**Architecture Creation Evaluation**

This article shows different ways how to build the architecture at runtime. The architecture presented is a combination of layered architecture and the Model-View-ViewModel pattern. The evaluation was made to show the additional effort that arises without Dependency Injection (DI) (Mvvm4Layer1), with DI by factories (Mvvm4Layer2-4) and with DI container (Mvvm4Layer5).

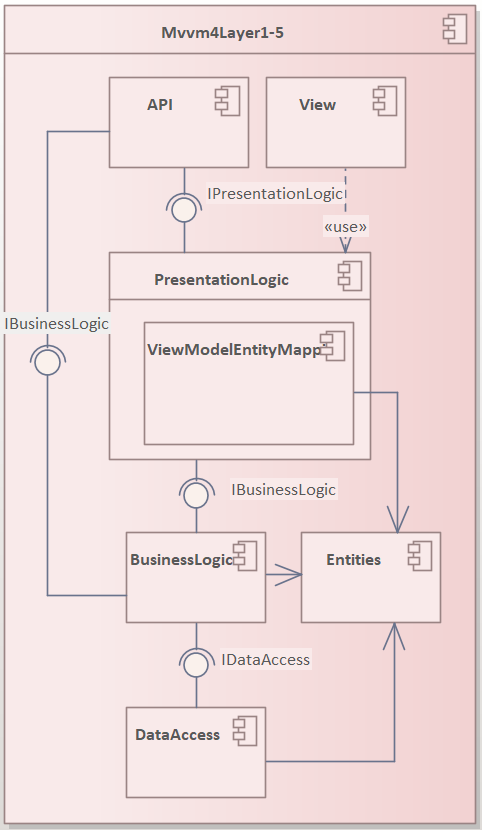
Layered architecture is a quick and easy way to build a working software architecture. But note that there are better ways: Clean, Hexagonal, Onion Architecture etc. Better because the domain is in the centre.

**Naming:**

Mvvm: Model-View-ViewModel-Pattern

4Layer: Strictly speaking, there are 6 layers. However, the Entity component is seen here in parallel with PresentationLogic, BusinessLogic and DataAccess. In addition, the API is not considered a necessary part of the component. Accordingly, View, PresentationLogic, BusinessLogic and the DataAccess form the core architecture, which all depend on the entities.

**Architectural overview:**



**Mvvm4Layer1 Component:**

**Dependency Injection:**

No Dependency Injection for architecture structure! The class representing the layer n (e.g. BusinessLogic-Class) creates the class representing the layer n-1 (DataAccess-Class).

-> BusinessLogic-Class creates DataAccess-Class

This results in a direct dependency that can only be resolved to a limited extent. For example, we could create another constructor in which the required dependency can be initiated.

**Architecturally important:**

-> In this example the API of Mvvm4Layer1 Component must know something about the PresentationLogic- and BusinessLogic-Layer

**Tests**

Constructors are not independent testable (Unit Test):

