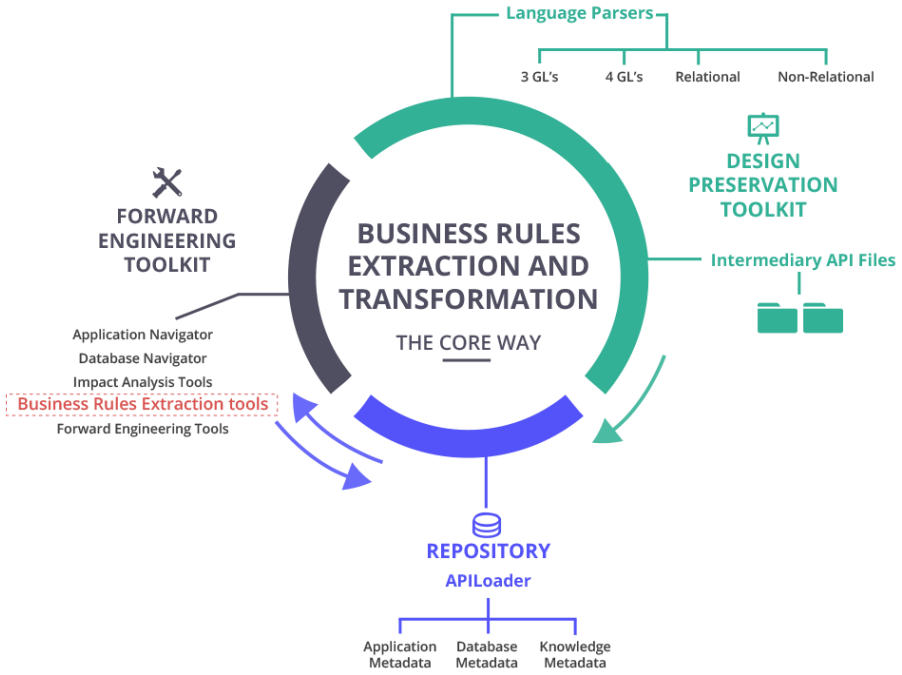
# View Models

- Reglas de negocio
- Models
- Capa Anticorrupción
- View Models

# Reglas de negocio (Modelos)

Las reglas de negocio en el contexto del software son declaraciones que definen o limitan algún aspecto del comportamiento del sistema. Estas reglas son esenciales para el correcto funcionamiento de una aplicación y pueden abordar una variedad de aspectos, desde la validación de datos hasta la lógica de procesos.



# Reglas de negocio (Modelos)

Las reglas de negocio se utilizan para garantizar que el software opere de acuerdo con los requisitos y las políticas establecidas. Para esto se suele armar el modelo o núcleo de una aplicación

El modelo de arquitectura limpia separa la lógica de negocio de:

- Los periféricos de entrada/salida
- Sintaxis SQL
- Estructuras de datos que se importan de una librería de tercero
- El motor que renderiza las plantillas html
- El formato en que se debe enviar la respuesta al cliente ya sea JSON o XML

# Models (Modelos)

- Evita que la lógica de negocio tenga conocimientos o responsabilidades sobre cosas cómo:

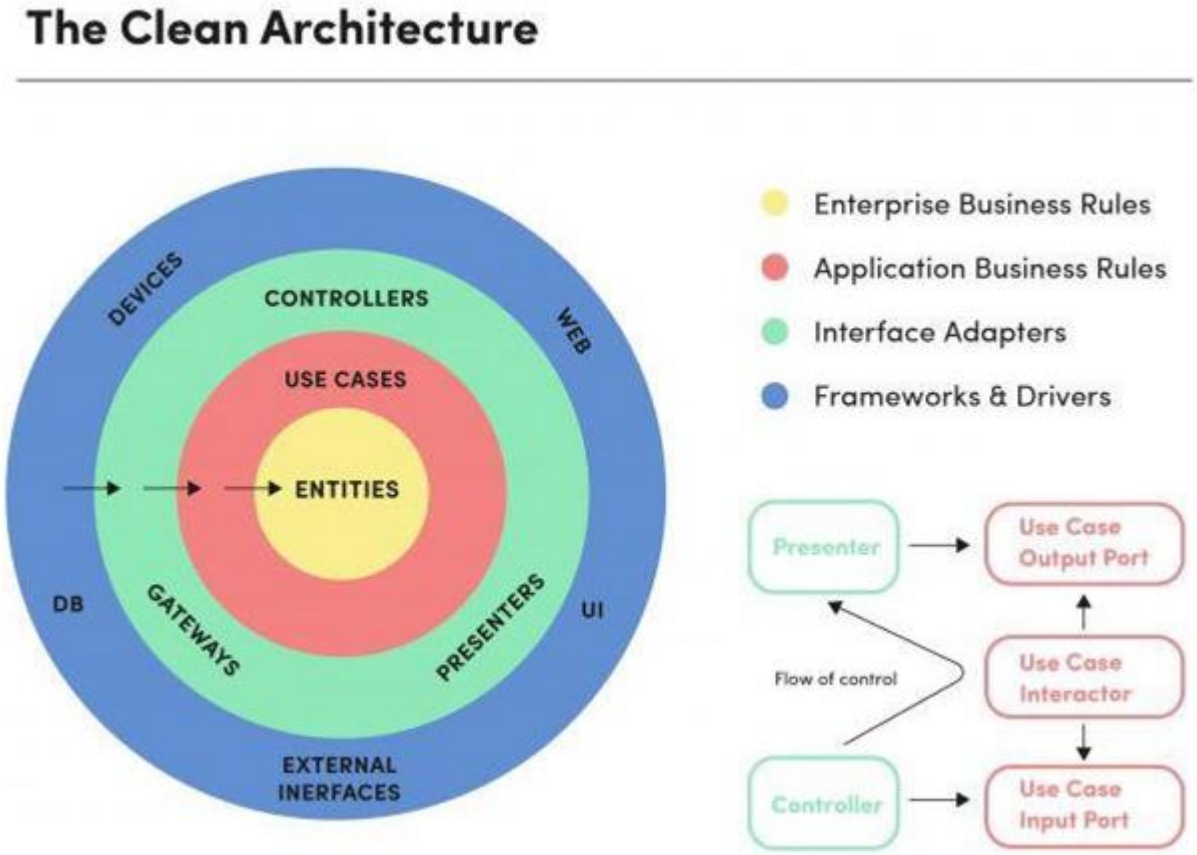
    - Si se le está activando por una petición http o por un websocket
    - Si se está invocando desde un dispositivo android o iOS
    - Conexiones a bases de datos u recursos
    - Etc.

# Models (Modelos)

- Entidades : las entidades son el conjunto de reglas comerciales relacionadas que son críticas para el funcionamiento de una aplicación.

- Son completamente independientes de cualquier otro elemento dentro de la arquitectura. Cuando se emplea una Arquitectura Limpia, las Entidades no saben nada sobre las otras capas.

