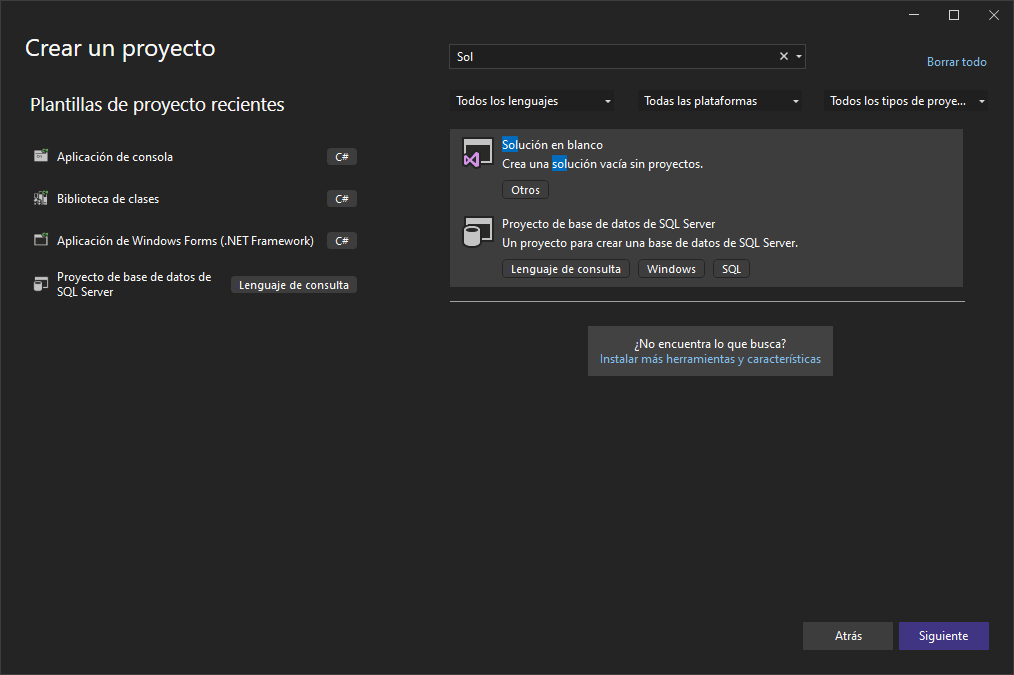
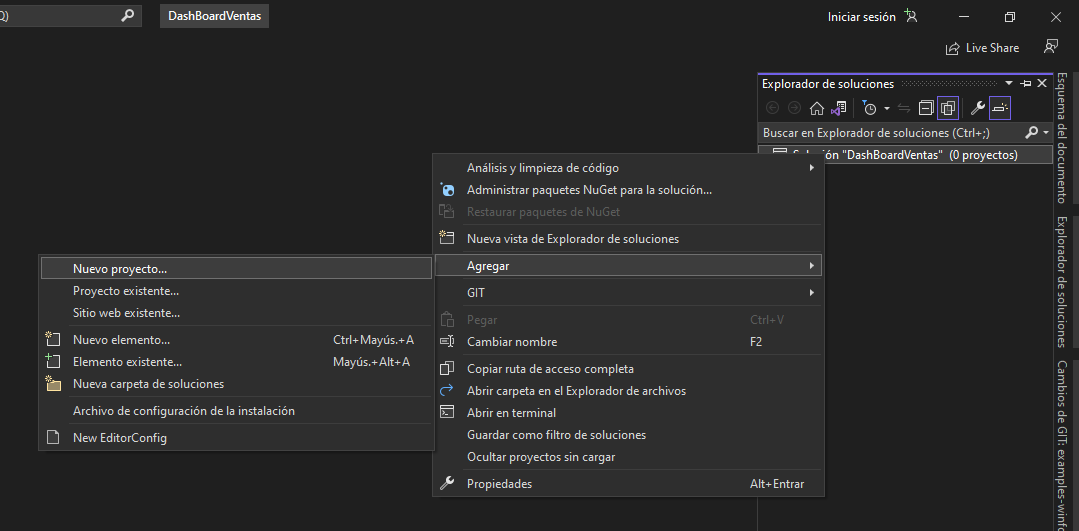
ESTRUCTURAS Y BUENAS PRÁCTICAS

