FluentArch: uma abordagem moderna para verificação de conformidade arquitetural em C#

Arthur Castro
Departamento de Ciência da Computação
Universidade Federal de Lavras
Lavras, Brasil
arthur.castro@estudante.ufla.br

Ricardo Terra
Departamento de Ciência da Computação
Universidade Federal de Lavras
Lavras, Brasil
terra@ufla.br

RESUMO

Sistemas de software com longos ciclos de vida estão sujeitos à erosão arquitetural, comprometendo sua manutenibilidade e qualidade. No ecossistema C#, ferramentas como NetArchTest.eNhancedEdition e ArchUnitNET auxiliam na verificação de conformidade arquitetural, mas apresentam duas limitações importantes: (i) operam sobre o *bytecode* compilado, restringindo a análise a construções pós-compilação e (ii) oferecem expressividade limitada na definição de regras personalizadas. Este trabalho apresenta o `FluentArch`, uma API fluente baseada no compilador Roslyn que permite a definição e verificação de regras arquiteturais diretamente sobre o código-fonte. A abordagem possibilita uma análise com granularidade mais fina, identificação precisa de dependências e suporte à criação de regras customizadas, superando restrições das soluções existentes. O uso prático da ferramenta em um projeto *open source* de terceiros evidenciou sua efetividade na detecção de violações arquiteturais e sua aplicabilidade em cenários reais, reforçando seu potencial como alternativa moderna e flexível para verificação arquitetural em sistemas C#.
Demo vídeo: https://doi.org/10.6084/m9.figshare.29139881

PALAVRAS-CHAVE

Conformidade arquitetural, Interface fluente, C#.

1 Introdução

Arquitetura de software é geralmente definida como um conjunto de decisões de projeto que têm impacto em diversos aspectos da construção e evolução de sistemas de software [17]. Entretanto, alterações realizadas por pessoas que não compreendem as decisões arquiteturais originais quase sempre causam a degradação do sistema [17]. Com o tempo, o software começa a infringir os conceitos definidos em sua concepção, fazendo com que a arquitetura atual se distancie daquela idealizada inicialmente [13].

Com o avanço e popularização de técnicas de análise estática e dinâmica de código [3, 9], ferramentas vêm sendo criadas no intuito de usar tais técnicas para domínios específicos. Por exemplo, SonarQube [19] e NDepend [15] para inspeções da qualidade e de aspectos de segurança de projetos. Já, no contexto deste estudo, algumas ferramentas oferecem APIs (*Application Programming Interfaces*) que permitem esse tipo de análise, como o Roslyn[1], API oficial do compilador C#, e o Mono.Cecil[2], mantido com apoio da Microsoft.

O compilador Roslyn fornece uma API rica para análise estática diretamente sobre o código-fonte, permitindo a navegação pela árvore de sintaxe abstrata (AST) e o acesso a informações semânticas detalhadas. Essa abordagem viabiliza análises precisas e de alta granularidade, mas exige maior esforço de implementação e uma curva de aprendizado mais elevada. Em contrapartida, o Mono.Cecil atua sobre o *bytecode* compilado, facilitando a integração com testes e *pipelines*, porém limitado àquilo que é mantido após a compilação, o que restringe a análise de elementos exclusivamente sintáticos, como convenções de nomenclatura e atributos decorativos.

Este artigo apresenta o `FluentArch`, uma abordagem moderna para verificação de conformidade arquitetural em projetos C#. A ferramenta oferece uma API fluente [5, 18] que simplifica a definição e verificação de regras arquiteturais, abstraindo a complexidade da API do compilador Roslyn. Diferentemente de soluções baseadas em Mono.Cecil [4, 16], que operam sobre o *bytecode* e se limitam a informações pós-compilação, o `FluentArch` realiza análises diretamente no código-fonte, permitindo maior precisão e granularidade. A API possibilita a criação de camadas lógicas e regras personalizadas, tornando a especificação arquitetural mais expressiva e adequada a diferentes estilos e cenários.

A viabilidade do `FluentArch` foi demonstrada por meio de sua aplicação no sistema *open source* N-Tier-Architecture, baseado na arquitetura *N-layer*. Foram definidas e verificadas regras de dependência entre camadas, restrições de instanciação e herança, além da proibição de métodos em determinadas classes. Como resultado, a ferramenta foi capaz de identificar 27 violações arquiteturais, incluindo dependências implícitas em testes e classes alocadas indevidamente. Os achados reforçam a capacidade do `FluentArch` em detectar violações relevantes e apoiar a manutenção da arquitetura planejada.

O restante do artigo está organizado como a seguir. A Seção 2 introduz ferramentas similares já existentes no mercado, bem como suas principais limitações. A Seção 3 descreve o `FluentArch` e a Seção 4 discute o seu projeto e implementação. A Seção 5 aplica a ferramenta em um projeto *open source* de terceiros. Por fim, a Seção 6 conclui e apresenta trabalhos futuros.

