

# **UNIVERSIDAD JOSÉ CECILIO DEL VALLE**



**UNIVERSIDAD  
JOSE CECILIO DEL VALLE**

## **TEMA:**

Investigación N° 1 acerca de GitHub

## **ASIGNATURA:**

IIT2035 – ADM2012 Programación II

## **PRESENTADO POR:**

Alejandro Josué Díaz Zepeda, #2021220030

## **CATEDRÁTICO:**

Kevin Eduardo Fúnez Fúnez

## **FECHA:**

03 de julio del 2022

# INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

## TABLA DE CONTENIDO

INTRODUCCIÓN:.....	5
OBJETIVOS .....	6
Objetivo General:.....	6
Objetivos Específicos: .....	6
PATRONES DE DISEÑO EN C#.....	7
¿En qué Consiste?.....	7
Historia.....	8
Clasificación de los Patrones de Diseño: .....	8
1.      Patrones Credenciales .....	9
•          Abstract Factory:.....	9
•          Builder .....	10
•          Factory Method.....	11
•          Prototype .....	11
•          Singleton .....	11
2.      Patrones Estructurales.....	12
•          Adapter.....	12
•          Bridge.....	12
•          Composite .....	12
•          Decorator .....	13
•          Facade .....	13

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

• Flyweight .....	13
• Proxy .....	14
3. Patrones de Comportamiento.....	14
• Chain of Responsibility .....	14
• Command.....	15
• Iterator.....	15
• Mediator.....	16
• Memento .....	16
• Observer.....	16
• State .....	17
• Strategy .....	17
• Template Method.....	18
• Visitor .....	18
SOLID.....	19
S: Principio de responsabilidad única (SRP) .....	19
Ejemplo:.....	19
O: Principio abierto/cerrado.....	21
Ejemplo:.....	21
L: Principio de sustitución de Liskov .....	24
Ejemplo:.....	24
I: Principio de segregación de interfaz (ISP) .....	28

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

Ejemplo:	28
D: Principio de Inversión de Dependencia .....	31
Ejemplo:	31
CONCLUSIONES:.....	37
ANEXOS .....	38
Ejemplo Singleton.....	40
Ejemplo Prototype .....	41
Ejemplo Factory Method .....	41
Ejemplo Strategy.....	43
Ejemplo Mediator .....	46
Ejemplo State .....	49
Ejemplo Composite.....	52
Ejemplo Builder .....	53
Ejemplo Abstract Factory .....	56
Ejemplo Bridge .....	59
Ejemplo Adapter .....	61
Ejemplo Decorator .....	62
Ejemplo Facade.....	65
Ejemplo Flyweight.....	68
Ejemplo Proxy .....	71
Ejemplo Chain of Responsibility .....	73
Ejemplo Command .....	76

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

Ejemplo Iterator .....	79
Ejemplo Memento.....	82
Ejemplo Observer .....	86
Ejemplo Template Method .....	88
Ejemplo Visitor.....	91
<b>BIBLIOGRAFÍA .....</b>	<b>95</b>

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

### **INTRODUCCIÓN:**

En el siguiente documento se estará presentando toda la información relacionada a los diferentes patrones de diseño que podemos encontrar para la programación, específicamente aplicado al lenguaje de programación C#.

Se estará conceptualizando cada clase de patrón de diseño, al mismo proporcionando ejemplos específicos a modo de facilitar la comprensión de los mismos al igual que los SOLID.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

### OBJETIVOS

#### **Objetivo General:**

Comprender qué es los patrones de diseño y el SOLID para poder hacer uso de ellos mediante una explicación detallada de cómo usarla en C#, determinando sus comandos y conceptos relacionados a estos, de modo que se facilite su uso.

#### **Objetivos Específicos:**

- Definir patrones de diseño
- Dar ejemplos de patrones de diseño
- Explicar a qué hacer referencia el SOLID

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

### PATRONES DE DISEÑO EN C#

Los **patrones de diseño** son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en tu código.

No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular. Puedes seguir los detalles del patrón e implementar una solución que encaje con las realidades de tu propio programa.

A menudo los patrones se confunden con algoritmos porque ambos conceptos describen soluciones típicas a problemas conocidos. Mientras que un algoritmo siempre define un grupo claro de acciones para lograr un objetivo, un patrón es una descripción de más alto nivel de una solución. El código del mismo patrón aplicado a dos programas distintos puede ser diferente.

#### ¿En qué Consiste?

La mayoría de los patrones se describe con mucha formalidad para que la gente pueda reproducirlos en muchos contextos. Aquí tienes las secciones que suelen estar presentes en la descripción de un patrón:

El **propósito** del patrón explica brevemente el problema y la solución.

La **motivación** explica en más detalle el problema y la solución que brinda el patrón.

La **estructura** de las clases muestra cada una de las partes del patrón y el modo en que se relacionan.

El **ejemplo de código** en uno de los lenguajes de programación populares facilita la asimilación de la idea que se esconde tras el patrón.

Algunos catálogos de patrones enumeran otros detalles útiles, como la aplicabilidad del patrón, los pasos de implementación y las relaciones con otros patrones.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

### Historia

El concepto de los patrones fue descrito por Christopher Alexander en El lenguaje de patrones. El libro habla de un “lenguaje” para diseñar el entorno urbano. Las unidades de este lenguaje son los patrones. Pueden describir lo altas que tienen que ser las ventanas, cuántos niveles debe tener un edificio, cuan grandes deben ser las zonas verdes de un barrio, etcétera.

La idea fue recogida por cuatro autores: Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm. En 1995, publicaron Patrones de diseño, en el que aplicaron el concepto de los patrones de diseño a la programación. El libro presentaba 23 patrones que resolvían varios problemas del diseño orientado a objetos y se convirtió en un éxito de ventas con rapidez. Al tener un título tan largo en inglés, la gente empezó a llamarlo “el libro de la ‘gang of four’ (banda de los cuatro)”, lo que pronto se abrevió a “el libro Gof”.

Desde entonces se han descubierto decenas de nuevos patrones orientados a objetos. La “metodología del patrón” se hizo muy popular en otros campos de la programación, por lo que hoy en día existen muchos otros patrones no relacionados con el diseño orientado a objetos.

### Clasificación de los Patrones de Diseño:

1. Los **patrones creacionales** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
2. Los **patrones estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
3. Los **patrones de comportamiento** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

### 1. Patrones Credenciales

Dentro de los patrones credenciales podemos encontrar los siguientes:

- **Abstract Factory:**

Permite producir familias de objetos relacionados sin especificar sus clases concretas.

#### Aplicación

*Problema:* Imagina que estás creando un simulador de tienda de muebles. Tu código está compuesto por clases que representan lo siguiente:

- i) Una familia de productos relacionados, digamos: Silla + Sofá + Mesilla.
- ii) Algunas variantes de esta familia. Por ejemplo, los productos Silla + Sofá + Mesilla están disponibles en estas variantes: Moderna, Victoriana, ArtDecó. Ver imagen 1.1.

Necesitamos una forma de crear objetos individuales de mobiliario para que combinen con otros objetos de la misma familia. Los clientes se enfadan bastante cuando reciben muebles que no combinan. Ver Imagen 1.2.

