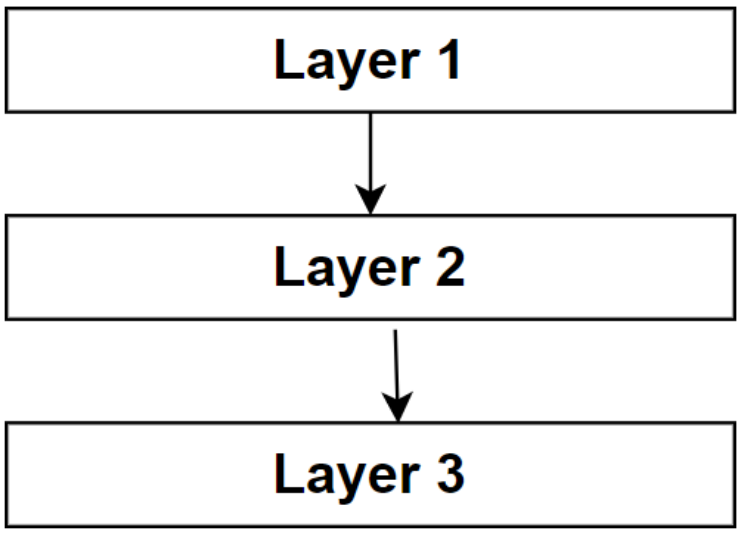
7. Layered Architecture

In this chapter we learn why a (large) application needs layers and how we can structure an application into 4 layers.

7.1 What is a Layered Architecture [1]

Layered architectures are said to be the most common and widely used architectural framework in software development. It is also known as an n-tier architecture and describes an architectural pattern composed of several separate horizontal layers that function together as a single unit of software. A layer is a logical separation of components or code:



Components that are related or that are similar are usually placed on the same layers. However, each layer is different and contributes to a different part of the overall system.

7.2 Seperation of Concerns

Complex applications need some “Separation of concerns” that groups the code into components and on a higher level into layers. Each component / layer takes on a specific part of the functionality with as few overlaps between the components / layers as possible.
In other words: the components / layers should be **loosely coupled**.

Thanks to “Separation of concerns” a lot of the complexity gets encapsulated while having a well-defined interface to the rest of the code.
This makes the code more readable, easier to extend with new features and easier to maintain and test.

Separation of concerns can be applied at multiple levels in the code. [4]

At a lower level the code is grouped into classes. Each class encapsulates complexity while having a public interface (the public properties and methods) that other classes can use.

On a higher level the code can be grouped into layers.
Each layer contains components (classes) that specialize in a certain aspect of the application. The “Presentation layer”, for example, contains the components that are responsible for the user interface (UI) of the application, while the “Infrastructure layer” contains the components that are responsible for communicating with infrastructure (e.g. a database).

A layer has a public interface that other layers can use (except the top layer), while encapsulating the inner workings of the layer.

7.3 Number of layers

The number of layers in a layered architecture is not set to a specific number and is usually dependent on the developer or software architect. It is important to note that this framework will usually always have a user interaction layer, a layer for processing, and a layer that deals with data processing. In some applications, some layers are combined. For example, it is common to find the business layer and persistence layer combined into a single layer. This just means that the functions and responsibilities of these two layers have been grouped to occur at a single layer. [1]

7.4 Layered architecture in .NET

A .NET solution can contain multiple projects. By using a different project for each layer the components (e.g. classes) are physically separated and it will be more difficult to mistakenly use a component of another layer that should not be used.

Also different projects result in different assemblies (dll’s). This enables you to hide certain components (classes) from other layers by using the **internal** keyword.

The internal keyword is an access modifier (like public, private, protected) [2]. An internal class for example is only visible for classes in the same project (assembly). When using a layered approach, prefer internal over public classes. This simplifies the public interface for other layers.

7.5 Example architecture: 4-tier architecture

In this course we will be using a 4-tier approach, with the following layers:

- **Domain Layer**
- **Application Logic/Business Layer**
- **Infrastructure/Persistence/Database Layer**
- **Presentation Layer**

7.4.1 Domain Layer

This layer models the state and behavior of the business domain. It contains a class for each entity in the domain. In this class the **state** is modeled with **properties** and the **behavior** is modeled with **methods**.

The domain layer is the **core of the application**. It does not depend on any other layer.

7.4.2 (Application) Logic Layer

The logic layer models the use cases of the application and handles aspects related to accomplishing functional requirements. In the past this layer was sometimes called “Business layer”.