2 Ferramentas relacionadas

Esta seção introduz as duas principais ferramentas de código aberto e com interfaces fluentes relacionadas ao `FluentArch`.[3] As Seções 2.1 e 2.2 descrevem, respectivamente, as ferramentas NetArchTest.eNhancedEdition e ArchUnitNET enquanto a Seção 2.3 destaca as principais limitações compartilhadas entre essas ferramentas.

[1] https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/
[2] https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/

[3] O NDepend não foi incluído por não utilizar interface fluente, mas sim, CQLinq [14].

2.1 NetArchTest.eNhancedEdition

A biblioteca NetArchTest.eNhancedEdition é uma evolução da original NetArchTest. A nova versão corrigiu limitações existentes e encontra-se em manutenção ativa, com atualizações frequentes e novos recursos.

A biblioteca utiliza o *Mono.Cecil* para carregar os *assemblies* compilados do projeto, o que permite a análise da estrutura de tipos diretamente sobre o *bytecode* gerado pela compilação. Tal abordagem viabiliza a execução das validações em conjunto com os testes de unidade, favorecendo sua integração a *pipelines* de CI/CD. No entanto, limita-se às informações presentes apenas após a compilação, não contemplando elementos declarativos e sintáticos exclusivos do código-fonte.

A API permite a definição de regras por meio da composição de predicados, condições e conjunções. Na Listagem 1, é ilustrada uma regra arquitetural para garantir que a camada *Services* não dependa da camada *Controllers*. Entre as linhas 4 e 8, os tipos localizados no *namespace* `MyApp.Services` são selecionados pelo predicado `ResideInNamespace`. A condição dessa regra é definida entre as linhas 9 e 10 pela sequência de métodos: `ShouldNot` e `HaveDependencyOn` que validam a ausência de dependência com o módulo de *namespace* `MyApp.Controllers`. O método `GetResult` escrito na linha 11 gera o resultado da análise, verificado por meio do `Assert`.

Listagem 1 Exemplo com NetArchTest.eNhancedEdition

```
1  [Test]
2  public void Services_ShouldNotReference_Controllers()
3  {
4      var result = Types
5          .InAssembly( typeof(MyApp.Services).Assembly)
6          .That()
7          .ResideInNamespace("MyApp.Services")
8          .ShouldNot()
9          .HaveDependencyOn("MyApp.Controllers")
10         .GetResult();
11
12     Assert.IsTrue(result.IsSuccessful);
13 }
```

2.2 ArchUnitNET

Inspirada na biblioteca ArchUnit para Java [2], a ArchUnitNET foi desenvolvida com o objetivo de oferecer funcionalidades equivalentes no contexto C# [4]. Assim como a NetArchTest.eNhancedEdition, permite a definição de regras por meio de testes de unidade, utilizando o Mono.Cecil para análise de *bytecode*.

A Listagem 2 ilustra o uso da biblioteca para criar um teste com a mesma regra definida na NetArchTest.eNhancedEdition, classes do *namespace* `MyApp.Services` não devem poder depender de classes do *namespace* `MyApp.Controllers`. Essa biblioteca já apresenta o conceito de camadas, possibilitando definir como um conjunto de classes, baseadas no *namespace*, possibilitando o reúso em outras partes do código, como demonstrado sua definição entre as linhas 9 e 17 e seu uso nas linhas 24 e 26.

A API segue o modelo baseado em predicados, condições e conjunções, similar ao da NetArchTest.eNhancedEdition. No entanto, oferece funcionalidades adicionais como:

- Regras de dependência entre módulos e classes;

Listagem 2 Exemplo com ArchUnitNET

```
1  private static readonly Architecture Architecture =
2      new ArchLoader().LoadAssemblies(
3          System.Reflection.Assembly
4              .Load("ExampleClassAssemblyName"),
5          System.Reflection.Assembly
6              .Load("ForbiddenClassAssemblyName")
7  ).Build();
8
9  private readonly IObjectProvider<IType> ServiceLayer =
10     Types().That()
11     .ResideInNamespace("MyApp.Services")
12     .As("Example Layer");
13
14 private readonly IObjectProvider<IType> ControllerLayer =
15     Types().That()
16     .ResideInNamespace("MyApp.Controllers")
17     .As("Forbidden Layer");
18
19 [Fact]
20 public void ExampleLayerShouldNotAccessForbiddenLayer()
21 {
22     IArchRule exampleLayerShouldNotAccessForbiddenLayer =
23         Types().That().Are(ServiceLayer)
24         .Should()
25         .NotDependOnAny(ControllerLayer)
26         .Because("it's forbidden");
27
28     exampleLayerShouldNotAccessForbiddenLayer
29         .Check(Architecture);
30 }
```

- Verificação de convenções de nomenclatura;
- Restrições por *namespace*;
- Controle de acesso a atributos;
- Detecção de dependências cíclicas; e
- Geração e validação de diagramas UML a partir do código.

