Ben Morris.
Software architecture and system design.

Writing ArchUnit style tests for .Net and C# to enforce architecture rules

27 November 2018

> *It can be quite a challenge to preserve architectural design patterns in code bases over the long term. Many patterns can only be enforced by convention, which relies on a rigorous and consistent process of code review.*

This discipline often breaks down as projects grow, use cases become more complex and developers come and go. Inconsistencies creep in as new features are added in any way that fit, giving rise to *pattern rot* that undermines the coherence of the code base.

ArchUnit attempts to address these problems for java-based code bases by providing tests that encapsulate conventions in areas such as class design, naming and dependency. These can be added to any unit test framework and incorporated into a build pipeline. It uses a fluid API that allows you to string together readable rules that can be used in test assertions. A couple of examples are shown below:

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

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

This library doesn't have any direct, open source equivalent in the .Net world. There are plenty of excellent static analysis tools that evaluate application structure, but they are aimed more at enforcing generic best practice rather than application-specific conventions. Many tools can be press-ganged into creating custom rules for a specific architecture, but it's more difficult to integrate them into a test suite and create a *self-testing architecture*.

For example, you can define project-specific rules in SonarQube if you're prepared to brave the SDK, but these are installed onto a separate application server.
The LINQ implementation in nDepend lets you interrogate code structure to a very detailed

level, but the catch here is that it's a proprietary toolset that comes with per seat license fees.

The .Net ecosystem does not appear to have any low friction means of creating architectural rules that act on *compiled components*. This is less about the generic analysis of static code of assess code quality and more to do with checking compiled assemblies against application-specific constraints.

Determining type-level dependencies in .Net

NetArchTest is a basic implementation created to fill this gap that is inspired by parts of the ArchUnit API. The intention was to write a simple .Net Standard library that can be used to assert self-testing architectures on both .Net Framework and .Net Core projects.

The main challenge for implementing this is in determining *dependencies* between different classes. This is where the real architectural rot sets in over time. Layers start to bypass each other, interfaces are only implemented selectively, naming conventions start to fall by the wayside.

You can determine dependencies at an *assembly* easily enough level using reflection's `GetReferencedAssemblies()` method. The real challenge comes in determining dependencies *between* different *types*. This level of dependency analysis requires you to analyse the underlying code line-by-line. A dependency might be lurking in a variable declaration buried within a method.

The Roslyn project exposes code processing functionality used in the .Net compiler, but it works by parsing code files rather than reading compiled assets. This makes it a little trickier to incorporate within unit tests as part of a build pipeline as you need access to all the source code. Also, at the time of writing Roslyn does not directly support loading .Net Standard projects.

A more straightforward alternative is the Mono.Cecil library. This was written to read the Common Intermediate Language format which both .Net Framework and .Net Core projects are compiled down into. As well as providing an API that lets you walk through most of the constructs in a type (e.g. properties, fields, events, etc.) you can also inspect the instructions in each individual method. This is where you can pick up those dependencies that are buried within methods.

Building an fluent API for .Net

Building a fluent API for NetArchTest was quite straightforward, allowing users to create coherent sentences made up of predicates, conjunctions and conditions.

The starting point for any rule is the static `Types` class, where you load a set of types from a path, assembly or namespace.

```
var types = Types.InCurrentDomain();
```

Once you have selected the types you can filter them using one or more *predicates*. These can be chained together using `And()` or `Or()` conjunctions:

```
types.That().ResideInNamespace("MyProject.Controllers");

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

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

Once the set of classes have been filtered you can apply a set of conditions using the Should() or ShouldNot() methods, e.g.

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

Finally, you obtain a result from the rule by using an executor, i.e. use `GetTypes()` to return the types that match the rule or `GetResult()` to determine whether the rule has been met.

Enforcing layered architecture

One interesting aspect of the ArchUnit API is the ability to define slices or layers. This can be used to enforce dependency checking between different parts of an architecture. Although I have not explicitly added layer definitions to the API, they can be identified by namespace as shown below:

```
// Controllers should not directly reference repositories
var result = Types.InCurrentDomain()
    .That()
    .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
    .ShouldNot()
    .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
    .GetResult();
```

Application specific rules

The following are some examples of the kind of application-specific rules that can be created with the API. I have shied away from asserting the kind of best practice conventions or type design guidance asserted by static analysis tools. The objective is to create rules that express specific system conventions rather encapsulating generic best practice.

```
// Only classes in the data namespace can have a dependency on System.Data
var result = Types.InCurrentDomain()
    .That().HaveDependencyOn("System.Data")
    .And().ResideInNamespace(("ArchTest"))
    .Should().ResideInNamespace(("NetArchTest.SampleLibrary.Data"))
    .GetResult();

// All the classes in the data namespace should implement IRepository
var result = Types.InCurrentDomain()
    .That().ResideInNamespace(("NetArchTest.SampleLibrary.Data"))
    .And().AreClasses()
    .Should().ImplementInterface(typeof(IRepository<>))
```

```
    .GetResult();

// Classes that implement IRepository should have the suffix "Repository"
var result = Types.InCurrentDomain()
    .That().ResideInNamespace(("NetArchTest.SampleLibrary.Data"))
    .And().AreClasses()
    .Should().HaveNameEndingWith("Repository")
    .GetResult();

// All the service classes should be sealed
var result = Types.InCurrentDomain()
    .That().ImplementInterface(typeof(IWidgetService))
    .Should().BeSealed()
    .GetResult();
```

The code and package

The full code for the library is available on GitHub.

The code base includes a few of samples that demonstrate some of the more useful applications.

The released package is available on NuGet as NetArchTest.Rules.