🧱 1. Fundamentos de Arquitectura Hexagonal (Ports and Adapters)

🎯 Objetivo General:

> **Separar el núcleo del dominio** (es decir, la lógica central del problema que resuelve nuestra aplicación) de los **detalles técnicos** (como bases de datos, frameworks, servidores, etc.)

🔍 ¿Qué es el "núcleo del dominio"?

Imagina que estás creando una aplicación para manejar pedidos de comida.

El "núcleo del dominio" sería:

- Saber qué es un pedido
- Qué reglas debe cumplir un pedido
- Cómo calcular el precio total
- Qué estados puede tener (creado, en preparación, entregado)

Nada de eso tiene que ver con **cómo se guarda en la base de datos**, **si llega por una API**, o **si se imprime en pantalla**. Eso es **infraestructura**. Lo importante es que puedas razonar sobre tu aplicación **como si fueras un chef, no un programador**.

Ahora desglosamos los **beneficios clave** de separar esto como pide la arquitectura hexagonal:

✅ 1. Alta testabilidad

¿Qué significa?
Podemos probar fácilmente si nuestra lógica funciona sin tener que conectarnos a una base de datos o levantar un servidor web.

Ejemplo simple:
Si tienes una función que calcula el total de un pedido con descuentos, puedes probarla con un test así:

```Go
pedido := NuevoPedido([]Item{Pizza, Bebida})
total := pedido.CalcularTotal()
assert.Equal(t, 25000, total)
```

No necesitas guardar nada en la base de datos. ¡Y eso es bueno!

✅ 2. Bajo acoplamiento

¿Qué significa?
Cambiar una parte de tu sistema (como el proveedor de correo o la base de datos) **no rompe las otras partes**.

Ejemplo simple:
Hoy envías un correo de confirmación usando Gmail. Mañana quieres usar SendGrid. Si tienes un adaptador que implementa una interfaz `EmailSender`, puedes cambiar la implementación sin tocar la lógica del negocio.

```Go
type EmailSender interface {
    EnviarCorreo(destinatario string, mensaje string) error
}
```

La lógica del dominio solo sabe que hay un `EmailSender`, no cómo funciona por dentro.

✅ 3. Alta cohesión

¿Qué significa?
Cada parte del sistema hace solo lo que le corresponde y lo hace bien.

Ejemplo:

- El dominio se encarga de las reglas del negocio.
- La capa de infraestructura se encarga de hablar con la base de datos.
- La capa de entrada (controladores) se encarga de recibir la solicitud del usuario.

Esto hace que el código sea **más ordenado y más fácil de entender**.

✅ 4. Independencia de frameworks, bases de datos, y protocolos

¿Qué significa?
Puedes cambiar de tecnología sin reescribir todo el sistema.

Ejemplo:

- Puedes cambiar de PostgreSQL a MongoDB solo cambiando un adaptador.
- Puedes exponer tu sistema como API REST hoy y como gRPC mañana, sin tocar el dominio.
- Incluso podrías ejecutar tu sistema sin internet, todo en memoria.

🎓 Analogía:

Piensa en una cafetera.

- El dominio es el proceso de hacer café (moler, hervir, filtrar).
- La infraestructura es si usas una cafetera italiana, una prensa francesa o una máquina de cápsulas.
- El café siempre debe salir bien... **sin importar la máquina.**

Cuando el dominio está **bien separado**, puedes:

- Probarlo en aislamiento
- Reusarlo en otro entorno
- Cambiar la tecnología sin miedo
- Comprenderlo mejor

🧠 2. Componentes Clave

🧠 A. Dominio (Core / Núcleo de la Aplicación)

El dominio representa la esencia del problema que queremos resolver con nuestro sistema.
Es como el "cerebro" de la aplicación: toma decisiones, válida reglas y mantiene la lógica de negocio.

📌 **Regla de oro:**

El dominio no debe depender de frameworks, bases de datos, controladores web, ni servicios externos.

1. 🧍 Entidades (Entities)

¿Qué son?
Son objetos que tienen identidad propia y que pueden cambiar con el tiempo, pero siguen siendo los mismos.

Ejemplo:
Un `Usuario` puede cambiar su nombre o correo, pero sigue siendo el mismo usuario por su ID.

```go
type Usuario struct {
    ID    string
    Nombre string
    Email  string
}
```

Pista para saber si es una entidad:
¿Puedo diferenciar dos objetos por su identidad, aunque su contenido sea igual?

→ Es una entidad.

2. 🎯 Objetos de Valor (Value Objects)

¿Qué son?
Son objetos sin identidad, que se definen solo por sus datos. Son inmutables: una vez creados, no cambian.

Ejemplos comunes:

- **Un correo electrónico (`Email`)**
- **Una cantidad de dinero (`Money`)**
- **Un rango de fechas**

```Go
type Email struct {
    valor string
}

func (e Email) Value() string {
    return e.valor
}
```

3. 🧱 Agregados (Aggregates)

¿Qué son?
Un agregado es un conjunto de entidades y objetos de valor que funcionan como una unidad.
Tiene una raíz (root) que es la única entrada para modificarlo.

Ejemplo:
Un `Pedido` (`Order`) que contiene:

- **Cliente (entidad)**
- **Lista de productos (entidades)**

- **Dirección de envío (objeto de valor)**

```go
type Pedido struct {
    ID       string
    Cliente  Cliente
    Productos []Producto
    Estado   string
}
```

La clase `Pedido` es la raíz del agregado. Si quieres cambiar algo del pedido, debes pasar por ella.

4. ⚖️ Servicios de Dominio

¿Qué son?
Son clases que contienen reglas del negocio que:

- **No pertenecen naturalmente a una entidad**
- **Involucran varios objetos**

Ejemplo:
`CalculadoraDePrecioDeEnvio` puede recibir un pedido y calcular su costo de envío según la distancia, peso, etc.

```go
type CalculadoraEnvio struct {}


func (c CalculadoraEnvio) Calcular(pedido Pedido)
float64 {
    // lógica de negocio pura
}
```

5. 📦 Repositorios (Interfaces)

¿Qué son?
Son interfaces (no implementaciones) que definen cómo se guarda o recupera un agregado.

Ejemplo:

```go
type PedidoRepository interface {

    Guardar(pedido Pedido) error

    BuscarPorID(id string) (*Pedido, error)

}
```

Importante:

- **No contiene lógica de base de datos.**
- **Solo define lo que el dominio necesita saber para funcionar.**

6. 🏭 Fábricas (Factories)

¿Qué son?
Son clases que encapsulan la lógica compleja de creación de entidades o agregados.

¿Por qué se usan?
Porque a veces crear un objeto correctamente requiere varios pasos, validaciones o cálculos.

Ejemplo:

```go
type FabricaPedido struct {}


func (f FabricaPedido) CrearPedido(cliente Cliente,
productos []Producto) (Pedido, error) {

    // lógica de validación y creación
```

```
}
```

💡 Todo esto NO debe depender de:

- **Bases de datos (PostgreSQL, Mongo, etc.)**
- **Frameworks (Gin, Spring, Express)**
- **Interfaces de usuario**
- **Servicios externos (APIs, correos, etc.)**

🧭 B. Capa de Aplicación (Use Cases)

> Esta capa **no contiene reglas de negocio**, sino que **organiza y coordina** cómo se usan las reglas que ya existen en el **dominio** para cumplir con una tarea específica del sistema.

📦 1. Casos de Uso (Application Services)

¿Qué son?
Son las **acciones que el sistema puede realizar**. Cada uno representa una operación útil desde el punto de vista del usuario o del negocio.

Ejemplo:

- `RegisterUser` → registra un nuevo usuario
- `PlaceOrder` → crea un pedido
- `SendInvoice` → envía una factura por correo

Ejemplo en pseudocódigo:

```Go
type RegisterUserService struct {
```

```
    userRepo UserRepository // Output port
    emailSender EmailSender // Output port
}

func (s RegisterUserService) Register(cmd RegisterUserCommand) error {
    usuario := NuevoUsuario(cmd.Nombre, cmd.Email)
    err := s.userRepo.Guardar(usuario)
    if err != nil {
        return err
    }
    return s.emailSender.EnviarBienvenida(usuario.Email)
}
```

👉 El servicio:

- **Crea un usuario** usando lógica del dominio
- **Guarda** el usuario usando un repositorio
- **Envía un correo** con un adaptador externo

🧩 2. Input Ports (Puertos de entrada)

¿Qué son?
Son **interfaces** que definen **qué operaciones externas están permitidas** en el sistema.
Los **controladores** (como REST o GraphQL) llaman a estos puertos para activar casos de uso.

Ejemplo:

```Go
type RegisterUserInputPort interface {
    Register(cmd RegisterUserCommand) error
}
```

Esto permite que cualquier adaptador externo (como una API REST o un mensaje de Kafka) llame al caso de uso **sin depender directamente del servicio**.

🔌 3. Output Ports (Puertos de salida)

¿Qué son?
Son **interfaces** que definen lo que la capa de aplicación necesita que otro sistema haga.

👉 Se usan para **abstraer la infraestructura externa**, como:

- Base de datos
- Envío de correos
- Servicios externos (pasarela de pagos, notificaciones)

Ejemplo:

```go
type EmailSender interface {
    EnviarBienvenida(email string) error
}

type UserRepository interface {
    Guardar(usuario Usuario) error
}
```

👉 Estos puertos son **implementados más tarde** por adaptadores (por ejemplo, un repositorio en PostgreSQL o un servicio SMTP).

📤 4. DTOs / Requests / Responses

¿Qué son?
Son estructuras de datos **simples** usadas para **transportar información entre capas**.
Sirven para:

- Recibir datos del mundo externo (input)
- Devolver resultados (output)
- Evitar que el dominio tenga que entender cosas del exterior (como formatos JSON, IDs string, etc.)

Ejemplo:

```go
type RegisterUserCommand struct {
    Nombre string
    Email  string
}

type RegisterUserResponse struct {
    UserID string
    Mensaje string
}
```

🧠 ¿Por qué esta separación es útil?

Componente	¿Para qué sirve?	¿Ventaja?
Casos de uso	Ejecutar acciones completas del sistema	Orquestación clara
Input ports	Conectar controladores con la lógica	Desacopla presentación y lógica
Output ports	Delegar tareas externas (DB, email, etc.)	Testeable, intercambiable
DTOs	Mover datos de forma segura	Evita fugas de detalles técnicos al dominio

🧩 C. Adaptadores (Infrastructure y Interface)

Esta capa conecta el mundo real (navegadores, bases de datos, redes, APIs externas) con nuestra aplicación.
Los adaptadores implementan las interfaces (puertos) que definimos en el dominio y la aplicación.

🔁 ¿Por qué se llaman “adaptadores”?

Porque “adaptan” el lenguaje del mundo exterior al lenguaje interno de nuestra aplicación, y viceversa.
Así como un cargador adapta el enchufe de la pared a tu teléfono.

1️⃣ Adaptadores de Entrada (Driving Adapters)

👉 Son los puntos de entrada de las acciones al sistema.
Son los encargados de recibir solicitudes del usuario o de otro sistema externo.

Ejemplos comunes:

📡 a. Controladores HTTP (REST, GraphQL)

- **Reciben una solicitud HTTP (`POST /usuarios`)**
- **Toman los datos del body y los convierten en un DTO**
- **Llaman al input port (caso de uso)**

```go
func (c *UserController) RegistrarUsuario(w
http.ResponseWriter, r *http.Request) {

    var cmd RegisterUserCommand

    json.NewDecoder(r.Body).Decode(&cmd)

    err := c.inputPort.Register(cmd)

    // manejar la respuesta

}
```

🖥️ b. Interfaz gráfica (UI en desktop o web)

- **Un botón en la pantalla puede llamar al caso de uso internamente**
- **La lógica sigue pasando por el input port**

💬 c. Cola de mensajes (Kafka, RabbitMQ)

- **Reciben un evento externo (ej: "NuevoPedidoCreado")**
- **Transforman el mensaje en un DTO**
- **Llaman al caso de uso correspondiente**

2️⃣ Adaptadores de Salida (Driven Adapters)

👉 **Son los implementadores de los puertos de salida, que hacen tareas como guardar en base de datos, enviar correos, o llamar APIs.**

Ejemplos comunes:

💽 a. Base de datos (PostgreSQL, Mongo, etc.)

- **Implementan el `UserRepository`**
- **Traducen un `Usuario` a una entidad de tabla**
- **Usan SQL, ORM, o consultas para persistir o leer**

```go
type UserRepositoryPostgres struct {
    db *sql.DB
}

func (r *UserRepositoryPostgres) Guardar(usuario Usuario) error {
```

```
    _, err := r.db.Exec("INSERT INTO usuarios ...",
usuario.ID, usuario.Nombre, ...)

    return err

}
```

📤 b. Servicios externos (Email, APIs)

- **Implementan `EmailSender`**
- **Usan un cliente HTTP o SMTP para enviar correos**

```
Go
type SMTPSender struct {}


func (s SMTPSender) EnviarBienvenida(email string)
error {

    // Llamar a servicio SMTP real

}
```

📡 **c. Envío de eventos a colas**

- **Implementan puertos como `EventoPublisher`**
- **Publican mensajes a Kafka, RabbitMQ, etc.**

🔄 Conexión: Implementan los puertos definidos antes

Tipo de pu	Lo implementa el...	Ejemplo

ert o		
Inp ut Po rt	**Adaptad or de entrada**	**Controller REST llama al caso de uso**
Ou tp ut Po rt	**Adaptad or de salida**	**Repositorio guarda en DB, servicio de email**

🎓 Analogía:

Piensa que el dominio y los casos de uso son como una fábrica.
Los adaptadores de entrada traen los pedidos a la fábrica.
Los adaptadores de salida son las máquinas que embalan, entregan o guardan lo que se produce.

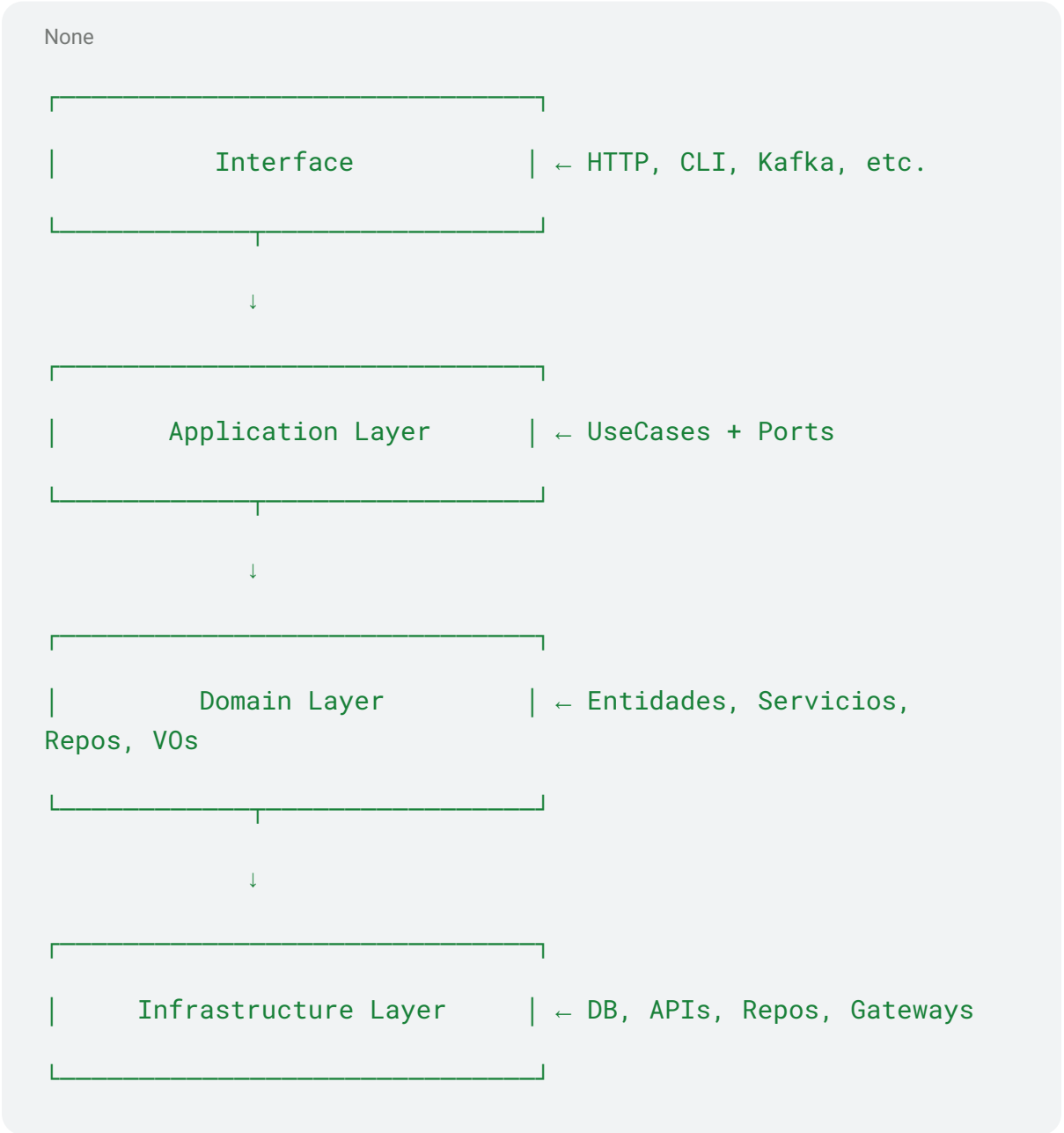
🧭 3. Separación de Capas en Arquitectura Hexagonal

La arquitectura hexagonal organiza el código **en capas bien definidas**, donde cada una tiene una responsabilidad específica y **comunican entre sí solo mediante interfaces (puertos)**.

🎯 Objetivo:

Mantener el sistema desacoplado, testeable, escalable y fácil de mantener.

📊 Vista general en forma de capas (de arriba hacia abajo):



🧩 1. Interface Layer (Capa de Entrada)

📍 ¿Qué hace?
Recibe la entrada del "mundo exterior" (usuario, red, CLI, eventos) y la transforma en una llamada al caso de uso correspondiente.

📌 Ejemplos:

- Controladores HTTP (`POST /users`)

- CLI (`register-user --name Andrés`)
- Kafka (`UsuarioRegistrado` evento)

🧠 **No tiene lógica de negocio. Solo traduce y delega.**

⚙️ 2. Application Layer (Capa de Aplicación o Casos de Uso)

📍 **¿Qué hace?**
Coordina las operaciones del sistema usando entidades del dominio.

📌 Contiene:

- Casos de uso: `RegisterUser`, `PlaceOrder`, etc.
- **Input Ports** (interfaces que exponen los casos de uso)
- **Output Ports** (interfaces que definen qué necesita el sistema: repos, email, etc.)

🔧 No sabe cómo se envía un correo o cómo se guarda en la base de datos. Solo sabe *que* necesita hacerlo, usando interfaces.

🧠 3. Domain Layer (Capa del Dominio)

📍 **¿Qué hace?**
Contiene la lógica **pura** del negocio. Es el **corazón** del sistema.

📌 Incluye:

- Entidades: `Usuario`, `Orden`, etc.
- Value Objects: `Email`, `Dinero`, etc.
- Reglas de negocio: validaciones, comportamiento
- Interfaces como `UserRepository` o `OrderPolicy`

✅ **No depende de ninguna librería externa.**
✅ **Es la capa más estable.**

🏗️ 4. Infrastructure Layer (Capa de Infraestructura)

📍 **¿Qué hace?**
Implementa los detalles técnicos definidos por los puertos de salida.

📌 Ejemplos:

- Repositorios que usan PostgreSQL, MongoDB
- Envío de correos con SMTP o SendGrid
- APIs externas (Stripe, Twilio)

⚠️ **Esta capa depende de tecnologías concretas**, pero está **fuera del núcleo del sistema**.

🎓 Analogía

Imagina una empresa:

- **Interface Layer**: Recepción o sitio web (recibe solicitudes)
- **Application Layer**: Gerente que coordina los procesos
- **Domain Layer**: Los expertos o ingenieros que conocen el negocio
- **Infrastructure Layer**: El correo, la fábrica, el banco — herramientas externas que ayudan, pero no son el corazón

---

🔄 Comunicación entre capas

✔️ **Solo comunican hacia abajo usando interfaces.**

Por ejemplo:

- El **controlador (interface)** llama a un **caso de uso (application)**
- El **caso de uso** utiliza un **repositorio o servicio de email (infra)** implementando un **puerto de salida**

❌ No se permite que la infraestructura llame directamente al dominio.