Apesar das funcionalidades avançadas, a biblioteca possui limitações quanto à criação de regras personalizadas. Usuários que necessitam de maior expressividade ou cenários específicos ficam restritos às condições predefinidas da biblioteca, o que pode dificultar a aplicação em arquiteturas mais complexas.

2.3 Limitações compartilhadas

Ambas as ferramentas analisam o *bytecode* e não o código-fonte original (`.cs`), o que impõe restrições como determinadas construções da linguagem C# não serem analisadas corretamente, incluindo uso de `nameof`, conversões dinâmicas e expressões *lambda*.

Além disso, operam com uma granularidade relativamente grossa ao identificar dependências. Isso impede, por exemplo, a diferenciação entre tipos específicos de relacionamentos, tais como:

- Acesso a atributos ou métodos;
- Instanciação de objetos;
- Herança e implementação de interfaces;
- Ativação de exceções; e
- Uso de anotações (*attributes*).

Essas limitações reduzem a aplicabilidade das ferramentas em contextos que demandam validações de maior granularidade, comuns em arquiteturas robustas e evoluídas [8]. Ademais, como ambas ferramentas não oferecem suporte à reutilização de regras arquiteturais, a especificação dessas regras pode se tornar uma atividade

repetitiva, elevando o esforço necessário para verificar múltiplos projetos com os mesmos critérios arquiteturais [7].

3 FluentArch

Considerando as limitações compartilhadas pelas ferramentas existentes, o FluentArch foi desenvolvido como uma API fluente capaz de realizar análises com granularidade mais fina sobre as dependências entre elementos de sistemas C#. Ao contrário das abordagens que operam exclusivamente sobre o *bytecode*, o FluentArch realiza a análise diretamente sobre o código-fonte (.cs), por meio do Roslyn — compilador oficial da linguagem C# e plataforma de análise de código mantida pela Microsoft [11]. Essa estratégia possibilita a identificação mais precisa e detalhada das relações entre módulos, incluindo dependências sutis que não são preservadas após a compilação.

Por exemplo, a regra que estabelece que uma classe da camada de serviço não deve depender da camada de controle pode ser expressa de forma clara e concisa com o uso do FluentArch, conforme ilustrado na Listagem 3.

Listagem 3 Exemplo de verificação arquitetural no FluentArch

```
1  var arch = Architecture.Build(solution);
2  ILayer camadaControle = arch.All()
3      .ResideInNamespace("MyApp.Controllers.*");
4  ILayer camadaServico = arch.All()
5      .ResideInNamespace("MyApp.Services.*")
6      .And()
7      .HaveNameEndingWith("Service");
8
9  var violations = camadaServico
10     .Cannot().Depend(camadaControle)
11     .Check();
```

O FluentArch atua como um intermediário entre o modelo de código exposto pelo Roslyn, com sua complexa estrutura sintática e semântica, e o programa desenvolvido pelo engenheiro de software para garantir a conformidade arquitetural. Dessa forma, o mantenedor tem liberdade para escrever regras utilizando restrições predefinidas ou definir suas próprias regras customizadas.

A API foi concebida com foco em simplicidade, legibilidade e flexibilidade, superando limitações observadas em ferramentas existentes. Para isso, foram introduzidos conceitos como camadas reutilizáveis, regras customizadas e granularidade fina na detecção de dependências, com uma leitura fluente e expressiva das regras arquiteturais.

3.1 Estrutura

Para iniciar a escrita das regras, é necessário carregar a Solution, objeto da biblioteca Microsoft.CodeAnalysis [12]. Esse objeto é responsável por conter todos os projetos com extensão .cs e suas dependências mútuas. É importante destacar que apenas uma instância de Solution pode ser carregada por execução do programa.

Seguindo uma organização comumente adotada por outras bibliotecas baseadas em interfaces fluentes voltadas à validação arquitetural, a definição de regras no FluentArch é estruturada nos seguintes três componentes principais:

Filtros têm como função selecionar os elementos do sistema que serão analisados, permitindo o agrupamento lógico de tipos em camadas, que correspondem a módulos arquiteturais. A API oferece recursos para esse propósito, como filtros baseados em nome ou *namespace* (e.g., linhas 3, 5 e 7 da Listagem 3).

Condições estabelecem os critérios que os elementos filtrados devem satisfazer. A biblioteca disponibiliza métodos para verificar relações de dependência entre módulos e também possibilita a criação de condições personalizadas, adaptadas às necessidades do projeto (e.g., linha 10 da Listagem 3).

Conectores permitem a composição de filtros e condições por meio de operadores lógicos e sinalizam o encerramento da regra. Métodos como And e Check são empregados para concatenar instruções e avaliar a conformidade do código com a regra definida (e.g., linhas 6 e 11 da Listagem 3).

Embora o FluentArch apresente um conjunto mais enxuto de métodos predefinidos para definição de módulos e condições, quando comparado às demais bibliotecas analisadas, a abordagem se destaca por oferecer maior extensibilidade ao usuário por meio do suporte à criação de regras customizadas.

A configuração mínima de uma regra arquitetural é composta por: (i) a definição de um filtro para identificar os tipos a serem considerados e (ii) a especificação de uma condição que expressa a restrição arquitetural. Opcionalmente, pode-se utilizar do conector Check, responsável por verificar a conformidade do código-fonte.

3.2 Regras preestabelecidas

O FluentArch oferece suporte a quatro tipos principais de regras arquiteturais, cuja semântica é suficientemente expressiva para abranger a maioria das restrições comumente especificadas no processo de verificação de conformidade arquitetural [13, 20]:

Divergência: ocorre quando uma dependência existente no código viola uma restrição definida. Como pode ser observado na Listagem 4, podem ser especificadas em três semânticas distintas:

- Somente o módulo *A* pode depender do módulo *B* (linha 1);
- O módulo *A* pode depender somente do módulo *B* (linha 2); e
- O módulo *A* não pode depender do módulo *B* (linha 3).

Ausência: ocorre quando o código não estabelece uma dependência obrigatória. Pode ser especificado na seguinte semântica:

- O módulo *A* deve depender do módulo *B* (linha 4).

Listagem 4 Tipos de regras arquiteturais no FluentArch

```
1  moduloA.OnlyCan().Depend(moduloB).Check();
2  moduloA.CanOnly().Depend(moduloB).Check();
3  moduloA.Cannot().Depend(moduloB).Check();
4  moduloA.Must().Depend(moduloB).Check();
```

Além disso, é possível especificar o tipo de dependência analisado por meio dos operadores abaixo:

- *Access*: acesso a atributos ou métodos de outro módulo;
- *Declare*: declaração explícita ou implícita de tipos;
- *Create*: criação de objetos;
- *Extend*: herança de classes;
- *Implement*: implementação de interfaces;
- *Throw*: lançamento de exceções;
- *Handle*: *alias* para *Access* ou *Declare*;
- *Derive*: *alias* para *Extend* ou *Implement*; e
- *Depend*: verificação ampla, i.e., todas as anteriores.

Conforme ilustrado na Listagem 5, as regras das linhas 1-4 e da linha 5 fazem a mesma restrição arquitetural utilizando granularidade mais fina, i.e., impedem o módulo *A* de acessar e declarar tipos do módulo *B*.

Listagem 5 Regras com granularidade fina e uso de operador *alias*

```
1 moduloA.Cannot().Access(moduloB)
2         .And()
3         .Cannot().Declare(moduloB)
4         .Check();
5 moduloA.Cannot().Handle(moduloB).Check();
```

3.3 Camadas lógicas

No contexto do `FluentArch`, o conceito de camadas refere-se a agrupamentos lógicos de tipos do sistema. Essa abordagem permite definir módulos independentes da organização física dos arquivos, o que é especialmente útil em sistemas afetados por erosão arquitetural, nos quais o agrupamento físico pode não refletir a estrutura lógica das classes.

A Listagem 6 ilustra um exemplo de definição de uma camada que agrupa todas as classes que compõem o padrão *Repository*. A definição da camada é composta por filtros e conjunções lógicas (linha 4). No código apresentado, cria-se uma camada aplicando um filtro de *namespace* (linha 3) e do padrão do nome da classe (linha 5).

As camadas lógicas podem ser utilizadas em múltiplas regras arquiteturais, contribuindo para a manutenção e clareza na definição das restrições.

Listagem 6 Exemplo de uma camada no `FluentArch`

```
1 var arch = Architecture.Build(solution);
2 ILayer camadaRepositorio = arch.All()
3     .ResideInNamespace("Repositorios.*")
4     .And()
5     .HaveNameEndingWith("Repositorio");
```

3.4 Regras customizadas

O `FluentArch` tem como propósito principal facilitar e flexibilizar a criação de validadores arquiteturais. Para isso, a ferramenta não se limita às regras predefinidas, permitindo que o próprio engenheiro de software defina suas próprias condições, conforme as necessidades específicas de seu projeto.

Para criar uma regra customizada, é necessário implementar a interface `ICustomRule` e definir a lógica de validação no método `DefineCustomRule`. Esse método recebe um objeto que representa a entidade analisada e oferece acesso às informações extraídas da análise estática da AST do código-fonte.