It contains service classes that use the domain layer classes to complete use cases.Each service class had a public interface (for outer layers) and internal concrete implementation

Sometimes service classes may need to communicate with a database. The data access (repository) interfaces are **defined** in this layer, **but not implemented**. The concrete implementation is the responsibility of the infrastructure layer.

The logic layer should only communicate with the domain layer, no other layers.

7.4.3 Infrastructure Layer

This layer is responsible for handling data and/or databases as it links with a concrete storage medium (e.g. database).
The implementation of the data access interfaces (repositories) that were define in the logic layer are implemented can be found in this layer. It can also work with other infrastructure like a mail service, file system, … (though this is out of scope for this course).

The infrastructure layer references the logic layer and domain layer (no other layers).

7.4.4 Presentation Layer

The final layer contains components that are responsible for showing information to the end users and to interpret commands of these end users. It is responsible for user interactions with the software system.

The presentation layer uses the public interface of the logic layer and domain layer.

7.6 Practical wiring in .NET

Internal services (logic layer) and repositories (infrastructure layer) need to be instantiated somewhere.

The presentation layer is the layer that will be executing the program and needs to wire everything together. This forces us to reference the infrastructure layer and expose internal classes of infrastructure and logic layer (can be avoided with a bootstrapper project but that is out of scope for this course).

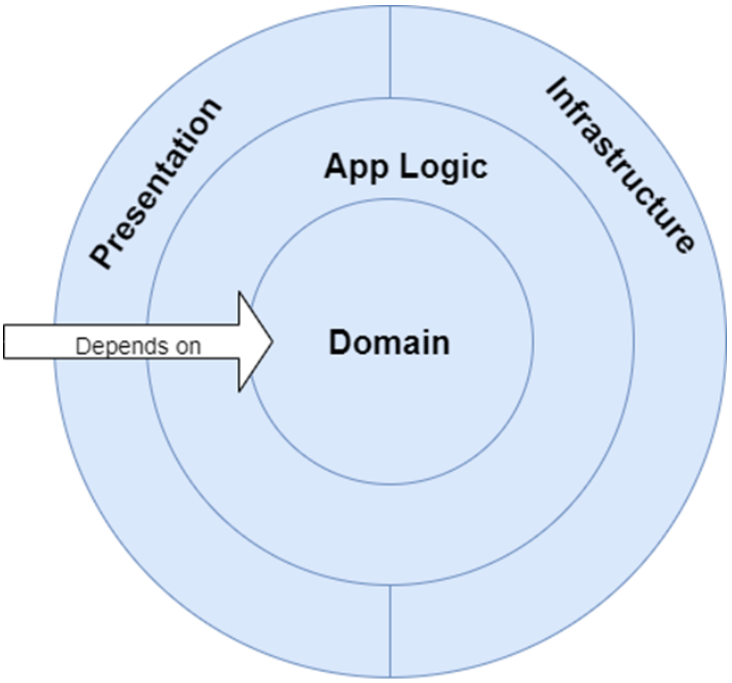
Expose internals:
In .NET: add the *InternalsVisibleTo attribute* to the logic and infrastructure layer in *Properties/AssemblyInfo.cs*:

```
AssemblyInfo.cs
HumanRelationsApp.Business
1  using System.Reflection;
2  using System.Runtime.CompilerServices;
3  using System.Runtime.InteropServices;
4
5  [assembly: InternalsVisibleTo("HumanRelationsApp.Presentation")]
```

The presentation layer will now have access to the internal classes!

Application startup:
Override OnStartUp method in App.xaml.cs and create concrete instances of services and repositories in this method. Next inject these instances in the Window via the constructor.

7.7 Dependencies



The figure above shows the different dependencies in a layered application. Outer layers can have a dependency on an inner layer. The inner layers have no knowledge of the outer layers and they do not need to.

The figure shows:

- Domain layer does not depend on other layers

- Application logic layer depends on domain layer
- Infrastructure layer depends on application logic and domain layer
- Presentation layer depends on application logic and domain layer
- Presentation layer and infrastructure layer don't know each other.

7.8 Sidenote: the classic 3-tier architecture

Some (classic) architectures use the infrastructure layer as the central layer (usually named "data layer"). Here the logic layer (usually named "business layer") sits on top of the data layer and the presentation layer on top of the business layer. Hence the name "3-tier architecture".

In this classic architecture there is no domain layer (or the domain layer and data layer are the same). This architecture builds on the idea that the **database of an application is designed first** and is the central building block of an application.

Today more and more architects believe that the (business) domain should be the central building block of an application. The way things are persisted becomes an implementation detail. The database model can be derived from the domain model.

Sources:

[1] https://www.baeldung.com/cs/layered-architecture
[2] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/internal
[3] https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html
[4] https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles