Ben Morris.
Arquitetura de software e projeto de sistemas.

Escrevendo testes de estilo ArchUnit para .Net e C# para impor regras de arquitetura

27 de novembro de 2018

> *Pode ser um grande desafio preservar os padrões de projeto de arquitetura em bases de código a longo prazo. Muitos padrões só podem ser aplicados por convenção, que depende de um processo rigoroso e consistente de revisão de código.*

Essa disciplina geralmente é interrompida à medida que os projetos crescem, os casos de uso se tornam mais complexos e os desenvolvedores vêm e vão. As inconsistências surgem à medida que novos recursos são adicionados de qualquer maneira que se encaixe, dando origem à *podridão de padrões* que prejudica a coerência da base de código.

O ArchUnit tenta resolver esses problemas para bases de código baseadas em Java fornecendo testes que encapsulam convenções em áreas como design de classe, nomenclatura e dependência. Eles podem ser adicionados a qualquer estrutura de teste de unidade e incorporados a um pipeline de compilação. Ele usa uma API fluida que permite agrupar regras legíveis que podem ser usadas em asserções de teste. Alguns exemplos são mostrados abaixo:

```
ArchRule minhaRegra = classes()
    .that().resideInAPackage("..service..")
    .should().onlyBeAccessed().byAnyPackage(..controller..", "..service.."

ArchRule myRule2 = classes().that().implement(Connection.class)
    .should().haveSimpleNameEndingWith("Connection")
```

Esta biblioteca não possui nenhum equivalente direto de código aberto no mundo .Net. Existem muitas ferramentas de análise estática excelentes que avaliam a estrutura do aplicativo, mas elas são mais voltadas para a aplicação de melhores práticas genéricas do

que para convenções específicas do aplicativo. Muitas ferramentas podem ser pressionadas para criar regras personalizadas para uma arquitetura específica, mas é mais difícil integrá-las em um conjunto de testes e criar uma *arquitetura de autoteste* .

Por exemplo, você pode definir regras específicas do projeto no SonarQube se estiver preparado para enfrentar o SDK , mas elas são instaladas em um servidor de aplicativos separado. A implementação do LINQ no nDepend permite interrogar a estrutura do código em um nível muito detalhado, mas o problema aqui é que é um conjunto de ferramentas proprietário que vem com taxas de licença por estação.

O ecossistema .Net não parece ter nenhum meio de baixo atrito para criar regras de arquitetura que atuem em *componentes compilados.* Isso tem menos a ver com a análise genérica de código estático para avaliar a qualidade do código e mais com a verificação de assemblies compilados em relação a restrições específicas do aplicativo.

Determinando dependências de nível de tipo em .Net

NetArchTest é uma implementação básica criada para preencher essa lacuna que é inspirada em partes da API ArchUnit. A intenção era escrever uma biblioteca .Net Standard simples que pudesse ser usada para validar arquiteturas de autoteste em projetos .Net Framework e .Net Core.

O principal desafio para implementar isso é determinar *as dependências* entre as diferentes classes. É aqui que a verdadeira podridão arquitetônica se instala ao longo do tempo. As camadas começam a ignorar umas às outras, as interfaces são implementadas apenas seletivamente, as convenções de nomenclatura começam a cair no esquecimento.

Você pode determinar dependências em um nível de *assembly* com bastante facilidade usando o método `GetReferencedAssemblies()` da reflexão . O verdadeiro desafio é determinar as dependências *entre* os diferentes *tipos* . Esse nível de análise de dependência exige que você analise o código subjacente linha por linha. Uma dependência pode estar escondida em uma declaração de variável enterrada em um método.

O projeto Roslyn expõe a funcionalidade de processamento de código usada no compilador .Net, mas funciona analisando arquivos de código em vez de ler ativos compilados. Isso torna um pouco mais complicado incorporar em testes de unidade como parte de um pipeline de compilação, pois você precisa acessar todo o código-fonte. Além disso, no momento em que escrevo, o Roslyn não suporta diretamente o carregamento de projetos .Net Standard .