Además, no queremos cambiar el código existente al añadir al programa nuevos productos o familias de productos. Los comerciantes de muebles actualizan sus catálogos muy a menudo, y debemos evitar tener que cambiar el código principal cada vez que esto ocurra.

*Solución:* Lo primero que sugiere el patrón Abstract Factory es que declaremos de forma explícita interfaces para cada producto diferente de la familia de productos (por ejemplo, silla, sofá o mesilla). Después podemos hacer que todas las variantes de los productos sigan esas interfaces. Por ejemplo, todas las variantes de silla pueden implementar la interfaz Silla, así como todas las variantes de mesilla pueden implementar la interfaz Mesilla, y así sucesivamente. Ver imagen 1.3.

El siguiente paso consiste en declarar la *Fábrica abstracta*: una interfaz con una lista de métodos de creación para todos los productos que son parte de la familia de productos (por ejemplo, crearSilla, crearSofá y crearMesilla). Estos métodos deben devolver productos **abstractos** representados por las interfaces que extrajimos previamente: Silla, Sofá, Mesilla, etc. Ver imagen 1.4.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

Ahora bien, ¿qué hay de las variantes de los productos? Para cada variante de una familia de productos, creamos una clase de fábrica independiente basada en la interfaz `FábricaAbstracta`. Una fábrica es una clase que devuelve productos de un tipo particular. Por ejemplo, la `FábricadeMueblesModernos` sólo puede crear objetos de `SillaModerna`, `SofáModerno` y `MesillaModerna`.

El código cliente tiene que funcionar con fábricas y productos a través de sus respectivas interfaces abstractas. Esto nos permite cambiar el tipo de fábrica que pasamos al código cliente, así como la variante del producto que recibe el código cliente, sin descomponer el propio código cliente. Ver imagen 1.5.

Digamos que el cliente quiere una fábrica para producir una silla. El cliente no tiene que conocer la clase de la fábrica y tampoco importa el tipo de silla que obtiene. Ya sea un modelo moderno o una silla de estilo victoriano, el cliente debe tratar a todas las sillas del mismo modo, utilizando la interfaz abstracta `Silla`. Con este sistema, lo único que sabe el cliente sobre la silla es que implementa de algún modo el método `sentarse`. Además, sea cual sea la variante de silla devuelta, siempre combinará con el tipo de sofá o mesilla producida por el mismo objeto de fábrica.

Queda otro punto por aclarar: si el cliente sólo está expuesto a las interfaces abstractas, ¿cómo se crean los objetos de fábrica? Normalmente, la aplicación crea un objeto de fábrica concreto en la etapa de inicialización. Justo antes, la aplicación debe seleccionar el tipo de fábrica, dependiendo de la configuración o de los ajustes del entorno.

- ***Builder***

Permite construir objetos complejos paso a paso. Este patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

### *Aplicación*

- Para evitar un “constructor telescopico”
- cuando quieras que el código sea capaz de crear distintas representaciones de ciertos productos (por ejemplo, casas de piedra y madera)
- Para construir árboles con el patrón *Composite* u otros objetos complejos.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

- ***Factory Method***

Proporciona una interfaz para la creación de objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

### Aplicación

- Cuando no conozcas de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar tu código.
- Cuando quieras ofrecer a los usuarios de tu biblioteca o framework, una forma de extender sus componentes internos.
- Cuando quieras ahorrar recursos del sistema mediante la reutilización de objetos existentes en lugar de reconstruirlos cada vez.

- ***Prototype***

Permite copiar objetos existentes sin que el código dependa de sus clases.

### Aplicación

- Cuando tu código no deba depender de las clases concretas de objetos que necesites copiar.
- Cuando quieras reducir la cantidad de subclases que solo se diferencian en la forma en que inicializan sus respectivos objetos. Puede ser que alguien haya creado estas subclases para poder crear objetos con una configuración específica.

- ***Singleton***

Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

### Aplicación

- Utiliza el patrón Singleton cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes; por ejemplo, un único objeto de base de datos compartido por distintas partes del programa.
- Utiliza el patrón Singleton cuando necesites un control más estricto de las variables globales.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

### 2. Patrones Estructurales

Dentro de los patrones estructurales podemos encontrar los siguientes:

- ***Adapter***

Permite la colaboración entre objetos con interfaces incompatibles.

#### Aplicación

- Utiliza la clase adaptadora cuando quieras usar una clase existente, pero cuya interfaz no sea compatible con el resto del código.
- Utiliza el patrón cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase.

- ***Bridge***

Permite dividir una clase grande o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.

#### Aplicación

- Utiliza el patrón Bridge cuando quieras dividir y organizar una clase monolítica que tenga muchas variantes de una sola funcionalidad (por ejemplo, si la clase puede trabajar con diversos servidores de bases de datos).
- Utiliza el patrón cuando necesites extender una clase en varias dimensiones ortogonales (independientes).
- Utiliza el patrón Bridge cuando necesites poder cambiar implementaciones durante el tiempo de ejecución.

- ***Composite***

Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

#### Aplicación

- Utiliza el patrón Composite cuando tengas que implementar una estructura de objetos con forma de árbol.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

- Utiliza el patrón cuando quieras que el código cliente trate elementos simples y complejos de la misma forma.
- ***Decorator***

Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.
- Aplicación**
  - Utiliza el patrón Decorator cuando necesites asignar funcionalidades adicionales a objetos durante el tiempo de ejecución sin descomponer el código que utiliza esos objetos.
  - Utiliza el patrón cuando resulte extraño o no sea posible extender el comportamiento de un objeto utilizando la herencia.
- ***Facade***

Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.
- Aplicación**
  - Utiliza el patrón Facade cuando necesites una interfaz limitada pero directa a un subsistema complejo.
  - Utiliza el patrón Facade cuando quieras estructurar un subsistema en capas.
- ***Flyweight***

Permite mantener más objetos dentro de la cantidad disponible de memoria RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.
- Aplicación**
  - Utiliza el patrón Flyweight únicamente cuando tu programa deba soportar una enorme cantidad de objetos que apenas quepan en la RAM disponible.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

- ***Proxy***

Permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.

### ***Aplicación***

- Inicialización diferida (proxy virtual). Es cuando tienes un objeto de servicio muy pesado que utiliza muchos recursos del sistema al estar siempre funcionando, aunque solo lo necesites de vez en cuando.
- Control de acceso (proxy de protección). Es cuando quieres que únicamente clientes específicos sean capaces de utilizar el objeto de servicio, por ejemplo, cuando tus objetos son partes fundamentales de un sistema operativo y los clientes son varias aplicaciones lanzadas (incluyendo maliciosas).
- Ejecución local de un servicio remoto (proxy remoto). Es cuando el objeto de servicio se ubica en un servidor remoto.
- Solicitudes de registro (proxy de registro). Es cuando quieres mantener un historial de solicitudes al objeto de servicio.
- Resultados de solicitudes en caché (proxy de caché). Es cuando necesitas guardar en caché resultados de solicitudes de clientes y gestionar el ciclo de vida de ese caché, especialmente si los resultados son muchos.
- Referencia inteligente. Es cuando debes ser capaz de desechar un objeto pesado una vez que no haya clientes que lo utilicen.

### **3. Patrones de Comportamiento**

Dentro de los patrones credenciales podemos encontrar los siguientes:

- ***Chain of Responsibility***

Permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

### ***Aplicación***

- Utiliza el patrón Chain of Responsibility cuando tu programa deba procesar distintos tipos de solicitudes de varias maneras, pero los tipos exactos de

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

solicitudes y sus secuencias no se conozcan de antemano.

- Utiliza el patrón cuando sea fundamental ejecutar varios manejadores en un orden específico.
- Utiliza el patrón Chain of Responsibility cuando el grupo de manejadores y su orden deban cambiar durante el tiempo de ejecución.

- ***Command***

Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.

### ***Aplicación***

- Utiliza el patrón Command cuando quieras parametrizar objetos con operaciones.
- Utiliza el patrón Command cuando quieras poner operaciones en cola, programar su ejecución, o ejecutarlas de forma remota.
- Utiliza el patrón Command cuando quieras implementar operaciones reversibles.

- ***Iterator***

Permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).

### ***Aplicación***

- Utiliza el patrón Iterator cuando tu colección tenga una estructura de datos compleja a nivel interno, pero quieras ocultar su complejidad a los clientes (ya sea por conveniencia o por razones de seguridad).
- Utiliza el patrón para reducir la duplicación en el código de recorrido a lo largo de tu aplicación.
- Utiliza el patrón Iterator cuando quieras que tu código pueda recorrer distintas estructuras de datos, o cuando los tipos de estas estructuras no se conozcan de antemano.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

- ***Mediator***

Permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.

### Aplicación

- Utiliza el patrón Mediator cuando resulte difícil cambiar algunas de las clases porque están fuertemente acopladas a un puñado de otras clases.
- Utiliza el patrón cuando no puedas reutilizar un componente en un programa diferente porque sea demasiado dependiente de otros componentes.
- Utiliza el patrón Mediator cuando te encuentres creando cientos de subclases de componente sólo para reutilizar un comportamiento básico en varios contextos.

- ***Memento***

Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.

### Aplicación

- Utiliza el patrón Memento cuando quieras producir instantáneas del estado del objeto para poder restaurar un estado previo del objeto.
- Utiliza el patrón cuando el acceso directo a los campos, consultores o modificadores del objeto viole su encapsulación.

- ***Observer***

Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

### Aplicación

- Utiliza el patrón Observer cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.
- Utiliza el patrón cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

### • *State*

Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.

#### *Aplicación*

- Utiliza el patrón State cuando tengas un objeto que se comporta de forma diferente dependiendo de su estado actual, el número de estados sea enorme y el código específico del estado cambie con frecuencia.
- Utiliza el patrón cuando tengas una clase contaminada con enormes condicionales que alteran el modo en que se comporta la clase de acuerdo con los valores actuales de los campos de la clase.
- Utiliza el patrón State cuando tengas mucho código duplicado por estados similares y transiciones de una máquina de estados basada en condiciones.

### • *Strategy*

Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

#### *Aplicación*

- Utiliza el patrón Strategy cuando quieras utilizar distintas variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.
- Utiliza el patrón Strategy cuando tengas muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.
- Utiliza el patrón para aislar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan importantes en el contexto de esa lógica.
- Utiliza el patrón cuando tu clase tenga un enorme operador condicional que cambie entre distintas variantes del mismo algoritmo.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

- ***Template Method***

Define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

### ***Aplicación***

- Utiliza el patrón Template Method cuando quieras permitir a tus clientes que extiendan únicamente pasos particulares de un algoritmo, pero no todo el algoritmo o su estructura.
- Utiliza el patrón cuando tengas muchas clases que contengan algoritmos casi idénticos, pero con algunas diferencias mínimas. Como resultado, puede que tengas que modificar todas las clases cuando el algoritmo cambie.

- ***Visitor***

Permite separar algoritmos de los objetos sobre los que operan.

### ***Aplicación***

- Utiliza el patrón Visitor cuando necesites realizar una operación sobre todos los elementos de una compleja estructura de objetos (por ejemplo, un árbol de objetos).
- Utiliza el patrón Visitor para limpiar la lógica de negocio de comportamientos auxiliares.
- Utiliza el patrón cuando un comportamiento solo tenga sentido en algunas clases de una jerarquía de clases, pero no en otras.

## SOLID

Los principios SOLID son los principios de diseño que nos permiten gestionar la mayoría de los problemas de diseño de software. Robert C. Martin compiló estos principios en la década de 1990. Estos principios nos brindan formas de pasar de un código estrechamente acoplado y poca encapsulación a los resultados deseados de las necesidades reales de un negocio débilmente acopladas y encapsuladas adecuadamente. SOLID es un acrónimo de lo siguiente.

- S: Principio de responsabilidad única (SRP)
- O: Principio abierto cerrado (OCP)
- L: Principio de sustitución de Liskov (LSP)
- I: Principio de segregación de interfaz (ISP)
- D: Principio de Inversión de Dependencia (DIP)

### **S: Principio de responsabilidad única (SRP)**

SRP dice "Cada módulo de software debe tener una sola razón para cambiar". Esto significa que cada clase, o estructura similar, en su código debe tener solo un trabajo que hacer. Todo en esa clase debe estar relacionado con un solo propósito. Nuestra clase no debe ser como una navaja suiza en la que, si es necesario cambiar una de ellas, es necesario modificar toda la herramienta. No significa que sus clases solo deban contener un método o propiedad. Puede haber muchos miembros siempre que se relacionen con una sola responsabilidad.

El principio de responsabilidad única nos brinda una buena manera de identificar clases en la fase de diseño de una aplicación y te hace pensar en todas las formas en que una clase puede cambiar. Una buena separación de responsabilidades se realiza solo cuando tenemos el panorama completo de cómo debe funcionar la aplicación.

*Ejemplo:*

```

1. public class UserService
2. {
3.     public void Register(string email, string password)
4.     {
5.         if (!ValidateEmail(email))
6.             throw new ValidationException("Email is not an email");
7.         var user = new User(email, password);

```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
8.          SendEmail(new MailMessage("mysite@nowhere.com", email) {
9.              Subject="Hello foo" });
10.         }
11.         public virtual bool ValidateEmail(string email)
12.         {
13.             return email.Contains("@");
14.         }
15.         public bool SendEmail(MailMessage message)
16.         {
17.             _smtpClient.Send(message);
18.         }
19.     }
```

Se ve bien, pero no sigue SRP. Los métodos SendEmail y ValidateEmail no tienen nada que hacer dentro de la clase UserService. Vamos a refractarlo.