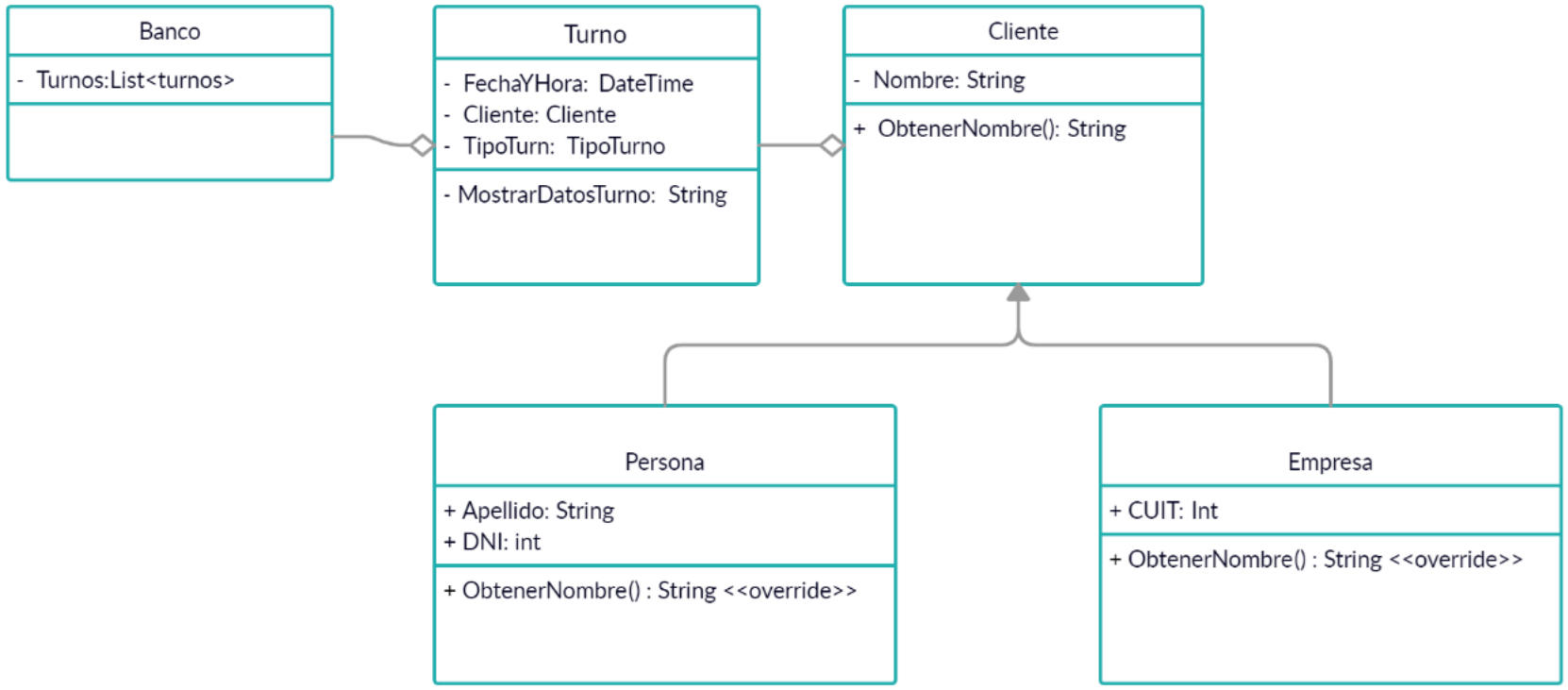
# Models (Modelos)




- Entidades: son el núcleo de un sistema y no deberían tener dependencias de de otras capas de la aplicación

# Models (Modelos)

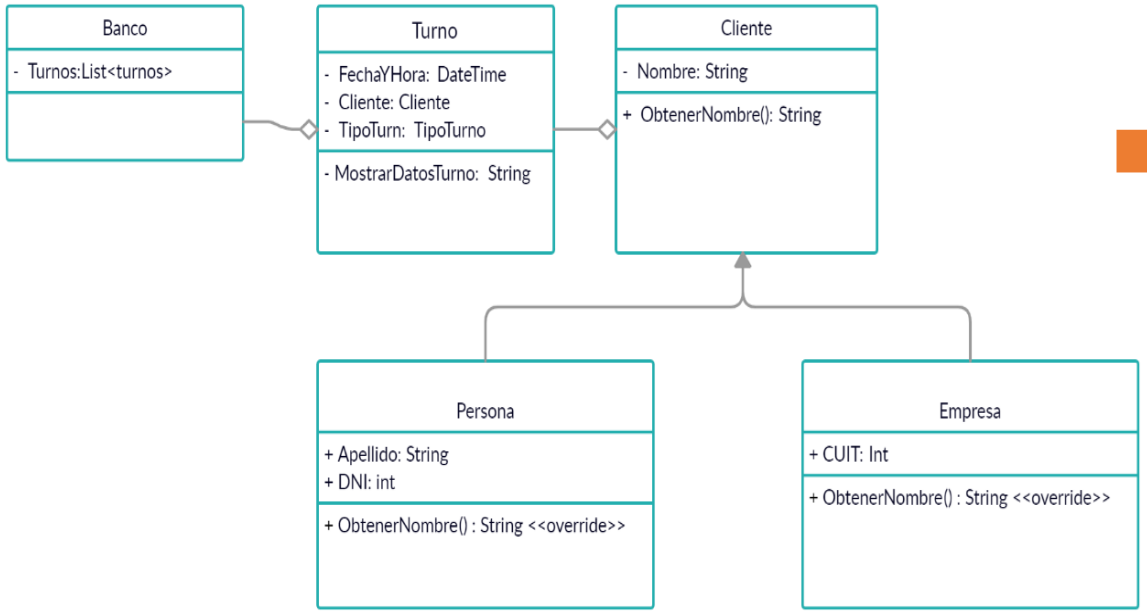
- Consideremos el caso anterior en la vista



# Models (Modelos)

- Consideremos el caso anterior en la vista

Turno en el backend (modelo)



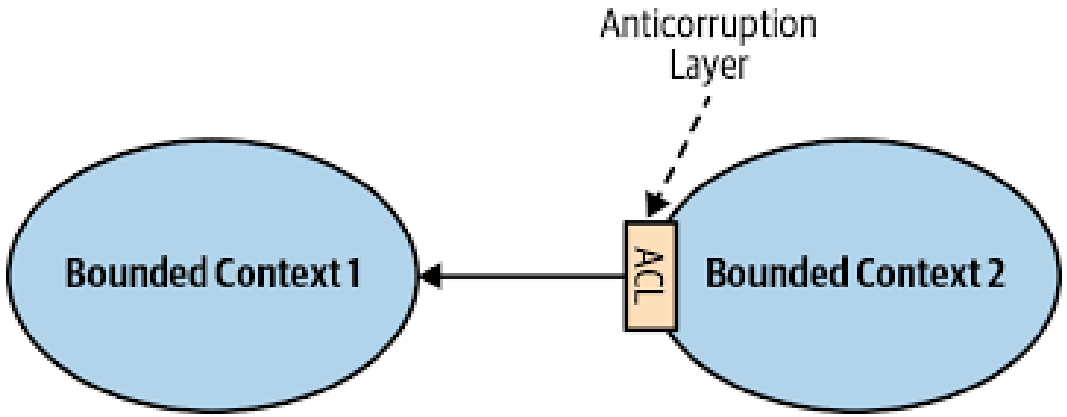

Turno en la vista

Fecha y hora

Nombre cliente

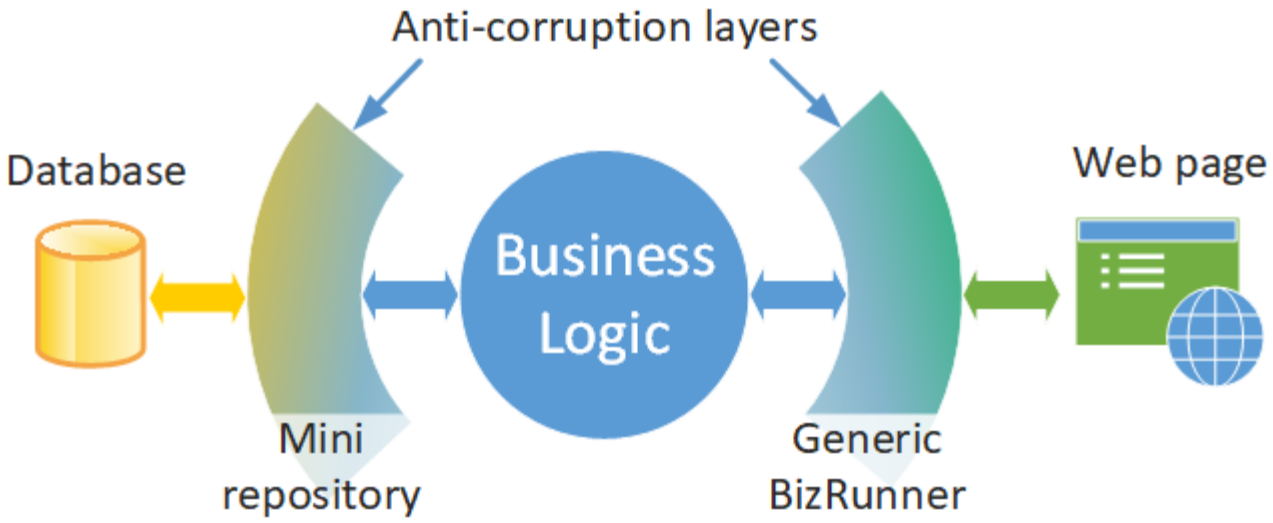
# Capa anticorrupción

El Diseño Dirigido por el Dominio (DDD) define una capa anti-corrupción como un patrón de adaptador que aísla una parte de un sistema, conocida en DDD como un "contexto delimitado", de otro contexto delimitado. Su función es garantizar que la semántica en un contexto delimitado no "corrompa" la semántica del otro contexto delimitado.



# Capa anticorrupción

Según la documentación de Microsoft “la capa anticorrupción Implementa una fachada o capa de adaptador entre diferentes subsistemas que no comparten la misma semántica.”




Img. https://www.thereformedprogrammer.net/wrapping-your-business-logic-with-anti-corruption-layers-net-core/

# Capa anticorrupción

Algunos ejemplos de uso de una capa anti-corrupción en una aplicación web son:

**Lógica de negocios:** Su lógica de negocios no debería tener que preocuparse por cómo funciona la base de datos o el front-end.

**Front-end**: Los usuarios “humanos” necesitan datos que les sean útiles, no necesariamente conocer cómo se diseñó la base de datos o las estructuras internas de un sistema.

**Web API:** Su API web debe proporcionar un servicio que se ajuste a las necesidades externas.

**Seguridad:** Las capas anti-corrupción también pueden ayudar con la seguridad, al pasar únicamente los datos exactos que la otra área necesita.

Img. https://www.thereformedprogrammer.net/wrapping-your-business-logic-with-anti-corruption-layers-net-core/

# View models

Llamamos ViewModel al objeto que se le pasa a una vista para facilitar la presentación de los datos. Normalmente se busca que estos objetos sean los más simples posibles y que tengan solo datos relacionados con la necesidad de la vista.

En pocas palabras, los View Models son los equivalentes de los modelos para un sistema pero orientado a una vista.

# View models

Modelos

Abstracciones que representa las reglas críticas de la aplicación. Permiten a la aplicación conocer Y comprender las reglas de negocio.

Aplicación

ViewModels

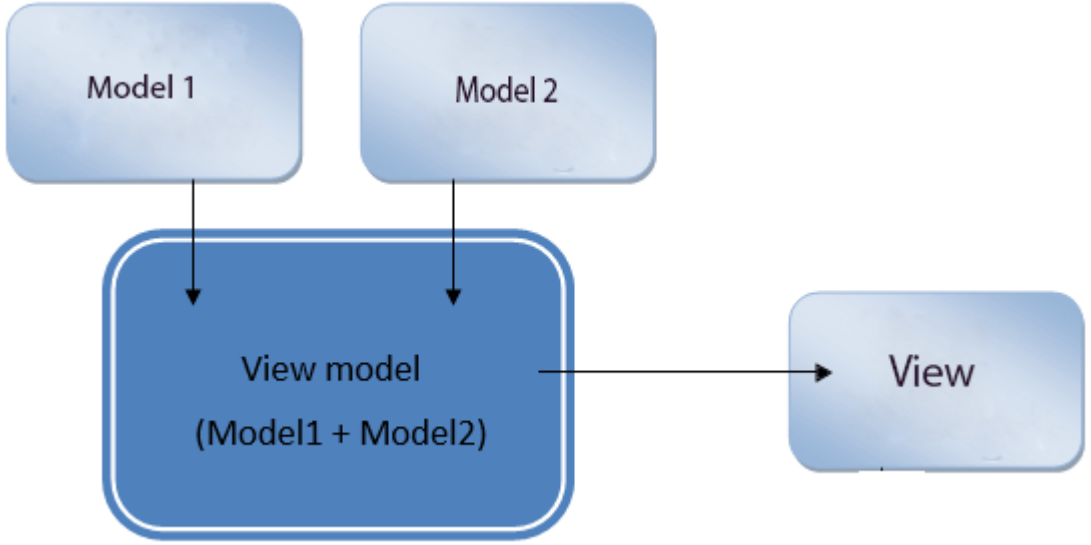
Abstracciones que facilitan la presentación y manipulación de datos en una vista

Vistas

# View models

En ciertas situaciones, es posible que un solo objeto de modelo no contenga todos los datos necesarios para una vista. En tales situaciones, necesitamos usar ViewModel en la aplicación.

Esta conversión en un solo objeto nos proporciona una mejor optimización una mejor abstracción y facilidad de uso en las Vistas



# View models

Escenario 1: Usando Dos Modelos para Llegar a un ViewModel

```
public class Usuario
{
    public int Id { get; set; }
    public string Nombre { get; set; }
}

public class HistorialCompras
{
    public int UsuarioId { get; set; }
    public decimal MontoTotalCompras { get; set; }
}
```

```
public class DetallesUsuarioViewModel
{
    public string Nombre { get; set; }
    public decimal MontoTotalCompras { get; set; }

    public DetallesUsuarioViewModel(Usuario usuario, HistorialCompras
historialCompras)
    {
        Nombre = usuario.Nombre;
        MontoTotalCompras = historialCompras.MontoTotalCompras;
    }
}
```

# View models

**Escenario 2: Simplificando un Modelo Complejo a un ViewModel más simple**

```
public class Producto
{
    public int Id { get; set; }
    public string Nombre { get; set; }
    public string Descripcion { get; set; }
    public decimal Precio { get; set; }
    public DateTime FechaDeAlta { get; set; }
    public DateTime FechaDeAlta { get; set; }
}
```

```
public class ProductoSimpleViewModel
{
    public string Nombre { get; set; }
    public decimal Precio { get; set; }

    public ProductoSimpleViewModel(Producto producto)
    {
        Nombre = producto.Nombre;
        Precio = producto.Precio;
    }
}
```

# View models

**Escenario 3: Modelo Simple con ViewModel que Contiene Más Datos**

```
public class Libro
{
    public int Id { get; set; }
    public string Titulo { get; set; }
    // Otras propiedades del libro
}
```

```
public class DetallesLibroViewModel
{
    public string Titulo { get; set; }
    public string Autor { get; set; }
    public string Categoria { get; set; }

public DetallesLibroViewModel(Libro libro, string autor, string
categoria)
    {
        Titulo = libro.Titulo;
        Autor = autor;
        Categoria = categoria;
     }
}
```

# View models

## Puntos Importantes

- Contiene campos que están representados en la vista (para Helpers LabelFor,EditorFor,DisplayFor)
- Puede tener reglas de validación específicas mediante anotaciones de datos o IDataErrorInfo.
- Poner solo datos/campos relevantes a la Vista que se quiere representa.
- Si solo se utilizan los campos de los view models, esto facilitará la renderización y el mantenimiento

# View models
# Conclusiones

Los ViewModel nos proporciona la separación de responsabilidades (SoC) adecuada.
La Vista solo necesita representar el objeto ViewModel único, y hay un propósito específico para todos y cada uno de los aspectos de la aplicación, lo que significa que la aplicación estará más organizada en el código, manteniendo separada la lógica de la aplicación de la lógica de presentación de los datos