🧩 4. Estructura de Carpetas

JAVA

```
None
src
└── main
    └── java
        └── com
            └── tuempresa
                └── tuapp
                    ├── TuAppApplication.java      ← clase principal
(SpringBootApplication)

                    ├── domain                     ← 📌 Capa del Dominio
(Núcleo)
                    │   ├── model                  ← Entidades y Value
Objects
                    │   ├── repository             ← Interfaces (puertos) de
Repositorios
                    │   └── service                ← Servicios de Dominio
(reglas complejas)

                    ├── application                ← ⚙️ Casos de Uso
(UseCases)
                    │   ├── port                   ← Interfaces (puertos) de
entrada y salida
                    │   │   ├── input              ← Input Ports
                    │   │   └── output             ← Output Ports
                    │   └── usecase                ← Implementaciones de
casos de uso

                    ├── adapter                    ← 🔌 Adaptadores
(Interface e Infra)
                    │   ├── in                     ← Entradas (driving
adapters)
                    │   │   ├── web                ← Controladores REST
(Spring @RestController)
                    │   │   └── cli                ← Opcional: Entrada por
línea de comandos
                    │   └── out                    ← Salidas (driven
adapters)
                    │       ├── persistence        ← Repositorios JPA
                    │       └── external           ← APIs externas, envío de
emails, etc.
```

```
                    └── config                          ← 🧾 Configuración
(Beans, JPA, CORS, Swagger, etc.)
```

✅ Ventajas de esta estructura

- **Alta testabilidad**: Puedes testear casos de uso sin tocar controladores o la base de datos.
- **Bajo acoplamiento**: Infraestructura reemplazable (JPA, Mongo, Rabbit, etc.).
- **Alta cohesión**: Cada parte está donde debe estar.
- **Fácil mantenimiento**: Se ubican fácilmente los casos de uso, lógica de dominio y adaptadores.

C#

None

```
/MyApp
|
├── MyApp.Domain                # Núcleo del dominio (no depende de nada)
|    ├── Entities               # Entidades (User, Order, etc.)
|    ├── ValueObjects           # Objetos de Valor (Email, Money, etc.)
|    ├── Services               # Servicios de dominio (reglas de negocio
puras)
|    ├── Repositories           # Interfaces para acceder a entidades
|    └── Exceptions             # Excepciones específicas del dominio
|
├── MyApp.Application           # Lógica de aplicación (casos de uso)
|    ├── UseCases               # Casos de uso (RegisterUser, etc.)
|    ├── Ports                  # Puertos (interfaces)
|    |    ├── Input             # Input ports (definidos por la aplicación)
|    |    └── Output            # Output ports (ej. IEmailSender,
IUserRepository)
|    ├── DTOs                   # Objetos para intercambio de datos
|    └── Services               # Servicios de orquestación (coordinación
de UC)
|
├── MyApp.Infrastructure       # Adaptadores de salida (hacia afuera del
dominio)
```

```
|    ├── Persistence             # Implementación de Repos (con EF Core /
ADO.NET)
|    |    ├── SqlServer          # Conexión a SQL Server
|    |    └── Migrations         # Migraciones si usas EF Core
|    ├── Email                   # Cliente SMTP o servicios externos
|    └── Configuration           # Archivos y lógica de configuración
|
├── MyApp.GUI                   # Adaptador de entrada (interfaz gráfica)
|    ├── Views                   # Formularios o pantallas (WPF, WinForms)
|    ├── ViewModels              # ViewModels si usas MVVM
|    └── Controllers             # Manejan eventos y llaman a casos de uso
|
├── MyApp.Tests                 # Pruebas unitarias
|    ├── Domain.Tests            # Pruebas del dominio puro
|    ├── Application.Tests       # Pruebas de los casos de uso
|    └── Integration.Tests       # Pruebas integradas (con infraestructura)
|
└── MyApp.Shared                # Código compartido (utils, constantes,
helpers)
```

🧩 Dependencias (relación entre capas)

- `MyApp.Domain` **no depende de nada**.
- `MyApp.Application` **depende solo del dominio**.
- `MyApp.GUI` **usa los puertos (Input)** definidos en `Application`.
- `MyApp.Infrastructure` **implementa los puertos (Output)** definidos por `Application` y **depende de detalles técnicos** como Entity Framework, ADO.NET o librerías SMTP.

🛠️ Ejemplo de flujo RegisterUser

1. En la GUI, el usuario llena un formulario de registro.
2. El controlador de GUI llama al caso de uso `RegisterUser` a través del `InputPort`.
3. `RegisterUser` orquesta la lógica y llama al `UserRepository` (OutputPort).

4. `Infrastructure` implementa `UserRepository` y lo conecta a SQL Server.