```
20. public class UserService
21. {
22.     EmailService _emailService;
23.     DbContext _dbContext;
24.     public UserService(EmailService aEmailService, DbContext aDbContext)
25.     {
26.         _emailService = aEmailService;
27.         _dbContext = aDbContext;
28.     }
29.     public void Register(string email, string password)
30.     {
31.         if (!_emailService.ValidateEmail(email))
32.             throw new ValidationException("Email is not an email");

33.         var user = new User(email, password);
34.         _dbContext.Save(user);
35.         _emailService.SendEmail(new MailMessage("myname@mydomain
36.             .com", email) {Subject="Hi. How are you!"});
37.     }
38. }
39. public class EmailService
40. {
41.     SmtpClient _smtpClient;
42.     public EmailService(SmtpClient aSmtpClient)
43.     {
44.         _smtpClient = aSmtpClient;
45.     }
46.     public bool virtual ValidateEmail(string email)
47.     {
48.         return email.Contains("@");
```

```

49.    }
50.    public bool SendEmail(MailMessage message)
51.    {
52.        _smtpClient.Send(message);
53.    }
54. }
```

**O: Principio abierto/cerrado**

El principio abierto/cerrado dice: "Un módulo/clase de software está abierto para extensión y cerrado para modificación".

Aquí "Abierto para extensión" significa que necesitamos diseñar nuestro módulo/clase de tal manera que la nueva funcionalidad se pueda agregar solo cuando se generen nuevos requisitos. "Cerrado por modificación" significa que ya hemos desarrollado una clase y ha pasado por pruebas unitarias. Entonces no deberíamos alterarlo hasta que encontremos errores. Como dice, una clase debe estar abierta para extensiones, podemos usar la herencia para hacer esto.

**Ejemplo:**

Supongamos que tenemos una clase Rectángulo con las propiedades Alto y Ancho.

```

1. public class Rectangle{
2.     public double Height {get;set;}
3.     public double Wight {get;set; }
4. }
```

Nuestra aplicación necesita la capacidad de calcular el área total de una colección de rectángulos. Como ya aprendimos el principio de responsabilidad única (SRP), no necesitamos poner el código de cálculo del área total dentro del rectángulo. Así que aquí creé otra clase para el cálculo del área.

```

1. public class AreaCalculator {
2.     public double TotalArea(Rectangle[] arrRectangles)
3.     {
4.         double area;
5.         foreach(var objRectangle in arrRectangles)
6.         {
7.             area += objRectangle.Height * objRectangle.Width;
8.         }
9.         return area;
10.    }
11. }
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

Hicimos. Hicimos nuestra aplicación sin violar SRP. No hay problemas por ahora. Pero, ¿podemos extender nuestra aplicación para que pueda calcular el área no solo de los rectángulos sino también del área de los círculos? Ahora tenemos un problema con el cálculo del área porque la forma de calcular el área del círculo es diferente. No es un gran trato. Podemos cambiar un poco el método TotalArea para que pueda aceptar una matriz de objetos como argumento. Verificamos el tipo de objeto en el bucle y hacemos el cálculo del área en función del tipo de objeto.

```
1. public class Rectangle{
2.     public double Height {get;set;}
3.     public double Wight {get;set; }
4. }
5. public class Circle{
6.     public double Radius {get;set;}
7. }
8. public class AreaCalculator
9. {
10.     public double TotalArea(object[] arrObjects)
11.     {
12.         double area = 0;
13.         Rectangle objRectangle;
14.         Circle objCircle;
15.         foreach(var obj in arrObjects)
16.         {
17.             if(obj is Rectangle)
18.             {
19.                 area += obj.Height * obj.Width;
20.             }
21.             else
22.             {
23.                 objCircle = (Circle)obj;
24.                 area += objCircle.Radius * objCircle.Radius * Math.P
I;
25.             }
26.         }
27.         return area;
28.     }
29. }
```

Terminamos con el cambio. Aquí introdujimos con éxito Circle en nuestra aplicación. Podemos agregar un Triángulo y calcular su área agregando un bloque "if" más en el método TotalArea de AreaCalculator. Pero cada vez que introducimos una nueva forma, necesitamos modificar el método TotalArea. Por lo tanto, la clase AreaCalculator no está cerrada para modificaciones. ¿Cómo podemos hacer nuestro diseño para evitar esta situación? En general, podemos hacer esto haciendo referencia a abstracciones para dependencias, como interfaces o

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

clases abstractas, en lugar de usar clases concretas. Dichas interfaces se pueden arreglar una vez desarrolladas para que las clases que dependen de ellas puedan confiar en abstracciones inmutables. Se puede agregar funcionalidad creando nuevas clases que implementen las interfaces. Así que refractemos nuestro código usando una interfaz.

```
1. public abstract class Shape  
2. {  
3.     public abstract double Area();  
4. }
```

Al heredar de Shape, las clases Rectangle y Circle ahora se ven así:

```
1. public class Rectangle: Shape  
2. {  
3.     public double Height {get;set;}  
4.     public double Width {get;set;}  
5.     public override double Area()  
6.     {  
7.         return Height * Width;  
8.     }  
9. }  
10. public class Circle: Shape  
11. {  
12.     public double Radius {get;set;}  
13.     public override double Area()  
14.     {  
15.         return Radius * Radus * Math.PI;  
16.     }  
17. }
```

Cada forma contiene su área con su propia forma de funcionalidad de cálculo y nuestra clase AreaCalculator será más simple que antes.

```
1. public class AreaCalculator  
2. {  
3.     public double TotalArea(Shape[] arrShapes)  
4.     {  
5.         double area=0;  
6.         foreach(var objShape in arrShapes)  
7.         {  
8.             area += objShape.Area();  
9.         }  
10.        return area;  
11.    }  
12. }
```

## L: Principio de sustitución de Liskov

El principio de sustitución de Liskov (LSP) establece que "debe poder usar cualquier clase derivada en lugar de una clase principal y hacer que se comporte de la misma manera sin modificaciones". Asegura que una clase derivada no afecte el comportamiento de la clase padre, en otras palabras, que una clase derivada debe ser sustituible por su clase base.

Este principio es solo una extensión del Principio Abierto Cerrado y significa que debemos asegurarnos de que las nuevas clases derivadas amplíen las clases base sin cambiar su comportamiento. Explicaré esto con un ejemplo del mundo real que viola LSP.

Un padre es médico mientras que su hijo quiere convertirse en jugador de críquet. Entonces aquí el hijo no puede reemplazar a su padre aunque ambos pertenezcan a la misma jerarquía familiar.

### Ejemplo:

Supongamos que necesitamos crear una aplicación para administrar datos usando un grupo de archivos de texto SQL. Aquí necesitamos escribir la funcionalidad para cargar y guardar el texto de un grupo de archivos SQL en el directorio de la aplicación. Entonces, necesitamos una clase que administre la carga y guarde el texto del grupo de archivos SQL junto con la clase SqlFile.

```

1. public class SqlFile
2. {
3.     public string FilePath {get;set;}
4.     public string FileText {get;set;}
5.     public string LoadText()
6.     {
7.         /* Code to read text from sql file */
8.     }
9.     public string SaveText()
10.    {
11.        /* Code to save text into sql file */
12.    }
13. }
14. public class SqlFileManager
15. {
16.     public List<SqlFile> lstSqlFiles {get;set;}
17.
18.     public string GetTextFromFiles()
19.     {
20.         StringBuilder objStrBuilder = new StringBuilder();
21.         foreach(var objFile in lstSqlFiles)
22.         {
23.             objStrBuilder.Append(objFile.LoadText());

```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
24.     }
25.     return objStrBuilder.ToString();
26. }
27. public void SaveTextIntoFiles()
28. {
29.     foreach(var objFile in lstSqlFiles)
30.     {
31.         objFile.SaveText();
32.     }
33. }
34. }
```

Hemos terminado con nuestra parte. La funcionalidad se ve bien por ahora. Después de un tiempo, nuestros líderes pueden decírnos que es posible que tengamos algunos archivos de solo lectura en la carpeta de la aplicación, por lo que debemos restringir el flujo cada vez que intente guardarlos.

Podemos hacerlo creando una clase "ReadOnlySqlFile" que hereda la clase "SqlFile" y necesitamos modificar el método SaveTextIntoFiles() introduciendo una condición para evitar llamar al método SaveText() en las instancias de ReadOnlySqlFile.

```
1. public class SqlFile
2. {
3.     public string LoadText()
4.     {
5.         /* Code to read text from sql file */
6.     }
7.     public void SaveText()
8.     {
9.         /* Code to save text into sql file */
10.    }
11. }
12. public class ReadOnlySqlFile: SqlFile
13. {
14.     public string FilePath {get;set;}
15.     public string FileText {get;set;}
16.     public string LoadText()
17.     {
18.         /* Code to read text from sql file */
19.     }
20.     public void SaveText()
21.     {
22.         /* Throw an exception when app flow tries to do save. */
23.         throw new IOException("Can't Save");
24.     }
25. }
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

Para evitar una excepción, debemos modificar "SqlFileManager" agregando una condición al bucle.

```
26.  public class SqlFileManager
27.  {
28.      public List<SqlFile?> lstSqlFiles {get;set}
29.      public string GetTextFromFiles()
30.      {
31.          StringBuilder objStrBuilder = new StringBuilder();
32.          foreach(var objFile in lstSqlFiles)
33.          {
34.              objStrBuilder.Append(objFile.LoadText());
35.          }
36.          return objStrBuilder.ToString();
37.      }
38.      public void SaveTextIntoFiles()
39.      {
40.          foreach(var objFile in lstSqlFiles)
41.          {
42.              //Check whether the current file object is read-
43.              //only or not. If yes, skip calling it's
44.              // SaveText() method to skip the exception.
45.              if(! objFile is ReadOnlySqlFile)
46.                  objFile.SaveText();
47.          }
48.      }
49.  }
```

Aquí modificamos el método SaveTextIntoFiles() en la clase SqlFileManager para determinar si la instancia es o no de ReadOnlySqlFile para evitar la excepción. No podemos usar esta clase ReadOnlySqlFile como sustituto de su padre sin alterar el código SqlFileManager. Entonces podemos decir que este diseño no sigue LSP. Hagamos que este diseño siga el LSP. Aquí presentaremos interfaces para independizar la clase SqlFileManager del resto de los bloques.

```
1. public interface IReadableSqlFile
2. {
3.     string LoadText();
4. }
5. public interface IWritableSqlFile
6. {
7.     void SaveText();
8. }
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

Ahora implementamos IReadableSqlFile a través de la clase ReadOnlySqlFile que lee solo el texto de los archivos de solo lectura.

```
1. public class ReadOnlySqlFile: IReadableSqlFile
2. {
3.     public string FilePath {get;set;}
4.     public string FileText {get;set;}
5.     public string LoadText()
6.     {
7.         /* Code to read text from sql file */
8.     }
9. }
```

Aquí implementamos tanto IWritableSqlFile como IReadableSqlFile en una clase SqlFile mediante la cual podemos leer y escribir archivos.

```
1. public class SqlFile: IWritableSqlFile, IReadableSqlFile
2. {
3.     public string FilePath {get;set;}
4.     public string FileText {get;set;}
5.     public string LoadText()
6.     {
7.         /* Code to read text from sql file */
8.     }
9.     public void SaveText()
10.    {
11.        /* Code to save text into sql file */
12.    }
13. }
```

Ahora el diseño de la clase SqlFileManager queda así:

```
1. public class SqlFileManager
2. {
3.     public string GetTextFromFiles(List<IReadableSqlFile> aLstReada
bleFiles)
4.     {
5.         StringBuilder objStrBuilder = new StringBuilder();
6.         foreach(var objFile in aLstReadableFiles)
7.         {
8.             objStrBuilder.Append(objFile.LoadText());
9.         }
10.        return objStrBuilder.ToString();
11.    }
12.    public void SaveTextIntoFiles(List<IWritableSqlFile> aLstWrit
ableFiles)
13.    {
14.        foreach(var objFile in aLstWritableFiles)
```

```
15.    {
16.        objFile.SaveText();
17.    }
18. }
19. }
```

Aquí, el método GetTextFromFiles() solo obtiene la lista de instancias de clases que implementan la interfaz IReadonlySqlFile. Eso significa las instancias de clase SqlFile y ReadonlySqlFile. Y el método SaveTextIntoFiles() obtiene solo las instancias de la lista de la clase que implementa la interfaz IWritableSqlFiles, en otras palabras, las instancias de SqlFile en este caso. Ahora podemos decir que nuestro diseño sigue el LSP. Y solucionamos el problema utilizando el principio de segregación de interfaz mediante (ISP) identificando la abstracción y el método de separación de responsabilidad.

### I: Principio de segregación de interfaz (ISP)

El Principio de segregación de la interfaz establece que "no se debe obligar a los clientes a implementar interfaces que no usan. En lugar de una interfaz gorda, se prefieren muchas interfaces pequeñas en función de grupos de métodos, cada uno sirviendo a un submódulo".

Podemos definirlo de otra manera. Una interfaz debe estar más relacionada con el código que la usa que con el código que la implementa. Entonces, los métodos en la interfaz se definen por qué métodos necesita el código del cliente en lugar de qué métodos implementa la clase. Por lo tanto, no se debe obligar a los clientes a depender de interfaces que no utilizan.

Al igual que las clases, cada interfaz debe tener un propósito/responsabilidad específicos (consulte SRP). No debería verse obligado a implementar una interfaz cuando su objeto no comparte ese propósito. Cuanto más grande sea la interfaz, más probable es que incluya métodos que no todos los implementadores pueden hacer. Esa es la esencia del Principio de Segregación de la Interfaz.

#### *Ejemplo:*

Supongamos que necesitamos crear un sistema para una empresa de TI que contenga roles como TeamLead y Programmer donde TeamLead divide una tarea enorme en tareas más pequeñas y las asigna a sus programadores o puede trabajar directamente en ellas.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

Según las especificaciones, necesitamos crear una interfaz y una clase TeamLead para implementarla.

---

---

```
1. public Interface ILead
2. {
3.     void CreateSubTask();
4.     void AssginTask();
5.     void WorkOnTask();
6. }
7. public class TeamLead : ILead
8. {
9.     public void AssignTask()
10.    {
11.        //Code to assign a task.
12.    }
13.    public void CreateSubTask()
14.    {
15.        //Code to create a sub task
16.    }
17.    public void WorkOnTask()
18.    {
19.        //Code to implement perform assigned task.
20.    }
21. }
```

El diseño se ve bien por ahora. Más tarde, se introduce en el sistema otro rol como Gerente, que asigna tareas a TeamLead y no trabajará en las tareas. ¿Podemos implementar directamente una interfaz ILead en la clase Manager, como la siguiente?

