por injeção de mocks.

4.6. Respeito aos princípios DRY/Single-responsibility

- O *BaseWeatherAdapter* centraliza lógica comum de requisição, enquanto subclasses apenas definem URL e parsing, evitando repetir código HTTP ou tratamento de mensagens de erro.

5. PADRÕES DE PROJETOS GoF

Justificativa detalhada para a escolha dos padrões GoF aplicados.

Descrição técnica detalhada sobre como os padrões foram implementados no projeto.

Análise crítica das vantagens e eventuais desvantagens identificadas.

5.1. Adapter

Porquê o Adapter?

A decisão de aplicar o padrão adapter nesse projeto surgiu da necessidade de integrar múltiplas APIs (OpenWeather, WeatherBit) com interfaces e formatos diferentes. Para isso, gostaríamos de abstrair as particularidades de cada uma das APIs, oferecendo uma única interface (*IWeatherProviderAdapter*), a qual fosse consumida pela aplicação independente do provedor da informação.

Como foi implementado?

- Interface comum:
    - Adapters (ex: *OpenWeatherAdapter*, *WeatherBitAdapter*) implementam essa interface, tradução do JSON da API externa

para o modelo interno *Weather*.

```
using Domain.Models;

namespace Application.Interfaces
{
    5 references
    public interface IWeatherProviderAdapter
    {
        2 references
        Task<Weather> GetCurrentAsync(string city, string state, string country);

        6 references
        string Name { get; }
    }
}
```

- Registro via DI:
    - O *IntegrationService* consome a interface, sem depender de implementações concretas.

```
1 reference
public static IServiceCollection AddWeatherIntegration(this IServiceCollection services)
{
    services.AddHttpClient<IWeatherProviderAdapter, OpenWeatherAdapter>();
    services.AddHttpClient<IWeatherProviderAdapter, WeatherBitAdapter>();

    return services;
}
```

```
using CrossCutting.Data;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using System.Diagnostics.CodeAnalysis;

namespace CrossCutting
{
    [ExcludeFromCodeCoverage]
    1 reference
    public static class DependencyInjectionConfiguration
    {
        1 reference
        public static void RegisterDependencies(this IServiceCollection services, IConfiguration configuration)
        {
            try
            {
                DataExtensions.RegisterData(services, configuration);

                services.AddLogging();

                ApplicationExtensions.RegisterApplication(services);

                ApiExtensions.RegisterApi(services, configuration);

                services.AddWeatherIntegration();
            }
            catch (Exception ex)
            {
                var loggerFactory = services.BuildServiceProvider().GetService<ILoggerFactory>();

                var logger = loggerFactory.CreateLogger(typeof(DependencyInjectionConfiguration));

                logger?.LogError(ex, "An error occurred while registering dependencies.");

                throw;
            }
        }

        1 reference
        public static IServiceCollection AddWeatherIntegration(this IServiceCollection services)
        {
            services.AddHttpClient<IWeatherProviderAdapter, OpenWeatherAdapter>();
            services.AddHttpClient<IWeatherProviderAdapter, WeatherBitAdapter>();

            return services;
        }
    }
}
```

Conclusões

A aplicação do Adapter facilita a adição de novos providers sem alterar a lógica central, porém pode gerar duplicação entre adapters, se muitos forem criados. Nesse caso poderia ser considerado a criação de composições.

5.2. Template Method

Porquê o Template Method?

Embora os provedores escolhidos precisem construir URLs e parsear JSON, grande parte do processo é idêntico: requisição HTTP, verificação, parsing. O Desejo era evitar a duplicação e aproveitar a semelhança para padronizar o processo.

Como foi implementado?

- Subclasses implementam as etapas específicas: BuildUrl(...), ParseWeather(...).

```
using Application.Interfaces;
using Domain.Models;
using Microsoft.Extensions.Configuration;
using System.Text.Json;

5 references
public abstract class BaseWeatherAdapter : IWeatherProviderAdapter
{
    6 references
    public abstract string Name { get; }

    private readonly HttpClient _http;
    private readonly string _apiKey;

    2 references
    protected BaseWeatherAdapter(HttpClient http, IConfiguration config, string configKey)
    {
        _http = http;
        _apiKey = config[configKey]
                  ?? throw new ArgumentException($"Configuration key '{configKey}' not found");
    }

    3 references
    protected abstract string BuildUrl(string city, string state, string country, string apiKey);

    3 references
    protected abstract Weather ParseWeather(JsonDocument doc, string city, string state, string country);

    2 references
    public async Task<Weather> GetCurrentAsync(string city, string state, string country)
    {
        var url = BuildUrl(city, state, country, _apiKey);
        using var response = await _http.GetAsync(url);

        if (!response.IsSuccessStatusCode)
        {
            var error = await response.Content.ReadAsStringAsync();
            throw new Exception(
                $"{Name} call failed ({response.StatusCode}): {error}"
            );
        }

        using var json = JsonDocument.Parse(await response.Content.ReadAsStringAsync());
        var weather = ParseWeather(json, city, state, country);
        weather.Provider = Name;
        return weather;
    }
}
```

Conclusões

O template method evita a repetição e centraliza o fluxo HTTP. No caso dos providers que divergem muito, como por exemplo fluxos de autenticação diferentes, pode ser que seja necessário quebrar o *BaseWeatherAdapter* ou talvez combinar com outro padrão.

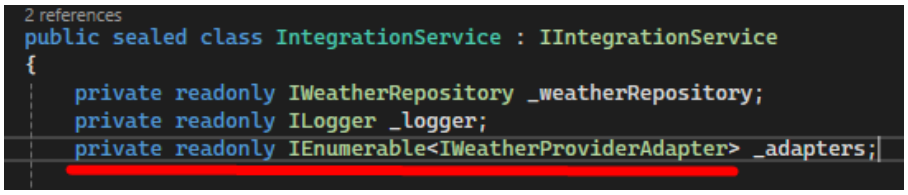
5.3. Strategy

Porque o Strategy?

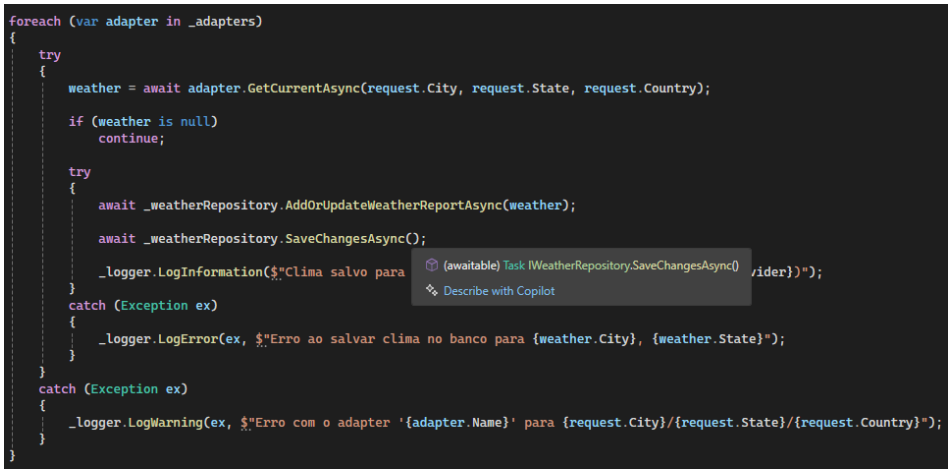
O Strategy foi aplicado para permitir a seleção dinâmica de provedor no runtime, inclusive suportar fallback entre múltiplos adaptadores. Esse padrão permite a troca do provedor sem alterar o cliente (*IntegrationService*).

Como foi implementado?

- O *IntegrationService* recebe *IEnumerable<IWeatherProviderAdapter> adapters* via DI.



- Alternativamente, *InformationService* poderia selecionar um único provider com base no parâmetro *Name*.



Conclusões

O padrão strategy permite a flexibilidade e inclusão de diferentes providers. Uma desvantagem é que ele executa todos por padrão, o que pode gerar um aumento de latência. O ideal seria filtrar pelo *Name* ou utilizar estratégias de fallback/timeout.

5.4. Repository

Porque o Repository?

A implementação do repository se deu para que fosse possível desacoplar a lógica de armazenamento usando EF Core da aplicação, além de facilitar testes com abstrações.

Como foi implementado?

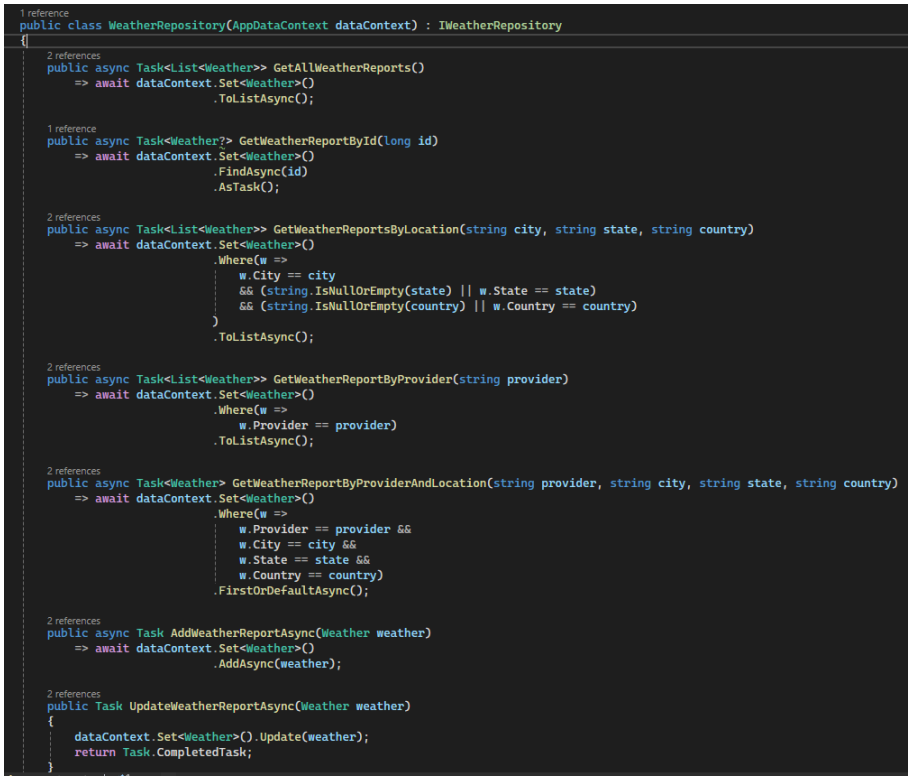
- Interface *IWeatherRepository* define métodos de CRUD: *GetAllWeatherReports(), AddOrUpdateWeatherReportAsync(), SaveChangesAsync().*

```
using Domain.Models;

namespace Domain.Interfaces.Repositories
{
    6 references
    public interface IWeatherRepository
    {
        2 references
        Task<List<Weather>> GetAllWeatherReports();

        1 reference
        Task<Weather?> GetWeatherReportById(long id);

        2 references
        Task<List<Weather>> GetWeatherReportsByLocation(string city, string state, string country);

        2 references
        Task<List<Weather>> GetWeatherReportByProvider(string provider);

        2 references
        Task<Weather> GetWeatherReportByProviderAndLocation(string source, string city, string state, string country);

        2 references
        Task AddWeatherReportAsync(Weather weather);

        2 references
        Task UpdateWeatherReportAsync(Weather weather);

        2 references
        Task AddOrUpdateWeatherReportAsync(Weather weather);

        2 references
        Task SaveChangesAsync();
    }
}
```

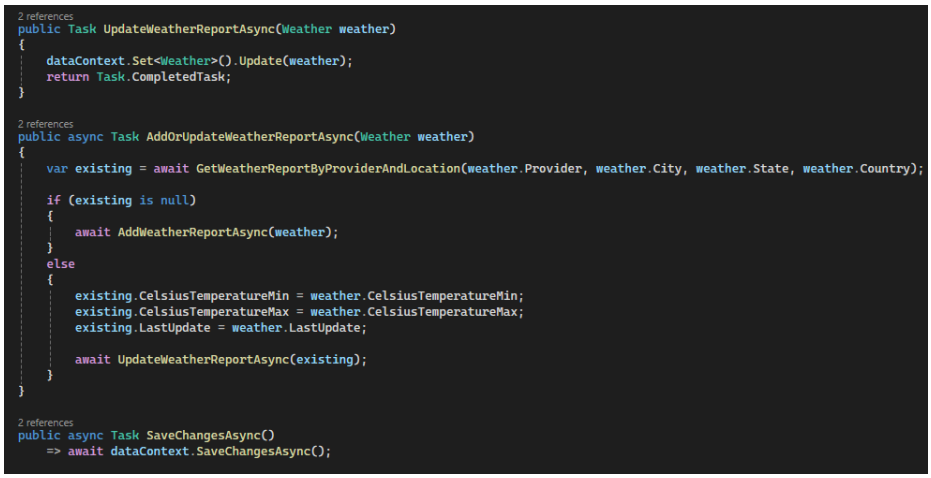
- Implementação concreta em *WeatherRepository* utiliza *AppDataContext* e EF Core para persistir *Weather* no banco.

```
namespace Domain.Models
{

    public class Weather
    {

        public long Id { get; set; }


        public string Provider { get; set; }


        public string City { get; set; }


        public string State { get; set; }


        public string Country { get; set; }


        public long CelsiusTemperatureMin { get; set; }


        public long CelsiusTemperatureMax { get; set; }

        7 references
        public DateTime LastUpdate { get; set; }
    }
}
```

```
public class AppDataContext : DbContext
{
    0 references
    public DbSet<Weather> Weather { get; set; }

    private readonly IConfiguration _configuration;

    0 references
    public AppDataContext(
        DbContextOptions<AppDataContext> options,
        IConfiguration configuration)
        : base(options)
    {
        _configuration = configuration;
    }

    0 references
    public AppDataContext(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
            var conn = _configuration.GetConnectionString("MySQLConnection")
                       ?? throw new InvalidOperationException(
                              "Connection string 'MySQLConnection' not found.");

            optionsBuilder
                .UseMySql(conn, ServerVersion.AutoDetect(conn))
                .UseExceptionProcessor();
        }
    }

    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Weather>(entity =>
        {
            entity.ToTable("weather");
            entity.HasKey(e => e.Id);
        });

        base.OnModelCreating(modelBuilder);
    }
}
```



```
1 reference
public class WeatherRepository(AppDataContext dataContext) : IWeatherRepository
{
    2 references
    public async Task<List<Weather>> GetAllWeatherReports()
        => await dataContext.Set<Weather>()
                            .ToListAsync();

    1 reference
    public async Task<Weather?> GetWeatherReportById(long id)
        => await dataContext.Set<Weather>()
                            .FindAsync(id)
                            .AsTask();

    2 references
    public async Task<List<Weather>> GetWeatherReportsByLocation(string city, string state, string country)
        => await dataContext.Set<Weather>()
                            .Where(w =>
                                w.City == city
                                && (string.IsNullOrEmpty(state) || w.State == state)
                                && (string.IsNullOrEmpty(country) || w.Country == country)
                            )
                            .ToListAsync();

    2 references
    public async Task<List<Weather>> GetWeatherReportByProvider(string provider)
        => await dataContext.Set<Weather>()
                            .Where(w =>
                                w.Provider == provider)
                            .ToListAsync();

    2 references
    public async Task<Weather> GetWeatherReportByProviderAndLocation(string provider, string city, string state, string country)
        => await dataContext.Set<Weather>()
                            .Where(w =>
                                w.Provider == provider &&
                                w.City == city &&
                                w.State == state &&
                                w.Country == country)
                            .FirstOrDefaultAsync();

    2 references
    public async Task AddWeatherReportAsync(Weather weather)
        => await dataContext.Set<Weather>()
                            .AddAsync(weather);

    2 references
    public Task UpdateWeatherReportAsync(Weather weather)
    {
        dataContext.Set<Weather>().Update(weather);
        return Task.CompletedTask;
    }
```