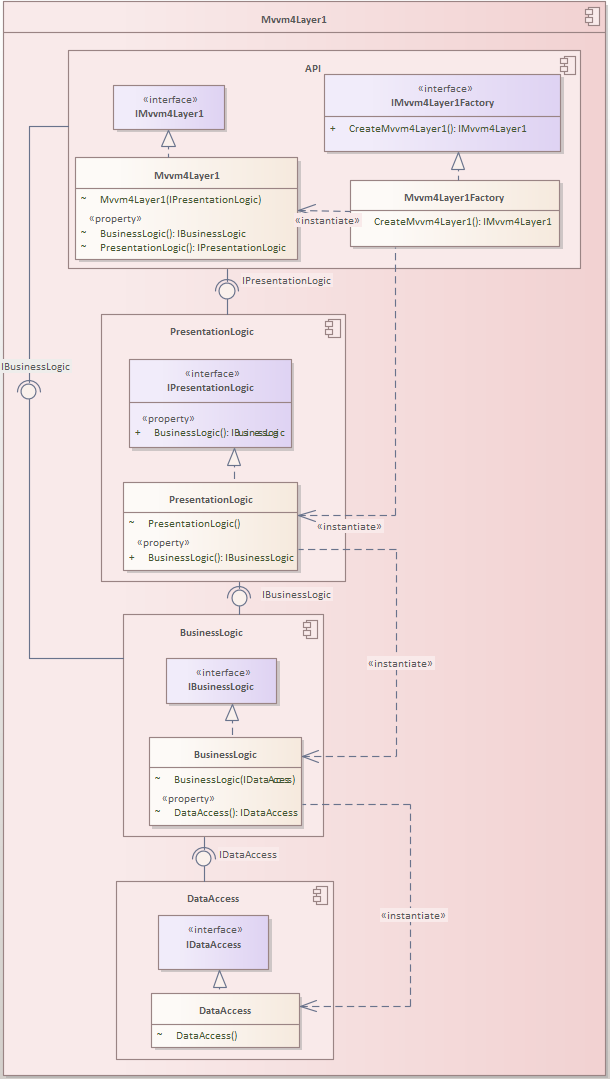
* Mvvm4Layer1Factory -> you can not resolve dependencies to PresentationLogic-, BusinessLogic-, DataAccess- and Mvvm4Layer1-Class.
* PresentationLogic-Constructor instantiates: BusinessLogic-Class
* BusinessLogic-Constructor instantiates: DataAccess-Class
* DataAccess-Constructor

-> you cannot resolve dependencies to the classes instantiating in the constructors

Rest of class could be made testable as follows: you would introduce a constructor only for the tests where the required dependency can be injected. Or use setter injection.

I like to avoid the setter injection, because with it is very inconvenient to force an intra-packet set. The problem with internal setters is that they cannot be used in a public interface (in C#).

**Diagram:**



**Mvvm4Layer2 Component:**

**Dependency Injection:**

Dependency Injection (DI) for architecture structure with one Factory: Mvvm4Layer2Factory.

Mvvm4Layer2Factory creates the PresentationLogic-, BusinessLogic- and DataAccess-Class.

**Architecturally important:**

In this example the API of Mvvm4Layer2 Component must know something about the PresentationLogic-, BusinessLogic- and DataAccess-Layer.

**Tests:**

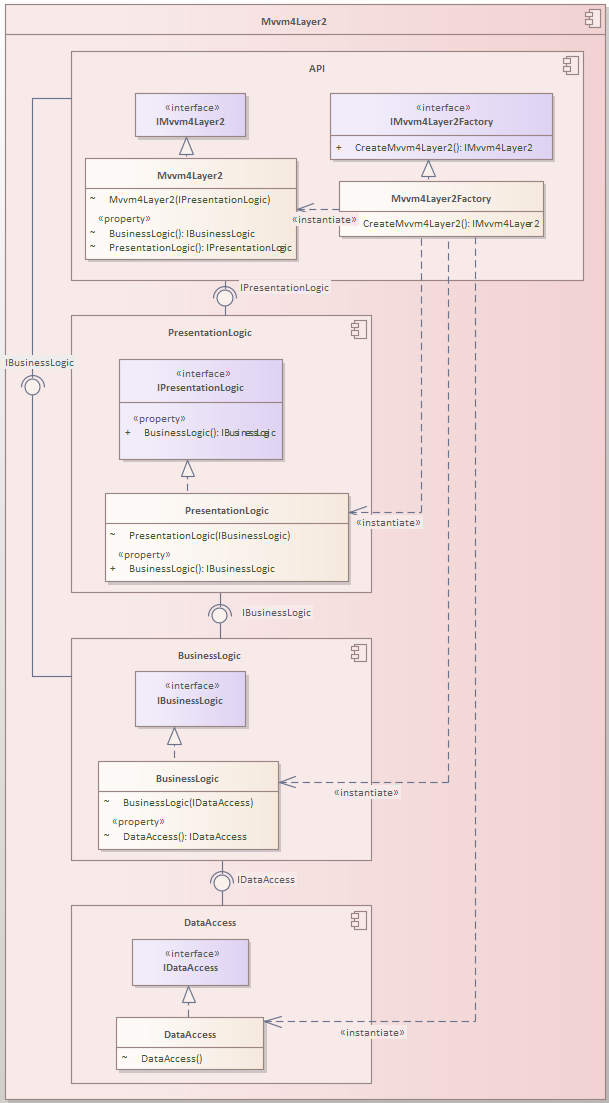
Mvvm4Layer2Factory is not independent testable (Unit Test):