```
1. public class Manager: ILead
2. {
3.     public void AssignTask()
4.     {
5.         //Code to assign a task.
6.     }
7.     public void CreateSubTask()
8.     {
9.         //Code to create a sub task.
10.    }
11.    public void WorkOnTask()
12.    {
13.        throw new Exception("Manager can't work on Task");
14.    }
15. }
```

Dado que el Gerente no puede trabajar en una tarea y, al mismo tiempo, nadie puede asignar tareas

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

al Gerente, este WorkOnTask() no debe estar en la clase Gerente. Pero estamos implementando esta clase desde la interfaz ILead, necesitamos proporcionar un Método concreto. Aquí estamos obligando a la clase Manager a implementar un método WorkOnTask() sin un propósito. Esto está mal. El diseño viola ISP. Corrijamos el diseño.

Dado que tenemos tres roles, 1. Gerente, que solo puede dividir y asignar las tareas, 2. TeamLead que puede dividir y asignar las tareas y también puede trabajar en ellas, 3. El programador que solo puede trabajar en tareas, necesitamos para dividir las responsabilidades segregando la interfaz ILead. Una interfaz que proporciona un contrato para WorkOnTask().

```
1. public interface IProgrammer
2. {
3.     void WorkOnTask();
4. }
```

Una interfaz que proporciona contratos para gestionar las tareas:

```
1. public interface ILead
2. {
3.     void AssignTask();
4.     void CreateSubTask();
5. }
```

Entonces la implementación se convierte en:

---

---

```
1. public class Programmer: IProgrammer
2. {
3.     public void WorkOnTask()
4.     {
5.         //code to implement to work on the Task.
6.     }
7. }
8. public class Manager: ILead
9. {
10.    public void AssignTask()
11.    {
12.        //Code to assign a Task
13.    }
14.    public void CreateSubTask()
15.    {
16.        //Code to create a sub tasks from a task.
17.    }
18. }
```

TeamLead puede administrar tareas y puede trabajar en ellas si es necesario. Luego, la clase

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

TeamLead debería implementar las interfaces IProgrammer e ILead.

```
1. {
2.     public void AssignTask()
3.     {
4.         //Code to assign a Task
5.     }
6.     public void CreateSubTask()
7.     {
8.         //Code to create a sub task from a task.
9.     }
10.    public void WorkOnTask()
11.    {
12.        //code to implement to work on the Task.
13.    }
14. }
```

Aquí separamos responsabilidades/propositos y los distribuimos en múltiples interfaces y también brindamos un buen nivel de abstracción.

### D: Principio de Inversión de Dependencia

El Principio de Inversión de Dependencia (DIP) establece que los módulos/clases de alto nivel no deben depender de los módulos/clases de bajo nivel. Ambos deberían depender de abstracciones. En segundo lugar, las abstracciones no deberían depender de los detalles. Los detalles deben depender de las abstracciones.

Los módulos/clases de alto nivel implementan reglas comerciales o lógica en un sistema (aplicación). Los módulos/clases de bajo nivel se ocupan de operaciones más detalladas; en otras palabras, pueden ocuparse de escribir información en bases de datos o pasar mensajes al sistema operativo o servicios.

Se dice que un módulo/clase de alto nivel que depende de módulos/clases de bajo nivel o alguna otra clase y sabe mucho sobre las otras clases con las que interactúa está estrechamente acoplado. Cuando una clase conoce explícitamente el diseño y la implementación de otra clase, aumenta el riesgo de que los cambios en una clase rompan la otra clase. Por lo tanto, debemos mantener estos módulos/clases de alto y bajo nivel lo más sueltos que podamos. Para hacer eso, necesitamos hacer que ambos dependan de abstracciones en lugar de que se conozcan.

#### Ejemplo:

Supongamos que necesitamos trabajar en un módulo de registro de errores que registra los rastros

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

de la pila de excepciones en un archivo. Sencillo, ¿no? Las siguientes son las clases que proporcionan la funcionalidad para registrar un seguimiento de pila en un archivo.

```
1. public class FileLogger
2. {
3.     public void LogMessage(string aStackTrace)
4.     {
5.         //code to log stack trace into a file.
6.     }
7. }
8. public class ExceptionLogger
9. {
10.    public void LogIntoFile(Exception aException)
11.    {
12.        FileLogger objFileLogger = new FileLogger();
13.        objFileLogger.LogMessage(GetUserReadableMessage(aException
14.    ));
15.    private GetUserReadableMessage(Exception ex)
16.    {
17.        string strMessage = string.Empty;
18.        //code to convert Exception's stack trace and message to u
ser readable format.
19.        ....
20.        ....
21.        return strMessage;
22.    }
23. }
```

Una clase de cliente exporta datos de muchos archivos a una base de datos.

```
1. public class DataExporter
2. {
3.     public void ExportDataFromFile()
4.     {
5.         try {
6.             //code to export data from files to database.
7.         }
8.         catch(Exception ex)
9.         {
10.             new ExceptionLogger().LogIntoFile(ex);
11.         }
12.     }
13. }
```

Enviamos nuestra aplicación al cliente. Pero nuestro cliente quiere almacenar este seguimiento de pila en una base de datos si ocurre una excepción de E/S. Hmm... está bien, no hay problema. También podemos implementar eso. Aquí necesitamos agregar una clase más que

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

proporcione la funcionalidad para registrar el seguimiento de la pila en la base de datos y un método adicional en ExceptionLogger para interactuar con nuestra nueva clase para registrar el seguimiento de la pila.

```
1. public class DbLogger
2. {
3.     public void LogMessage(string aMessage)
4.     {
5.         //Code to write message in database.
6.     }
7. }
8. public class FileLogger
9. {
10.    public void LogMessage(string aStackTrace)
11.    {
12.        //code to log stack trace into a file.
13.    }
14. }
15. public class ExceptionLogger
16. {
17.    public void LogIntoFile(Exception aException)
18.    {
19.        FileLogger objFileLogger = new FileLogger();
20.        objFileLogger.LogMessage(GetUserReadableMessage(aException));
21.    }
22.    public void LogIntoDataBase(Exception aException)
23.    {
24.        DbLogger objDbLogger = new DbLogger();
25.        objDbLogger.LogMessage(GetUserReadableMessage(aException));
26.    }
27.    private string GetUserReadableMessage(Exception ex)
28.    {
29.        string strMessage = string.Empty;
30.        //code to convert Exception's stack trace and message to user readable format.
31.        ....
32.        ....
33.        return strMessage;
34.    }
35. }
36. public class DataExporter
37. {
38.    public void ExportDataFromFile()
39.    {
40.        try {
41.            //code to export data from files to database.
42.        }
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
43.         catch(IOException ex)
44.     {
45.         new ExceptionLogger().LogIntoDataBase(ex);
46.     }
47.     catch(Exception ex)
48.     {
49.         new ExceptionLogger().LogToFile(ex);
50.     }
51. }
52. }
```

Pero siempre que el cliente quiera introducir un nuevo registrador, debemos modificar ExceptionLogger agregando un nuevo método. Si continuamos haciendo esto después de un tiempo, veremos una gran clase ExceptionLogger con un gran conjunto de métodos que brindan la funcionalidad para registrar un mensaje en varios objetivos. ¿Por qué ocurre este problema? Porque ExceptionLogger contacta directamente con las clases de bajo nivel FileLogger y DbLogger para registrar la excepción. Necesitamos modificar el diseño para que esta clase ExceptionLogger pueda acoplarse libremente con esas clases. Para hacer eso, necesitamos introducir una abstracción entre ellos para que ExceptionLogger pueda contactar la abstracción para registrar la excepción en lugar de depender directamente de las clases de bajo nivel.