Para viabilizar o uso de regras customizadas, durante a definição de uma regra arquitetural, a ferramenta itera sobre cada tipo presente na camada previamente definida e executa o método `DefineCustomRule`. Esse procedimento permite a verificação da condição especificada na regra customizada.

A Listagem 7 exemplifica a definição de uma regra que proíbe a existência de funções nas estruturas analisadas. Essa condição pode ser útil, por exemplo, para impor restrições ao módulo de DTOs, que deve servir apenas para transferência de dados entre camadas.

Listagem 7 Regra customizada para tipos sem funções

```
1 public class TypeCannotHaveFunctions : ICustomRule
2 {
3     public bool DefineCustomRule(TypeEntityDto type)
4     {
5         return !type.Functions.Any();
6     }
7 }
```

A Listagem 8 apresenta como a regra customizada é utilizada para realizar a verificação arquitetural. Essa capacidade de definir regras específicas amplia significativamente o poder expressivo do `FluentArch`, adaptando-o a diferentes tipos de validação. Por exemplo, (i) validações baseadas em métricas simples como LOC até métricas com cálculos complexos de coesão e acoplamento, (ii) validações baseadas em análises textuais, (iii) validações baseadas em retorno de análises de *Large Language Models* (LLMs), etc.

Listagem 8 Conformidade arquitetural de uma regra customizada

```
1 var arch = Architecture.Build(solution);
2 arch.All()
3     .ResideInNamespace("DTOs")
4     .UseCustomRule(new TypeCannotHaveFunctions())
5     .Check();
```

Embora para as condições preestabelecidas, o retorno inclui informações detalhadas sobre cada violação, como o caminho físico do arquivo e a linha exata em que a infração ocorreu, o `FluentArch` não fornece a linha exata da violação em caso de regras customizadas. Nesses casos, o retorno é restrito à identificação da classe responsável pela quebra da regra.

4 Projeto e implementação

Conforme ilustrado na Figura 1, a ferramenta `FluentArch` é organizada nos seguintes cinco principais módulos:

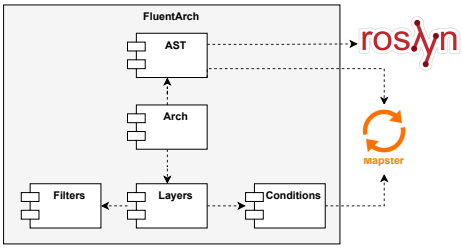

Figura 1: Arquitetura interna do `FluentArch`

O módulo `Arch` constitui o ponto de entrada da aplicação, sendo responsável por receber o caminho do projeto a ser analisado. Esse módulo centraliza todos os aspectos relacionados à arquitetura do projeto alvo e depende dos módulos `AST` e `Layers`.

O módulo `AST` concentra as classes responsáveis pela análise estática do código, com base na AST. Nesse módulo, encontram-se as classes do tipo *visitor*, encarregadas de percorrer a AST a fim de extrair informações essenciais para o funcionamento da ferramenta, como as dependências de uma determinada classe. Esses dados são organizados de forma a proporcionar uma abstração mais acessível ao usuário. Ressalta-se que esse módulo depende da biblioteca Roslyn, a qual viabiliza a análise da AST em projetos .NET,

bem como da biblioteca Mapster, responsável pelo mapeamento automático de dados.

O gerenciamento das estruturas de dados obtidas pelo módulo `AST` e disponibilizadas pelo módulo `Arch` é realizado pelo módulo `Layers`. Esse módulo é responsável por representar o agrupamento lógico de classes definido durante a análise. Ainda, suas classes atuam em conjunto com as classes do módulo `Filters`, o qual é responsável exclusivamente por definir critérios específicos que possibilitam a criação de camadas mais refinadas e direcionadas.

Por fim, após a definição das camadas, o módulo `Conditions` é acionado. Trata-se do módulo com o maior número de classes, em virtude da necessidade de manter classes especializadas para cada tipo de verificação (divergências ou ausências). Inclusive, é nesse módulo que se encontra a lógica responsável pela aplicação de regras customizadas. Assim como o módulo `AST`, o módulo `Conditions` também possui depende da biblioteca Mapster para o mapeamento automático de dados. Esse módulo representa a etapa final do processo, sendo responsável pela geração do resultado da verificação arquitetural.

As decisões arquiteturais influenciam diretamente a construção, o uso, a manutenção e a evolução de um projeto. Esta seção explica as decisões tomadas durante o desenvolvimento da biblioteca para que a comunidade possa contribuir com sua evolução.