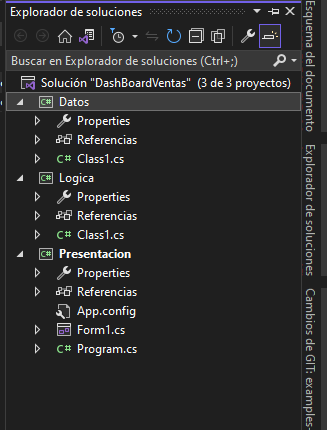
Conexión a la base de datos y 3 capas

1. Creamos una solución en blanco:



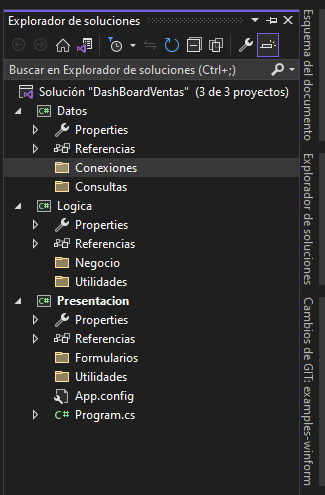
1. Y haciendo clic derecho vamos a agregar un nuevo proyecto de Windows form llamado presentación, otro biblioteca de clases llamado lógica y uno más biblioteca de clases llamado datos:

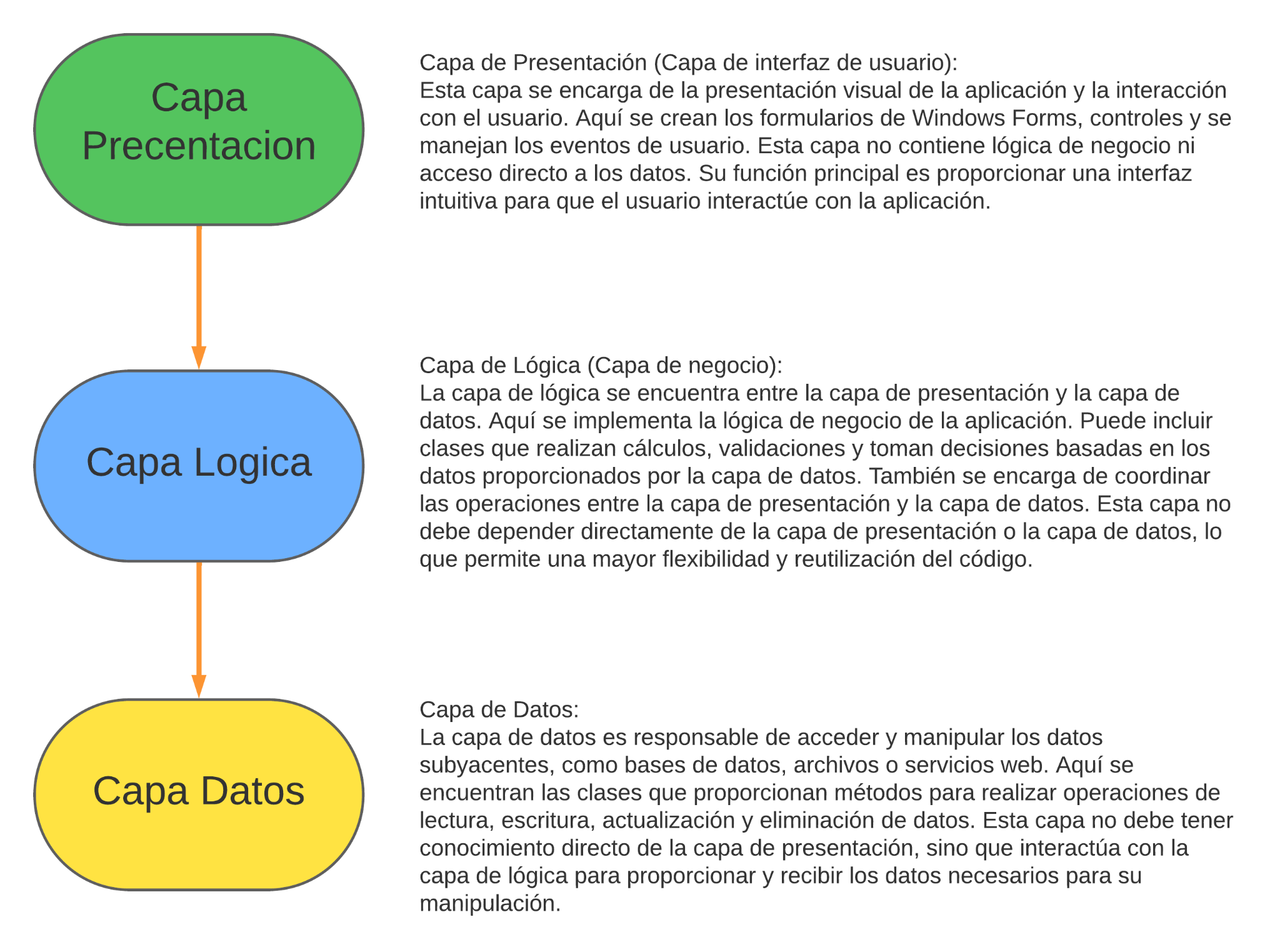
Debería quedar algo así:

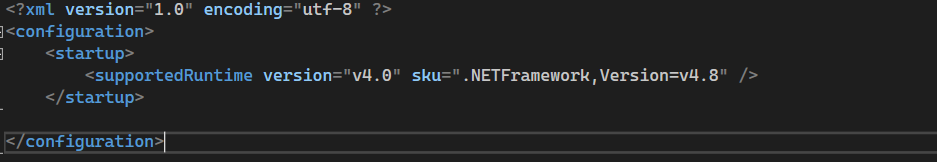


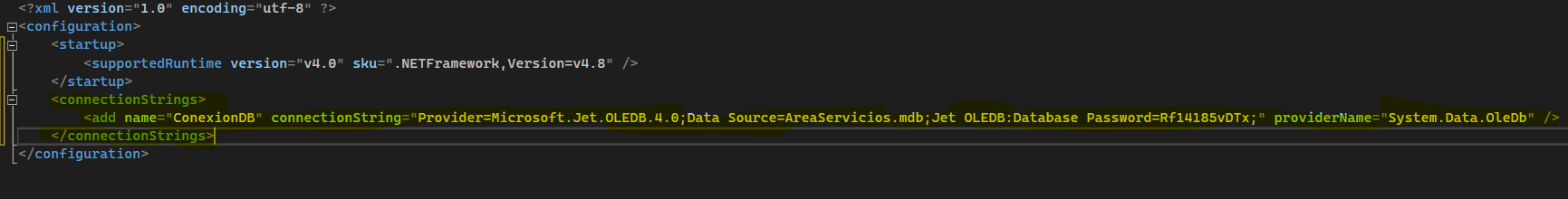
1. Borramos Class 1 de datos, Class 1 de lógica y Form1 de presentación.

Agregamos las siguientes carpetas para estructurar nuestro código:



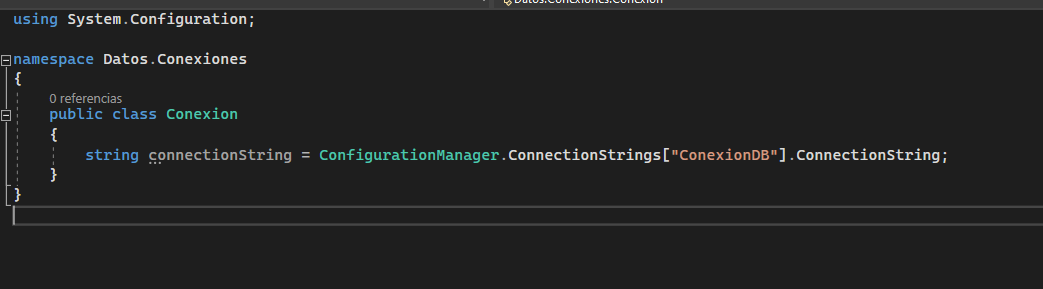
1. Estructura 3 capas: 
2. A continuación agregamos las referencias de la clase datos a la clase lógica y de la clase lógica la clase presentación.
3. String de conexión a la Base, se debe poner en el app config:





| * Colocar la cadena de conexión en el archivo de configuración es conveniente porque:  1. Centraliza la configuración de la aplicación. 2. Permite cambios flexibles sin recompilar. 3. Mejora la seguridad de la información confidencial. 4. Facilita la reutilización de la configuración en diferentes aplicaciones o componentes. |
| --- |

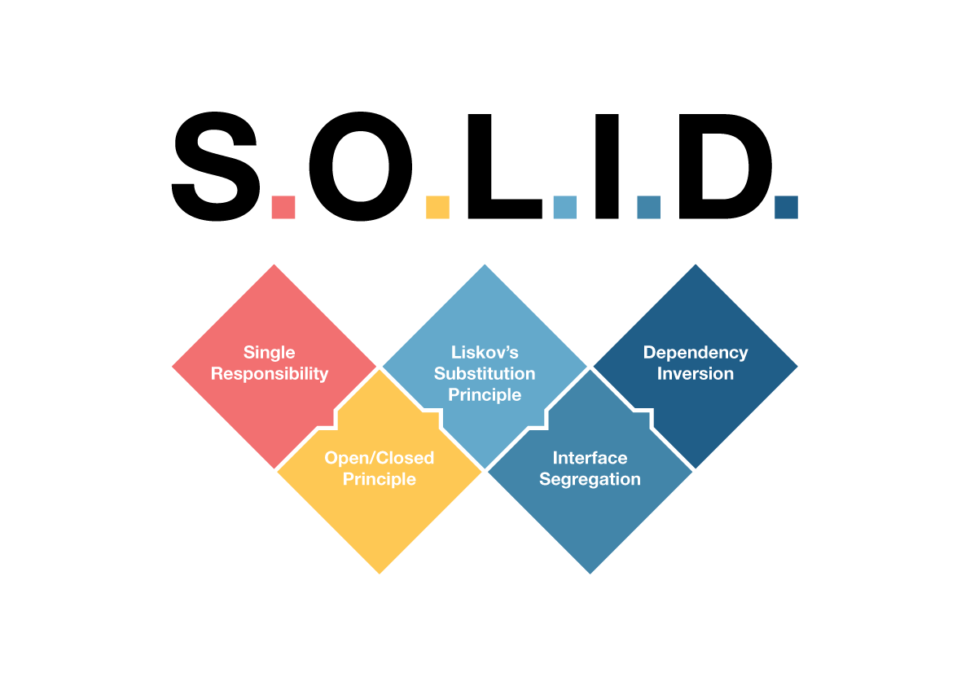
¿Cómo uso la cadena de conexión? En la capa de datos creamos una clase llamada conexión Dentro de la carpeta conexiones :



Aparece el configuration manager hacemos control. Sobre el nombre configuration manager e instalamos el paquete de nugget correspondiente.

De esta manera vamos a obtener el String de conexión.

Principios Solid



* Principio de Responsabilidad Única (SRP - Single Responsibility Principle): este principio establece que una clase debe tener una única responsabilidad. Debe haber una sola razón para que una clase cambie. Esto significa que una clase debe ser responsable de un único aspecto o funcionalidad dentro del sistema. Al adherirse a este principio, el código es más fácil de entender, mantener y reutilizar.
* Principio de Abierto/Cerrado (OCP - Open/Closed Principle): el OCP establece que las entidades de software (clases, módulos, etc.) deben estar abiertas para su extensión pero cerradas para su modificación. Esto significa que se deben poder agregar nuevas funcionalidades sin modificar el código existente. En lugar de modificar el código existente, se deben utilizar mecanismos de extensión como herencia, composición o implementación de interfaces.
* Principio de Sustitución de Liskov (LSP - Liskov Substitution Principle): el LSP establece que los objetos de una clase base deben ser reemplazables por objetos de sus clases derivadas sin alterar el correcto funcionamiento del programa. Esto significa que una clase derivada debe poder ser utilizada en lugar de la clase base sin generar errores o violar las expectativas del programa. Las clases derivadas deben cumplir con el contrato establecido por la clase base.
* Principio de Segregación de Interfaces (ISP - Interface Segregation Principle): el ISP establece que las interfaces deben ser específicas y cohesivas, en lugar de tener interfaces genéricas y sobrecargadas. Las interfaces deben contener solo los miembros que sean relevantes para las clases que las implementen. Esto evita la dependencia innecesaria de funcionalidades no utilizadas y permite un mejor diseño de las interfaces.
* Principio de Inversión de Dependencia (DIP - Dependency Inversion Principle): El DIP establece que los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones. Esto significa que las dependencias deben ser invertidas, de manera que los módulos de alto nivel dependan de interfaces o abstracciones en lugar de clases concretas. Esto promueve la flexibilidad, la reutilización y el desacoplamiento del código.