```
1. public interface ILogger
2. {
3.     void LogMessage(string aString);
4. }
```

Ahora nuestras clases de bajo nivel necesitan implementar esta interfaz.

```
1. public class DbLogger: ILogger
2. {
3.     public void LogMessage(string aMessage)
4.     {
5.         //Code to write message in database.
6.     }
7. }
8. public class FileLogger: ILogger
9. {
10.    public void LogMessage(string aStackTrace)
11.    {
12.        //code to log stack trace into a file.
13.    }
14. }
```

Ahora, pasamos al inicio de la clase de bajo nivel de la clase ExceptionLogger a la clase

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

DataExporter para hacer que ExceptionLogger se acople libremente con las clases de bajo nivel FileLogger y EventLogger. Y al hacerlo, estamos proporcionando a la clase DataExporter que decida qué tipo de registrador debe llamarse en función de la excepción que se produzca.

```
1. public class ExceptionLogger
2. {
3.     private ILogger _logger;
4.     public ExceptionLogger(ILogger aLogger)
5.     {
6.         this._logger = aLogger;
7.     }
8.     public void LogException(Exception aException)
9.     {
10.         string strMessage = GetUserReadableMessage(aException);
11.         this._logger.LogMessage(strMessage);
12.     }
13.     private string GetUserReadableMessage(Exception aException)
14.     {
15.         string strMessage = string.Empty;
16.         //code to convert Exception's stack trace and message to user readable format.
17.         ....
18.         ....
19.         return strMessage;
20.     }
21. }
22. public class DataExporter
23. {
24.     public void ExportDataFromFile()
25.     {
26.         ExceptionLogger _exceptionLogger;
27.         try {
28.             //code to export data from files to database.
29.         }
30.         catch(IOException ex)
31.         {
32.             _exceptionLogger = new ExceptionLogger(new DbLogger());
33.             _exceptionLogger.LogException(ex);
34.         }
35.         catch(Exception ex)
36.         {
37.             _exceptionLogger = new ExceptionLogger(new FileLogger()
38.             );
39.             _exceptionLogger.LogException(ex);
40.         }
41.     }
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

Eliminamos con éxito la dependencia de las clases de bajo nivel. Este ExceptionLogger no depende de las clases FileLogger y EventLogger para registrar el seguimiento de la pila. Ya no necesitamos cambiar el código de ExceptionLogger para ninguna funcionalidad de registro nueva. Necesitamos crear una nueva clase de registro que implemente la interfaz ILogger y debemos agregar otro bloque catch al método ExportDataFromFile de la clase DataExporter.

```
1. public class EventLogger: ILogger
2. {
3.     public void LogMessage(string aMessage)
4.     {
5.         //Code to write message in system's event viewer.
6.     }
7. }
```

Y necesitamos agregar una condición en la clase DataExporter como se muestra a continuación:

```
1. public class DataExporter
2. {
3.     public void ExportDataFromFile()
4.     {
5.         ExceptionLogger _exceptionLogger;
6.         try {
7.             //code to export data from files to database.
8.         }
9.         catch(IOException ex)
10.        {
11.            _exceptionLogger = new ExceptionLogger(new DbLogger());
12.
13.            _exceptionLogger.LogException(ex);
14.        }
15.        catch(SqlException ex)
16.        {
17.            _exceptionLogger = new ExceptionLogger(new EventLogger(
18.                ));
19.            _exceptionLogger.LogException(ex);
20.        }
21.        catch(Exception ex)
22.        {
23.            _exceptionLogger = new ExceptionLogger(new FileLogger(
24.                ));
25.            _exceptionLogger.LogException(ex);
26.        }
27.    }
28. }
```

**CONCLUSIONES:**

- Se ha logrado dar cada una de los patrones de diseños y lo que son los patrones de diseño de manera eficiente.
- Se mostró diversos ejemplos sobre los distintos patrones de diseño.
- Se explicó con claridad a lo que hace referencia en SOLID.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