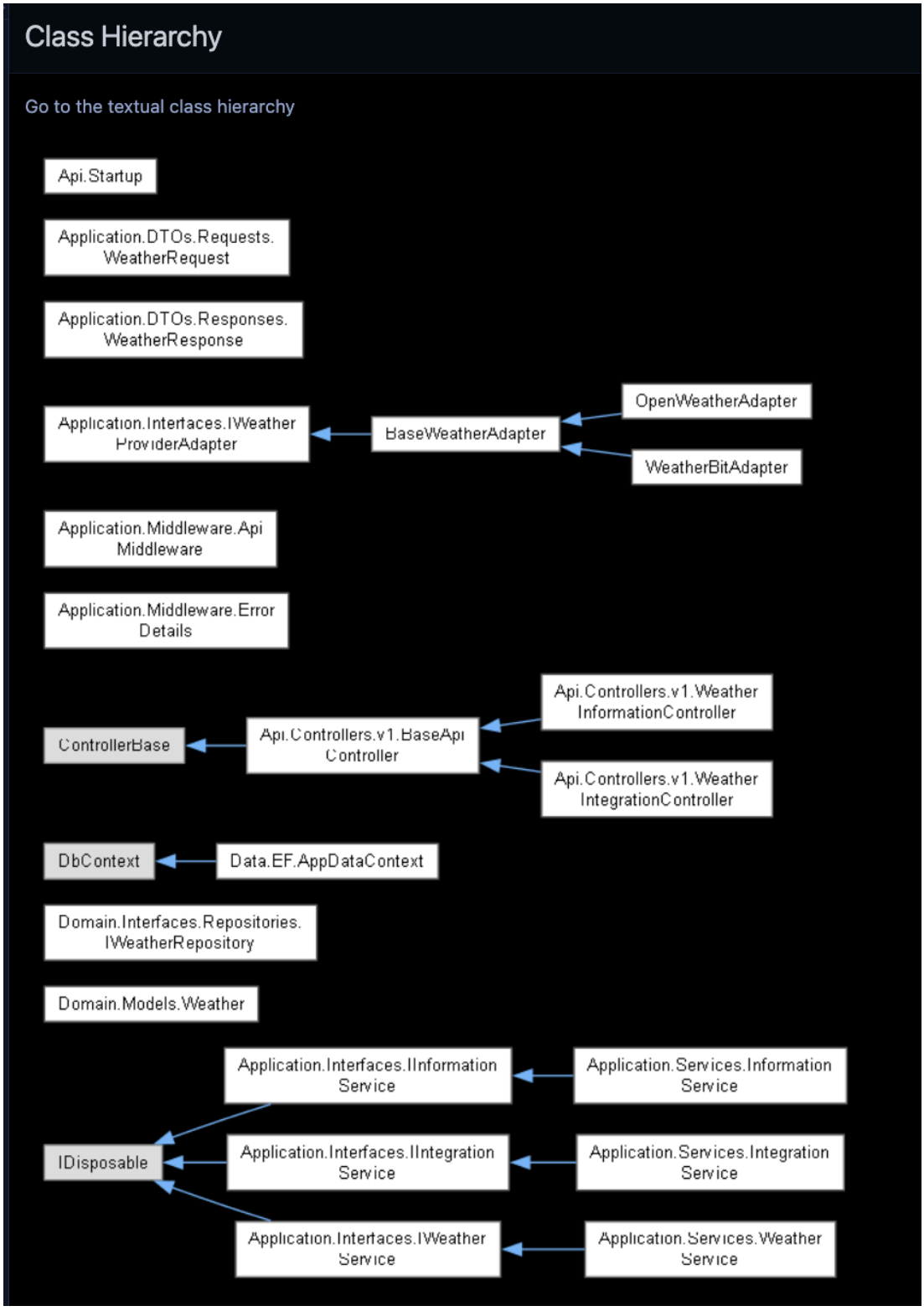
```
2 references
public Task UpdateWeatherReportAsync(Weather weather)
{
    dataContext.Set<Weather>().Update(weather);
    return Task.CompletedTask;
}

2 references
public async Task AddOrUpdateWeatherReportAsync(Weather weather)
{
    var existing = await GetWeatherReportByProviderAndLocation(weather.Provider, weather.City, weather.State, weather.Country);

    if (existing is null)
    {
        await AddWeatherReportAsync(weather);
    }
    else
    {
        existing.CelsiusTemperatureMin = weather.CelsiusTemperatureMin;
        existing.CelsiusTemperatureMax = weather.CelsiusTemperatureMax;
        existing.LastUpdate = weather.LastUpdate;

        await UpdateWeatherReportAsync(existing);
    }
}

2 references
public async Task SaveChangesAsync()
    => await dataContext.SaveChangesAsync();
```

Conclusões

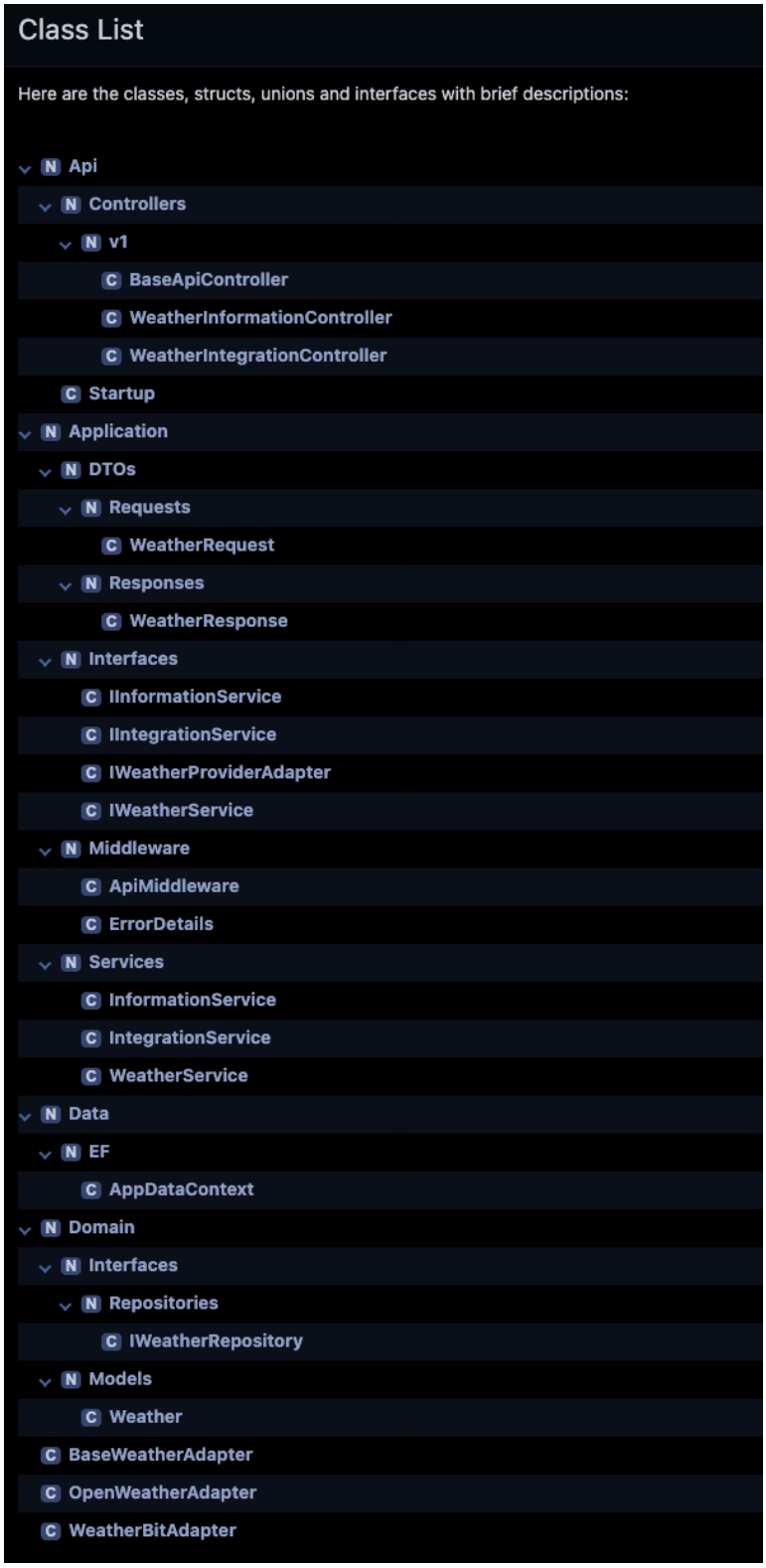
O repository permite a troca de mecanismo de persistência sem afetar a aplicacão, além de facilitar testes e mocks. A desvantagem nesse caso é o esforço necessário para definir métodos de repository, vale a pena em projetos maiores pela organização, mas pode ser overskill para códigos muito simples.

6. DIAGRAMAS E MODELOS

Diagrama de Classe Hierárquico:



Lista de Classes:



Lista de Namespaces:

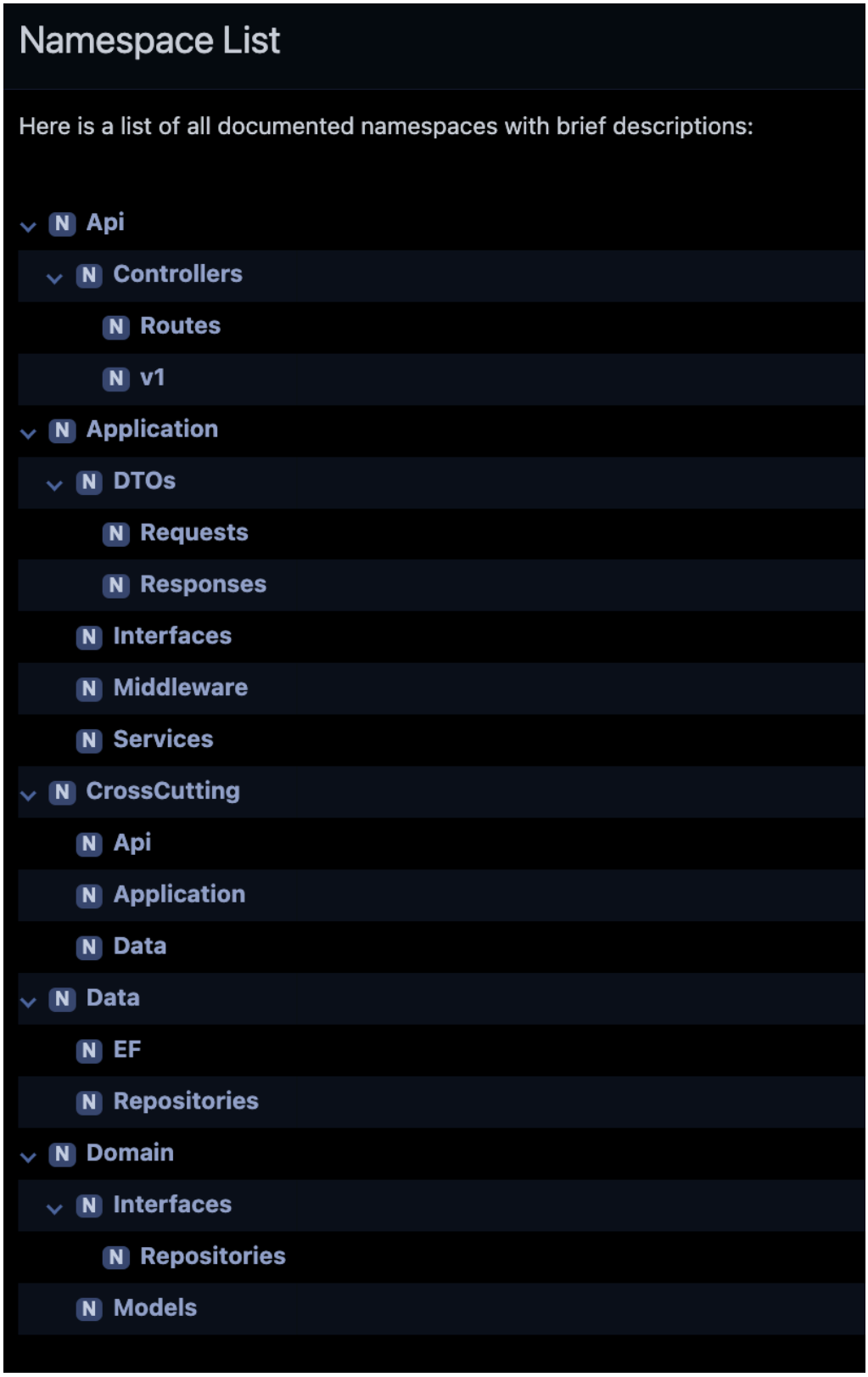
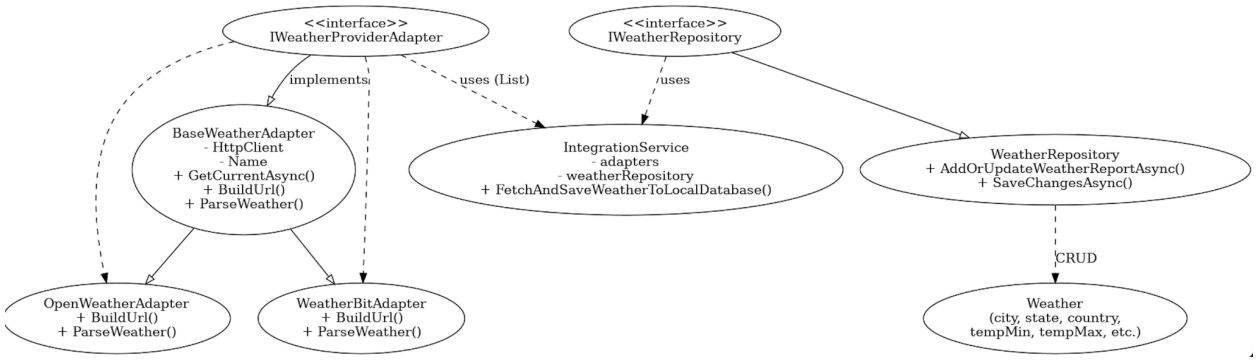


Diagrama de Classes:



7. CONCLUSÕES

A implementação do projeto demonstrou de forma clara e eficaz os benefícios da adoção de princípios sólidos de engenharia de software, especialmente por meio do uso dos princípios SOLID, da Clean Architecture e de padrões de projeto como Adapter, Template Method e Strategy.

Os princípios SOLID foram aplicados com consistência, resultando em um sistema altamente coeso e de baixo acoplamento. A divisão de responsabilidades, a inversão de dependências e a programação orientada a abstrações contribuíram diretamente para a manutenibilidade e testabilidade do código. A Clean Architecture, por sua vez, estruturou o projeto em camadas bem definidas, assegurando independência entre regras de negócio e implementações externas, o que facilitou tanto a evolução do sistema quanto a substituição de componentes sem impactos estruturais.

O uso do padrão Adapter permitiu integrar múltiplos provedores de dados meteorológicos de forma padronizada, promovendo extensibilidade. O Template Method foi fundamental para evitar duplicação de código e garantir um fluxo uniforme de requisições HTTP, enquanto o Strategy favoreceu a seleção dinâmica de provedores em tempo de execução, elevando a flexibilidade da solução.

De maneira geral, as práticas adotadas demonstraram-se altamente efetivas, oferecendo um projeto modular, escalável e alinhado com boas práticas modernas de desenvolvimento. Como recomendação para trabalhos futuros, propõe-se a adoção de mecanismos de resiliência (como retry policies e circuit breakers) e a automatização da validação de contratos externos, de forma a ampliar a robustez e a confiabilidade da aplicação em ambientes de produção.

8. REFERÊNCIAS BIBLIOGRÁFICAS

SOMMERVILLE, I. *Software Enfineerings Singapore,* New York: McGraw-Hill, 1983. Acesso em: 5 de junho de 2025.

GALLOTTI, Giocondo Marino Antonio (org.). *Arquitetura de software*. São Paulo: Pearson Education do Brasil, 2016. Acesso em: 5 de junho de 2025.

WEATHERBIT. Weatherbit API Dashboard. Disponível em: https://www.weatherbit.io/account/login?next=%2Faccount%2Fdashboard. Acesso em: 6 junho 2025.

OPENWEATHERMAP. Weather API - OpenWeather. Disponível em: https://openweathermap.org/api. Acesso em: 6 junho 2025.

REFACTORING.GURU. Padrão de Projeto Adapter. Disponível em: https://refactoring.guru/pt-br/design-patterns/adapter. Acesso em: 7 junho 2025.

REFACTORING.GURU. Template Method em Java. Disponível em: https://refactoring.guru/design-patterns/template-method. Acesso em: 8 junho 2025.

REFACTORING.GURU. Strategy em Java. Disponível em: https://refactoring.guru/design-patterns/strategy. Acesso em: 8 junho 2025.

9. Resumo Capítulos do Livro

1. Objetivo da atividade

Nesta atividade, foi proposta a leitura do Módulo 1 do livro ‘Arquitetura de Software’, do autor Giocondo Marino Antonio Gallotti. A partir da leitura, deve ser feito um breve resumo das arquiteturas descritas no módulo.

2. Conceitos de Software

Neste primeiro tópico de Conceitos Arquiteturais, o autor introduz o assunto com a seguinte pergunta ao leitor: ‘o que é um software?’, pois, para entendermos como é a sua arquitetura, primeiro devemos compreender a sua definição.

O software apresenta-se como um elemento intermediário entre o ser humano e a máquina. Essa intermediação é feita por meio do processamento de dados, que podem ser de qualquer origem, desde que esses dados possam ser convertidos para um formato que o computador consiga entender.

Além disso, a interação entre o usuário e o software ocorre através da chamada ‘interface homem-máquina’. Porém, o software que usamos, como um editor de texto, não “conversa” diretamente com o processador do computador. Essa comunicação é feita com o sistema operacional, que é o responsável por traduzir as informações recebidas, convertendo à linguagem de máquina, que assim o computador é capaz de entender.

Por último, é pontuado o funcionamento do software em si, que é a execução de algoritmos, esses que são séries de instruções que, para que uma tarefa seja atendida, devem seguir uma sequência correta.

3. Função e Desempenho

Para que possamos entender a organização de um software, podemos pensar que ele é dividido em dois elementos: função e desempenho. Estes elementos estão presentes desde o início de desenvolvimento do software e estão diretamente relacionados, mas essa relação pode variar dependendo do caso.

A função é a forma como o software manipula os dados, geralmente seguindo processos. Na parte da função, o desempenho não é crucial, apesar de ser importante que os processos aconteçam rapidamente. Por outro lado, ao tratar-se de softwares que controlam vários processos simultaneamente, o desempenho é um fator essencial no funcionamento dos programas.

Portanto, devemos buscar manter o equilíbrio entre função e desempenho, através de escolher e criar componentes que ajudem durante o desenvolvimento.

4. Fases do Desenvolvimento do Software

Neste tópico, para entendermos o desenvolvimento de software, ele é dividido em três partes, são elas: definição; desenvolvimento; verificação, liberação e manutenção.

4.1 Definição

Nessa fase, o principal objetivo é planejar o desenvolvimento antes de qualquer codificação. Essa etapa inicia-se com a criação de um plano de projeto, no qual será definido o propósito do software, problemas que ele deve resolver, suas soluções e quais necessidades ele atenderá, destacando casos de encomendas.

Ademais, uma análise de riscos é feita, considerando a possibilidade de riscos durante o desenvolvimento. Também é feito um levantamento de recursos, como equipe, prazos, custos e materiais. Essa parte é essencial para avaliar a viabilidade e compensação do projeto a ser executado.

Em seguida, ocorre o levantamento de requisitos, onde é detalhado as funcionalidades do software com a ajuda de protótipos, diagramas e pseudocódigos. Nessa fase, o cliente pode participar com o intuito de garantir que suas necessidades estão sendo compreendidas corretamente.

Por último, o documento de especificação de requisitos do software é criado, no qual a definição do que o sistema deve ou não deve dispor. Esse documento é revisado pelos envolvidos no projeto e servirá de base para a próxima fase.

4.2 Desenvolvimento

Na fase de desenvolvimento, o trabalho começa na análise dos documentos preparados na fase de definição. Com base nisso, é dado início à criação do algoritmo, a definição da estrutura do sistema e interfaces, sempre levando como referência o plano de projeto a fim de garantir a qualidade do que é feito. Tudo que é construído é documentado em uma especificação de projeto, onde são descritas as configurações do software e os procedimentos adotados para cada módulo.

Em seguida, é feita a codificação, ou seja, a etapa na qual o programa é escrito utilizando uma linguagem de programação ou ferramentas CASE. Destaca-se também o fato de que um bom código depende de um bom projeto, por isso ele deve ser claro, objetivo e bem estruturado para garantir concordância com as especificações determinadas. Por fim, é gerada a linguagem-fonte do sistema, separada por módulos.

4.3 Verificação, liberação e manutenção

Nesta última fase, o software passa por testes rigorosos com o objetivo de identificar erros de funcionalidade e de desempenho. Os testes primeiramente são feitos nos módulos separadamente, depois, na integração entre eles, e por último é feita uma validação geral do sistema. Em caso de falhas, é feito o debugging para correção.

Antes da liberação, ocorre uma checagem de qualidade que confere se toda a documentação está correta. E finalmente o software é entregue aos usuários finais, mas o trabalho continua através da manutenção do sistema, corrigindo problemas encontrados, adaptando e melhorando o produto.