* Mvvm4Layer2Factory -> you can not resolve dependencies to PresentationLogic-, BusinessLogic-, DataAccess- and Mvvm4Layer1-Class.

Testable (Unit Test):

* But you can test the constructors of PresentationLogic-, BusinessLogic-, DataAccess and Mvvm4Layer2-Class independent of other classes due to dependency injection.

**Diagram:**



**Mvvm4Layer3 Component:**

**Dependency Injection:**

Dependency Injection (DI) for architecture structure with factories

Mvvm4Layer3Factory creates the PresentationLogicFactory. Witch creates the PresentationLogic-Class for the Mvvm4Layer3-Class.

The API-Factory in the layer n (e.g. BusinessLogicFactory) creates the class representing the layer n and the API-Factory for layer n-1.

-> BusinessLogicFactory creates BusinessLogic-Class and DataAccessFactory

-> This can resolve the dependencies between the layer representing classes.

-> But the dependencies are now in the Factory-Classes.

-> e.g. PresentationLogicFactory depends on BusinessLogicFactory

**Architecturally important:**

In this approach each layer n only knows something about layer n-1.

-Exception: API-Subcomponent must know something about PresentationLogic- and BusinessLogic-Layer.

**Unit Test:**

* The layer representing classes are independent testable.
* The factories of each layer are harder to test because we can not resolve the dependencies to the created classes.

**Diagram:**

Ein Bild, das Text, drinnen, Screenshot, mehrere enthält.

Automatisch generierte Beschreibung

**Mvvm4Layer4 Component:**

**Dependency Injection:**

Dependency Injection (DI) for architecture structure realized by factories.

The Mvvm4Layer4Factory-Class creates PresentationLogic-, BusinessLogic- and DataAccess-Factory.

The PresentationLogic-, BusinessLogic- and DataAccess-Factory than create the PresentationLogic, BusinessLogic and DataAccess Class.