Uma alternativa mais direta é a biblioteca Mono.Cecil . Isso foi escrito para ler o formato Common Intermediate Language no qual os projetos .Net Framework e .Net Core são compilados. Além de fornecer uma API que permite percorrer a maioria das construções em um tipo (por exemplo, propriedades, campos, eventos etc.), você também pode inspecionar as instruções em cada método individual. É aqui que você pode pegar as dependências que estão enterradas nos métodos.

Construindo uma API fluente para .Net

Construir uma API fluente para o NetArchTest foi bastante simples, permitindo aos usuários criar frases coerentes compostas de predicados, conjunções e condições.

O ponto de partida para qualquer regra é a classe estática `Types` , onde você carrega um conjunto de tipos de um caminho, assembly ou namespace.

```
var tipos = Tipos .InCurrentDomain();
```

Depois de selecionar os tipos, você pode filtrá-los usando um ou mais *predicados.* Eles podem ser encadeados usando as conjunções `And()` ou `Or() :`

```
types.That().ResideInNamespace( "MeuProjeto.Controladores" );

types.That().AreClasses().And().HaveNameStartingWith( "Base" );

types.That().Inherit( typeof ( MyBase )).Or().ImplementInterface( typeof (
```

Uma vez que o conjunto de classes foi filtrado, você pode aplicar um conjunto de condições usando os métodos Should() ou ShouldNot(), por exemplo

```
types.That().ArePublic().Should().BeSealed();
```

Finalmente, você obtém um resultado da regra usando um executor, ou seja, use `GetTypes()` para retornar os tipos que correspondem à regra ou `GetResult()` para determinar se a regra foi atendida.

Como aplicar a arquitetura em camadas

Um aspecto interessante da API do ArchUnit é a capacidade de definir fatias ou camadas. Isso pode ser usado para impor a verificação de dependência entre diferentes partes de uma arquitetura. Embora eu não tenha adicionado explicitamente definições de camada à API, elas podem ser identificadas pelo namespace conforme mostrado abaixo:

```
// Os controladores não devem referenciar diretamente os repositórios
var result = Types .InCurrentDomain()
    .Que()
    .ResideInNamespace( "NetArchTest.SampleLibrary.Presentation" )
    .Não deveria()
    .HaveDependencyOn( "NetArchTest.SampleLibrary.Data" )
    .GetResult();
```

Regras específicas do aplicativo

A seguir estão alguns exemplos do tipo de regras específicas do aplicativo que podem ser criadas com a API. Tenho evitado afirmar o tipo de convenções de melhores práticas ou orientação de design de tipo afirmado por ferramentas de análise estática. O objetivo é criar regras que expressem convenções específicas do sistema, em vez de encapsular as melhores práticas genéricas.

```
// Somente classes no namespace de dados podem ter uma dependência de Syst
var result = Types .InCurrentDomain()
    .That().HaveDependencyOn( "System.Data" )
    .And().ResideInNamespace(( "ArchTest" ))
    .Should().ResideInNamespace(( "NetArchTest.SampleLibrary.Data" ))
    .GetResult();

// Todas as classes no namespace de dados devem implementar IRepository
var result = Types .InCurrentDomain()
    .That().ResideInNamespace(( "NetArchTest.SampleLibrary.Data" ))
    .And().AreClasses()
    .Should().ImplementInterface( typeof ( IRepository<> ))
    .GetResult();

// Classes que implementam IRepository devem ter o sufixo "Repository"
var result = Types .InCurrentDomain()
    .That().ResideInNamespace(( "NetArchTest.SampleLibrary.Data" ))
    .And().AreClasses()
    .Should().HaveNameEndingWith( "Repositório" )
    .GetResult();

// Todas as classes de serviço devem ser seladas
var result = Types.InCurrentDomain()
    .That().ImplementInterface( typeof ( IWidgetService ))
    .Should().BeSealed()
    .GetResult();
```

O código e o pacote

O código completo da biblioteca está disponível no GitHub .

A base de código inclui alguns exemplos que demonstram alguns dos aplicativos mais úteis.

O pacote lançado está disponível no NuGet como NetArchTest.Rules .