Mono.Cecil ⟶ Roslyn: Como discutido nas seções anteriores, as ferramentas que realizam análise sobre o *bytecode* C# apresentam limitações quanto à granularidade e à flexibilidade das regras. Para contornar esse problema, é possível utilizar a API de análise estática do Roslyn. No entanto, essa abordagem exige uma curva de aprendizado mais acentuada e um tempo de desenvolvimento maior. Para mitigar essa dificuldade, o `FluentArch` foi projetado para utilizar o Roslyn de forma transparente ao usuário, disponibilizando os dados já processados de maneira acessível. Dessa forma, espera-se que o usuário possa se concentrar exclusivamente na definição das regras arquiteturais, sem precisar lidar diretamente com os detalhes da análise sintática.

***Design* estrutural**: Ao utilizar o método estático `Build`, da instância *Singleton*, a análise é executada automaticamente sobre todos os arquivos `.cs` da solução C#. A partir desse momento, durante a execução das regras — sejam elas predefinidas ou personalizadas — o `FluentArch` já terá carregado todas as informações necessárias, permitindo que essas regras utilizem os dados sem retrabalho ou complexidade adicional.

A escrita das regras utiliza o padrão *Builder* para a construção do objeto final. Os filtros e as condições definidas produzem uma instância de `ArchRule`, que contém todas as classes analisadas e os resultados correspondentes.

Bibliotecas: Ademais, o projeto possui dependência de outras duas bibliotecas: XUnit [22] e Mapster [10]. Os testes foram criados com o objetivo de garantir que as evoluções da biblioteca não impactem versões anteriores. O XUnit [22] foi utilizado para implementar esses testes, seguindo o padrão de escrita Organizar, Agir e Verificar (*Arrange*, *Act* e *Assert*). A biblioteca Mapster foi escolhida por ser *open source* e popular no ecossistema C# e .NET. Ela é responsável pelo mapeamento automático das classes, um processo que reduziu significativamente o tempo de desenvolvimento da ferramenta ao abstrair toda a lógica de mapeamento.

5 Exemplo de aplicabilidade da ferramenta

Para demonstrar a viabilidade da ferramenta e ainda oferecer um guia concreto para compreendê-la, esta seção apresenta sua aplicação sobre o projeto N-Tier-Architecture [1], uma aplicação *open source* de terceiros que implementa a arquitetura *N-layer* em uma Web API. A escolha desse sistema deve-se, em parte, à presença de regras de dependência intra-modulares explicitamente documentadas. Conforme ilustrado na Figura 2, tais regras estabelecem uma direção de dependência descendente entre as camadas, de modo que uma camada inferior não deve depender de camadas superiores.

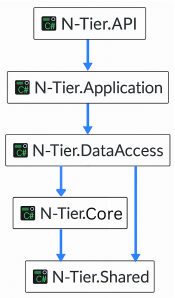

Figura 2: Regras entre camadas do N-Tier-Architecture

A Listagem 9 ilustra como as regras arquiteturais do sistema podem ser expressas utilizando o `FluentArch`. As linhas 3 a 20 são responsáveis pela definição das camadas lógicas do projeto, enquanto as linhas 22 a 38 especificam as restrições de dependência entre essas camadas. Na linha 40, os resultados de conformidade arquitetural de todas as regras definidas são consolidados na variável `violations`. É importante destacar que o `FluentArch` também permite a obtenção dos resultados individualmente, por meio da invocação do método `Check` para cada regra. No contexto desta seção, o foco principal está em garantir que o sistema esteja livre de violações arquiteturais em sua totalidade.

Além das regras de dependência entre camadas, a Listagem 9 ilustra outras regras arquiteturais que foram propositalmente introduzidas pelo primeiro autor deste artigo após estudo e análise do projeto N-Tier-Architecture:

- Nenhuma classe do sistema deve instanciar objetos de repositório; deve-se utilizar injeção de dependência (linhas 32-33).
- Todas as classes localizadas na camada *Application* e no *namespace* `N_Tier.Application.Exceptions` devem herdar de `Exception` (linhas 34-36).
- Todas as classes do *namespace Models* não podem conter métodos (linhas 37-38).

A aplicação do `FluentArch` resultou na identificação de 27 violações arquiteturais, conforme resumido na Listagem 10. Durante a análise, foi observado que os testes automatizados do N-Tier-Architecture estabeleciam dependências com as camadas *Application* e *DataAccess* – um comportamento tecnicamente esperado – mas não explicitado no diagrama de arquitetura fornecido pelo arquiteto do referido projeto. Essa constatação revela a capacidade do `FluentArch` de evidenciar dependências implícitas que não estão formalmente documentadas. Nesse caso em específico, o problema é a definição de regras as quais devem ser ajustadas por meio da

Listagem 9 Regras arquiteturais com `FluentArch`

```
1  var arch = Architecture.Build(solution);
2
3  ILayer camadaApi = arch.All()
4      .ResideInNamespace("N_Tier.API.*")
5      .As("Api layer");
6  ILayer camadaApplication = arch.All()
7      .ResideInNamespace("N_Tier.App.*")
8      .As("App layer");
9  ILayer camadaDataAccess = arch.All()
10     .ResideInNamespace("N_Tier.DataAccess.*")
11     .As("DataAccess layer");
12 ILayer camadaCore = arch.All()
13     .ResideInNamespace("N_Tier.Core.*")
14     .As("Core Layer")
15 ILayer camadaShared = arch.All()
16     .ResideInNamespace("N_Tier.Shared.*")
17     .As("Shared Layer);
18 ILayer camadaModels = arch.All()
19     .ResideInNamespace("N_Tier.App.Models.*")
20     .As("Models layer");
21
22 camadaApi
23     .OnlyCan().Depend(camadaApplication);
24 camadaApplication
25     .OnlyCan().Depend(camadaDataAccess));
26 camadaDataAccess
27     .Cannot().Depend(camadaApplication)
28     .And()
29     .Cannot().Depend(camadaApi);
30 camadaCore
31     .Cannot().Depend(camadaDataAccess);
32 arch.All()
33     .Cannot().Create(camadaDataAccess);
34 camadaApplication
35     .And().ResideInNamespace("N_Tier.App.Exceptions")
36     .Must().Extends("System.Exception");
37 camadaModels
38     .UseCustomRule(new TypeCannotHaveFunctionsRule());
39
40 var violations = arch.Check();
```

inclusão de `.And().ResideInNamespace("N_Tier.Test.*")` no final das linhas 23 e 25.

Listagem 10 Fragmento dos desvios detectados pelo `FluentArch`

```
1. Only the specified layer is allowed to Access from a class in module
   ↪ Application layer, but the type FactoryExtension is also doing so.
2. Only the specified layer is allowed to Access from a class in module
   ↪ Application layer, but the type TodoItemEndpointTests is also doing so.
3. Only the specified layer is allowed to Access from a class in module
   ↪ Application layer, but the type TodoListEndpointTests is also doing so.
...
25. Class UpdateTodoListModelValidator violates the custom rule.
26. Class ConfirmEmailModelValidator violates the custom rule.
27. Class CreateUserModelValidator violates the custom rule.
```

Além disso, a ferramenta detectou violações à regra que proíbe a presença de métodos nas classes pertencentes ao *namespace Models*, as quais não seriam detectadas pelas ferramentas da Seção 2 por não proverem regras customizadas. Esse desvio pode indicar tanto uma atribuição inadequada de responsabilidades a essas classes quanto uma possível má alocação no *namespace*. Uma linha de trabalho promissora seria incorporar sistemas de recomendação de correções de inconsistências arquiteturais [6, 21]. Uma inspeção mais detalhada revelou que as classes de origem das violações possuem o sufixo *Validator*, sugerindo que se tratam de componentes de validação, conceitualmente distintos de modelos de entidade.

Esses achados reforçam a utilidade prática do `FluentArch` como instrumento de verificação arquitetural, ao revelar desvios que não apenas infringem regras explícitas, mas também questionam a coerência entre a estrutura lógica e a intenção arquitetural do sistema. Ao fornecer – comparado com as ferramentas da Seção 2 – granularidade mais fina, possibilidade de regras customizadas e visibilidade sobre tais inconformidades, a ferramenta provê maior expressividade frente às ferramentas existentes.

6 Conclusão

Ferramentas automatizadas de verificação arquitetural tornam-se essenciais para preservar a integridade do projeto arquitetural ao longo do tempo, fornecendo *feedback* imediato sobre violações e evitando o processo de erosão arquitetural.

Nesse cenário, o `FluentArch` foi apresentado como uma alternativa moderna para validação arquitetural em sistemas C#. A biblioteca disponibiliza uma API fluente que simplifica a definição de regras arquiteturais, abstraindo a complexidade inerente ao uso direto do Roslyn, permitindo ao arquiteto focar no que realmente importa: a lógica de conformidade. A proposta se destaca por oferecer granularidade fina, criação de camadas lógicas reutilizáveis e suporte à definição de regras customizadas, com uma sintaxe expressiva e acessível.

A aplicação da ferramenta em um projeto *open source* com arquitetura *N-layer* demonstrou sua viabilidade por meio da definição e avaliação de diversas regras arquiteturais. O `FluentArch` foi capaz de identificar 27 violações, incluindo casos de instanciamento indevido, dependências invertidas e classes com responsabilidades inadequadas. Os resultados foram apresentados de forma clara e estruturada, auxiliando na análise e possível refatoração do sistema.

Comparado às bibliotecas consolidadas no estado-da-prática NetArchTest.eNhancedEdition e ArchUnitNET, o `FluentArch` apresenta vantagens significativas. Enquanto essas soluções operam sobre *bytecode* e oferecem menor expressividade para regras customizadas, o `FluentArch` atua diretamente sobre o código-fonte e permite análises mais precisas e flexíveis. Isso o torna especialmente adequado para cenários que demandam validações detalhadas e adaptadas a diferentes estilos arquiteturais.

Como trabalhos futuros, propõe-se a integração nativa com interfaces mais intuivas em ambientes de desenvolvimento integrado (IDEs) e *frameworks* de testes de unidade, ampliando as possibilidades de uso em *pipelines* de CI/CD. Além disso, está em desenvolvimento o suporte à análise de atributos (*annotations*) e à exclusão de *namespaces* irrelevantes, visando tornar a ferramenta ainda mais adaptável a arquiteturas reais e dinâmicas.

DISPONIBILIDADE DE ARTEFATO

O código fonte da `FluentArch` está publicamente disponível em: https://github.com/arthur-lucas-castro/FluentArch sob a licença MIT. O repositório inclui exemplos de uso, instruções de instalação e um guia para que desenvolvedores contribuam com o projeto.

AGRADECIMENTOS

Esta pesquisa é apoiada pela FAPEMIG, projeto APQ-03513-18.

REFERÊNCIAS

[1] Grigoras Alexandru. 2025. N-Tier-Architecture. https://github.com/nuyonu/N-Tier-Architecture GitHub repository. Acesso em: 19 maio 2025.

[2] ArchUnit. 2024. ArchUnit. https://www.archunit.org/ Acesso em: 14 maio 2025.

[3] Ian F. Darwin. 1988. *Checking C Programs with Lint.* O'Reilly Media, Sebastopol, EUA.

[4] ArchUnitNET Documentation. 2024. ArchUnitNET Documentation. https://archunitnet.readthedocs.io/en/latest/ Read the Docs. Acesso em: 12 maio 2025.

[5] Martin Fowler. 2005. Fluent Interface. https://www.martinfowler.com/bliki/FluentInterface.html Acesso em: 16 julho 2025.

[6] Negar Ghorbani, Tarandeep Singh, Joshua Garcia, and Sam Malek. 2024. Darcy: Automatic Architectural Inconsistency Resolution in Java. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1639–1657.

[7] Alessandro Gurgel, Isela Macia, Alessandro Garcia, Arndt von Staa, Mira Mezini, Michael Eichberg, and Ralf Mitschke. 2014. Blending and reusing rules for architectural degradation prevention. In *13th International Conference on Modularity*. 61–72.

[8] Jens Knodel and Daniel Popescu. 2007. A Comparison of Static Architecture Compliance Checking Approaches. In *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 12–21.

[9] James R. Larus. 1989. Program Instrumentation and Run-Time Monitoring. In *11th Conference on Programming Language Design and Implementation (PLDI)*. 132–143.

[10] MapsterMapper. 2024. Mapster. https://github.com/MapsterMapper/Mapster GitHub repository. Acesso em: 20 maio 2025.

[11] Microsoft. 2024. .NET Compiler Platform ("Roslyn"). https://github.com/dotnet/roslyn?tab=readme-ov-file GitHub. Acesso em: 15 maio 2025.

[12] Microsoft. 2025. Microsoft.CodeAnalysis. NuGet package. Disponível em: <https://www.nuget.org/packages/Microsoft.CodeAnalysis/>. Acesso em: 20 maio 2025.

[13] Gail Murphy, David Notkin, and Kevin Sullivan. 1995. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *3rd Symposium on Foundations of Software Engineering (FSE)*. 18–28.

[14] NDepend. 2024. CQLinq Syntax. https://www.ndepend.com/docs/cqlinq-syntax Acesso em: 17 jul. 2025.

[15] NDepend. 2024. NDepend. https://www.ndepend.com/ Acesso em: 14 maio 2025.

[16] NeVeSpl. 2024. NetArchTest.eNhancedEdition. https://github.com/NeVeSpl/NetArchTest.eNhancedEdition GitHub repository. Acesso em: 12 maio 2025.

[17] David Lorge Parnas. 1994. Software aging. In *16th International Conference on Software Engineering (ICSE)*. 279–287.

[18] Ori Roth and Yossi Gil. 2023. Fluent APIs in Functional Languages. *ACM on Programming Languages* 7, OOPSLA1, Article 105 (2023), 26 pages.

[19] SonarSource. 2024. SonarQube. https://www.sonarsource.com/products/sonarqube/ Acesso em: 14 maio 2025.

[20] Ricardo Terra and Marco Tulio Valente. 2009. A dependency constraint language to manage object-oriented software architectures. *Software—Practice and Experience* 39 (June 2009), 1073–1094.

[21] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. 2015. A Recommendation System for Repairing Violations Detected by Static Architecture Conformance Checking. *Software: Practice and Experience* 45, 3 (2015), 315–342.

[22] xUnit.net. 2024. xUnit.net. https://xunit.net/ Acesso em: 20 maio 2025.