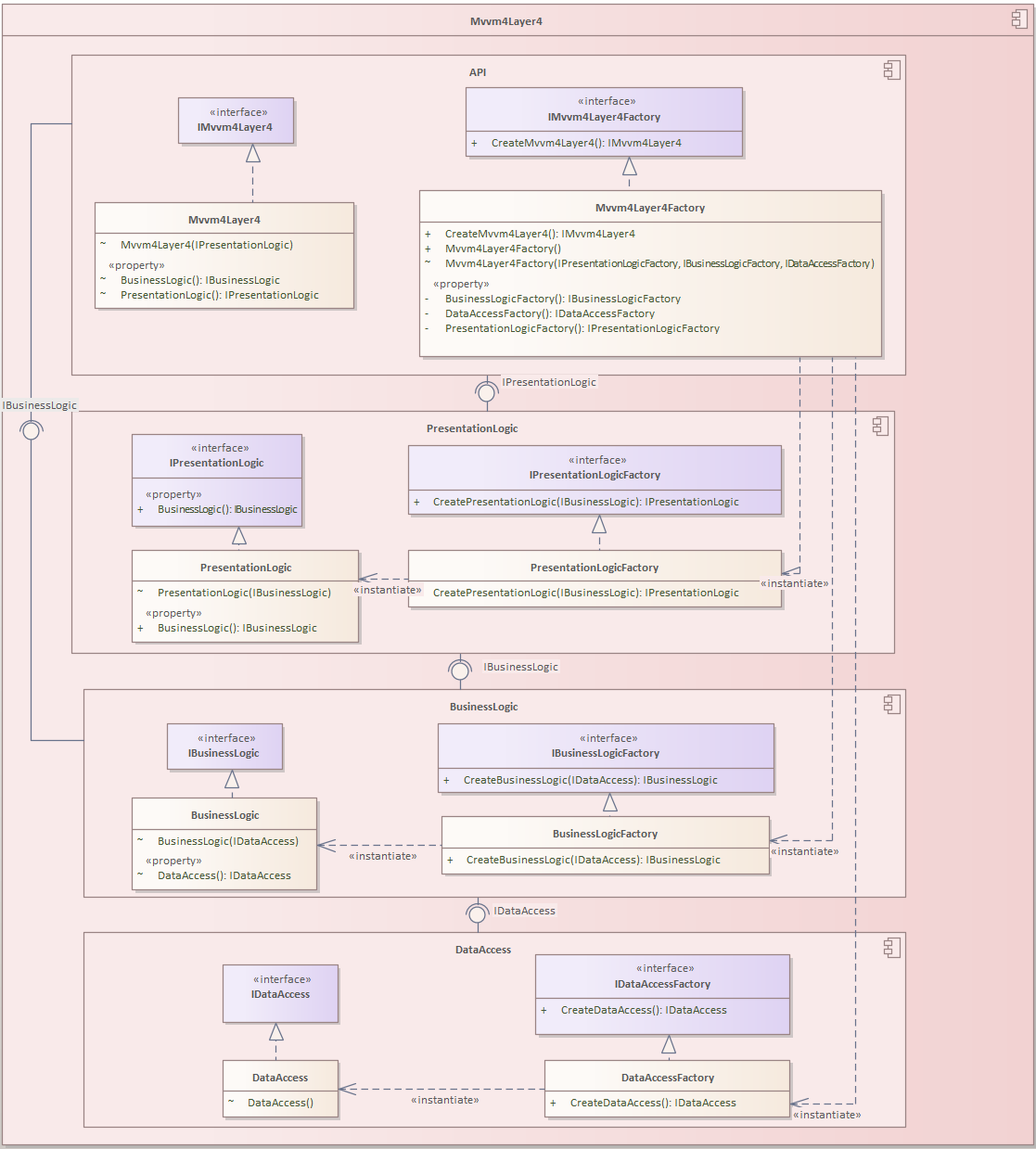
**Architecturally important:**

-> In this example the API of Mvvm4Layer4 Component must know something about the PresentationLogic-, BusinessLogic- and DataAccess-Layer.

**Unit Test:**

* The layer representing classes are independent testable.
* The factories of each layer are harder to test because we can not resolve the dependencies to the created classes.
* In this approach the Mvvm4Layer4Factory-Method witch creates the PresentationLogic-, BusinessLogic-, and DataAcces-Factory is also testable because of the interfaces of each factory and Constructor-Injection.

**Diagram:**



**Mvvm4Layer5 Component:**

**Dependency Injection:**

Dependency Injection (DI) for architecture structure with DI-Container (Ninject)

Mvvm4Layer5-Component needs to initialise Kernel-Bindings. This happens in Mvvm3Layer5Modulue-Class in the DependencyInjectionContainer-Folder

The client how want to use the Mvvm4Layer5-Class must use the Kernel (Ninject Container) to get it. The Kernel resolve all needed dependencies.

**Architecturally important:**

* Tn this approach API-SubComponent must know something about PresentationLogic- and BusinessLogic-Layer.
* The BusinessLogic layer must be known because the API must access business functionality directly.

Beware: Maybe it could be possible that someone who uses the same Ninject Kernel (Container) than our Component (Mvvm4Layer5) can use a Class we don't want anyone else to use.

BUT: this should not be possible when we set this Classes internal. Constructors of this classes must be public because Ninject needs them public. But we can set the types internal.

**Unit Test:**

* The layer representing classes PresentationLogic-, BusinessLogic-, and DataAccess-Class can be tested independent.

**Diagram:**