5. Componentes e Conectores

O tópico introduz a explicação de que a arquitetura de software é a estrutura interna de um sistema, ou seja, é a forma como o sistema se organiza e funciona. Porém, essa arquitetura dita não é estática, ela pode evoluir e mudar durante o desenvolvimento, seja por feedbacks, novas funções e correções.

O objetivo é que, além de ser clara e simples, a arquitetura tenha quatro características fundamentais:

- Flexibilidade: pois permite fazer alterações, melhorias e correções sem a necessidade de refazer grandes partes do projeto.
- Extensibilidade: para facilitar a incorporação de novos elementos e novas features.
- Portabilidade: execução do sistema em diferentes plataformas.
- Reutilização: permite usar a arquitetura desenvolvida em outros softwares. O autor apresenta um famoso modelo de arquitetura de software, proposto por Dewayne E. Perry e Alexander L. Wolf, chamado ‘Fundamentos para o estudo da arquitetura de software’, onde é expressa a seguinte fórmula: ‘Arquitetura = (Elementos + Organização + Decisões)’. Esses ‘elementos’ podem ser divididos em três tipos:
- Elementos de processamento (operam os dados).
- Elementos de dados (matéria-prima a ser processada).
- Elementos de conexão (amarram e conectam os outros elementos para que tudo funcione em conjunto.

6. Configurações e Padrões Arquiteturais

Neste tópico, o autor pontua que, antes de conhecer os padrões arquiteturais, é importante entender que cada um deles possui suas características, suas vantagens e desvantagens. Portanto, desenvolvedores precisam ter clareza sobre seus objetivos a fim de escolher o padrão mais adequado para seus projetos.

Um padrão arquitetural, de forma geral, descreve uma relação entre organização e elementos que já deram certo em projetos finalizados. Ele é descrito de forma abstrata, usando tabelas e diagramas.

6.1 Arquitetura MVC

Essa arquitetura é um padrão bastante presente em sistemas Web. Ele organiza o código de um sistema em três partes principais que se comunicam entre si. São os componentes: modelo (gerenciamento do sistema de dados e operações associadas), visão (como os dados são apresentados ao usuário) e controlador (gerenciamento da interação do usuário).

Ele é usado principalmente em sistemas onde existem múltiplas formas de visualizar e interagir com os mesmos dados.

Como vantagens, é possível citar a independência entre os componentes, pois os dados podem ser alterados sem afetar a forma como são exibidos.

Já desvantagens, em um sistema no qual as interações e o modelo de dados são muito simples, a implementação do MVC pode adicionar uma complexidade desnecessária.

6.2 Arquitetura em camadas

Nessa arquitetura, o sistema é organizado em camadas, onde cada uma oferece serviços para a camada imediatamente acima dela. Sendo assim, as camadas mais baixas contém os serviços mais gerais e fundamentais, que podem ser utilizados por todo o sistema.

Ela é utilizada em algumas situações, como: construção de novas funcionalidades em sistemas já existentes; desenvolvimento distribuído em equipes, com responsabilidades separadas; proteção multinível.

As vantagens dessa arquitetura se destacam na confiabilidade (adição de recursos de segurança em cada camada) e o fato de que é possível substituir uma camada inteira sem afetar o resto do sistema.

Já em desvantagens, é possível citar a difícil separação das responsabilidades de cada camada, a violação de camadas que pode quebrar a organização do padrão, e o desempenho, pois uma solicitação precisará passar e ser interpretada por múltiplas camadas.

6.3 Arquitetura de repositório

Na arquitetura de repositório, todos os dados do sistema são armazenados e gerenciados em um repositório central, logo os componentes não interagem entre si, somente com o repositório.

Esse padrão é utilizado principalmente em sistemas que trabalham com grandes volumes de informações que precisam ser armazenadas por um longo tempo.

Suas vantagens destacam a independência dos componentes e o gerenciamento centralizado.

Suas desvantagens sinalizam que esse padrão possui um ponto único de falha, pois, se o repositório falhar, o sistema inteiro será afetado, a ineficiência na comunicação e a dificuldade de distribuição.

6.4 Arquitetura cliente-servidor

Nessa arquitetura, a funcionalidade do sistema está organizada em serviços, onde cada serviço é prestado por um servidor. Os clientes são os usuários dos serviços e acessam os servidores para uso.

Ela é utilizada quando há a necessidade de acessar os dados do banco por uma série de locais.

A principal vantagem desse modelo é a possibilidade de distribuição dos servidores através de uma rede.

Já como desvantagens é possível citar a imprevisibilidade por causa da dependência da rede, problemas de gerenciamento por diferentes organizações e o fato de que cada serviço é um ponto único de falha.

6.5 Arquitetura de duto e filtro

Por último, a arquitetura de duto e filtro é caracterizada pela organização do processamento de dados, o qual cada componente de processamento (filtro) realiza um tipo de transformação de dados, assim os dados fluem como em um duto de um componente para outro.

Esse padrão é usado comumente em aplicações de processamento de dados em que as entradas são processadas em etapas separadas para gerarem saídas relacionadas.

As vantagens dessa arquitetura são o reuso da transformação dos dados, que é de fácil entendimento.

As desvantagens citam que o formato para as transferências de dados tem de ser acordado, o que aumenta o overhead do sistema e pode impossibilitar reuso de transformações devido a estruturas incompatíveis de dados.

10. REPOSITÓRIO GITHUB

https://github.com/Silvestrinih03/weather-padroes-arquitetura/tree/main