### ANEXOS

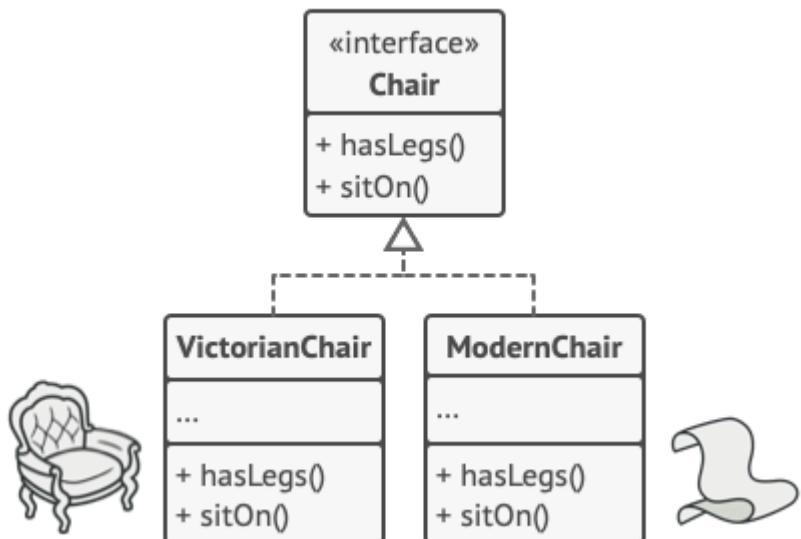


IMG 1.1

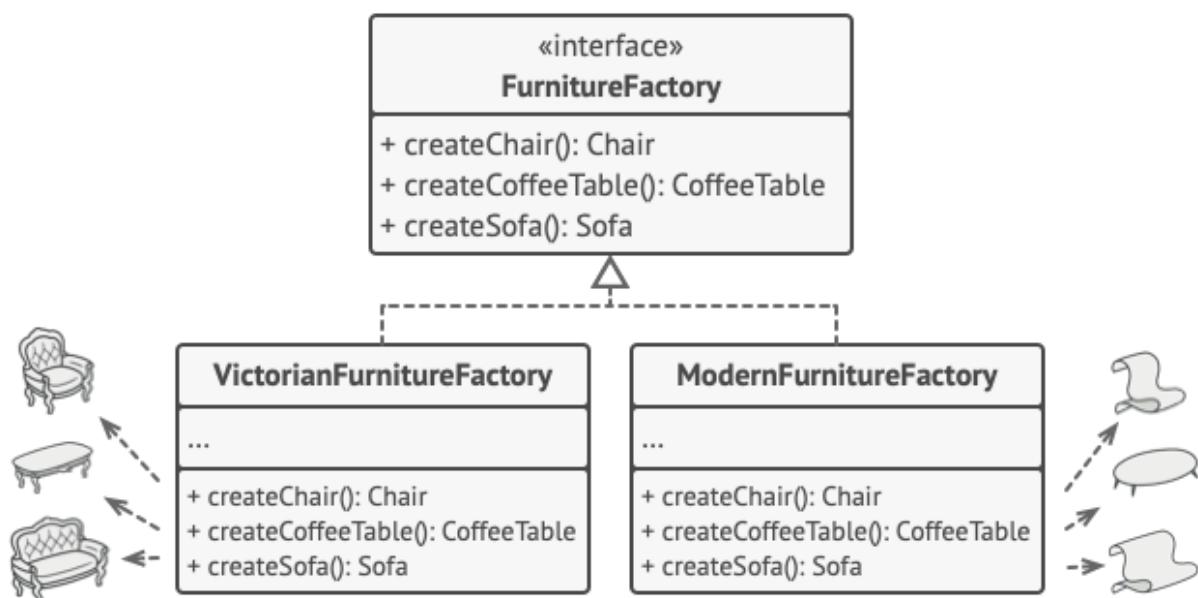


IMG 1.2

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID



IMG 1.3



IMG 1.4

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID



IMG 1.5

### Ejemplo Singleton

The screenshot shows the Microsoft Visual Studio IDE interface with the following details:

- Project Explorer:** Shows a solution named "patronesDiseno" containing a "patronesDiseno.singleton" file.
- Code Editor:** Displays the "singleton.cs" file with the following C# code:

```
namespace patronesDiseno
{
    internal class singleton
    {
        private static singleton instance = null;
        public string mensaje = "";

        protected singleton()
        {
            mensaje = "hola mundo";
        }

        public static singleton Instance
        {
            get
            {
                if (instance == null)
                    instance = new singleton();
                return instance;
            }
        }
    }
}
```
- Solution Explorer:** Shows the project structure with files "program.cs" and "singleton.cs".
- Status Bar:** Shows the current line (Línea: 6), character (Carácter: 36), and encoding (SPC | CRLF).
- Taskbar:** Shows the Windows taskbar with icons for File Explorer, Task View, Start, and Visual Studio.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

### Ejemplo Prototype

The screenshot shows the Microsoft Visual Studio IDE interface. The top menu bar includes Archivo, Editar, Ver, Git, Proyecto, Compilar, Depurar, Prueba, Analizar, Herramientas, Extensiones, Ventana, Ayuda, and Buscar (Ctrl+Q). The toolbar below has icons for Undo, Redo, Cut, Copy, Paste, Find, Replace, and others. The main editor window displays the following C# code:

```
namespace patronesDiseno
{
    //clase prototype superficial
    public class Animal : ICloneable
    {
        3 referencias
        public int Patas { get; set; }
        1 referencia
        public string Nombre { get; set; }

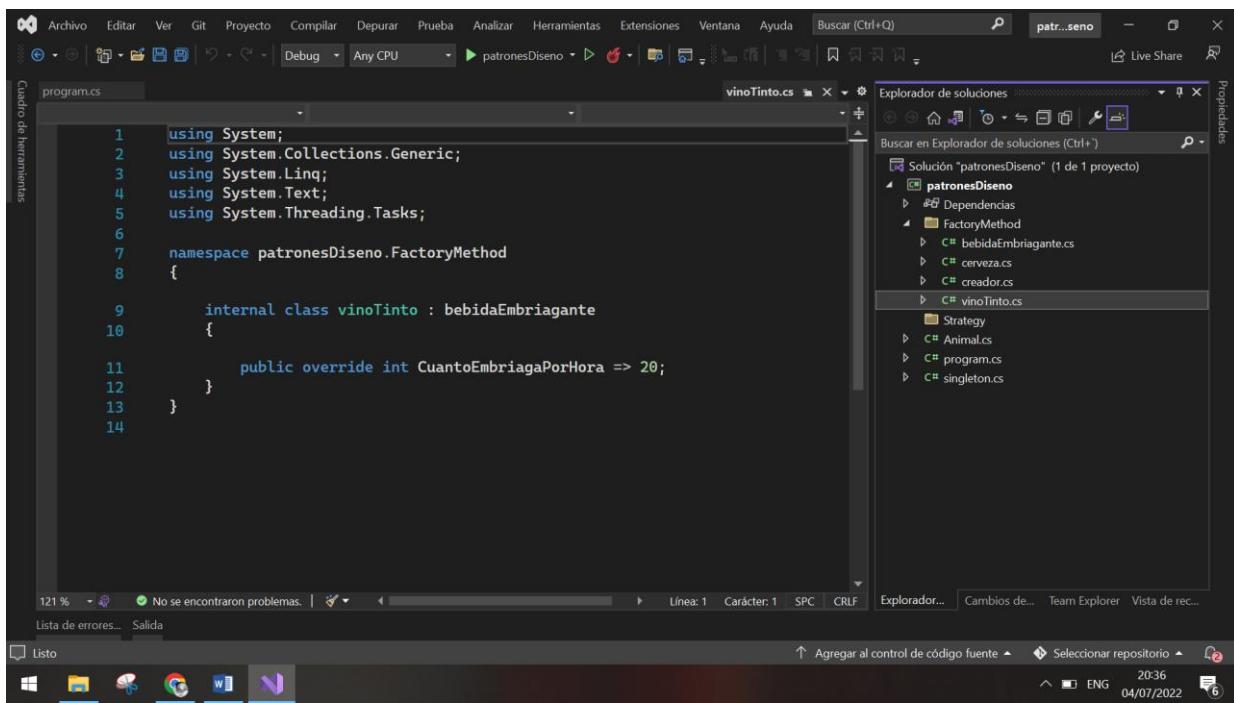
        1 referencia
        public object Clone()
        {
            return this.MemberwiseClone();
        }
    }
}
```

The code editor has syntax highlighting for C# and shows line numbers 1 through 16. To the right of the editor is the Solution Explorer, which lists the project "patronesDiseno" with files "Animal.cs", "program.cs", and "singleton.cs". Below the editor is the Task List, Error List, and Output windows. The status bar at the bottom shows the date and time as 04/07/2022 19:39.

### Ejemplo Factory Method

This screenshot of Microsoft Visual Studio displays the same C# code as the previous one, representing the Prototype design pattern. The code defines a class "Animal" that implements the "ICloneable" interface, featuring a "Clone" method that returns a copy of itself using "MemberwiseClone". The Solution Explorer on the right shows the project "patronesDiseno" with files "Animal.cs", "program.cs", and "singleton.cs". The interface and file structure are identical to the previous screenshot, indicating no changes have been made to the code.

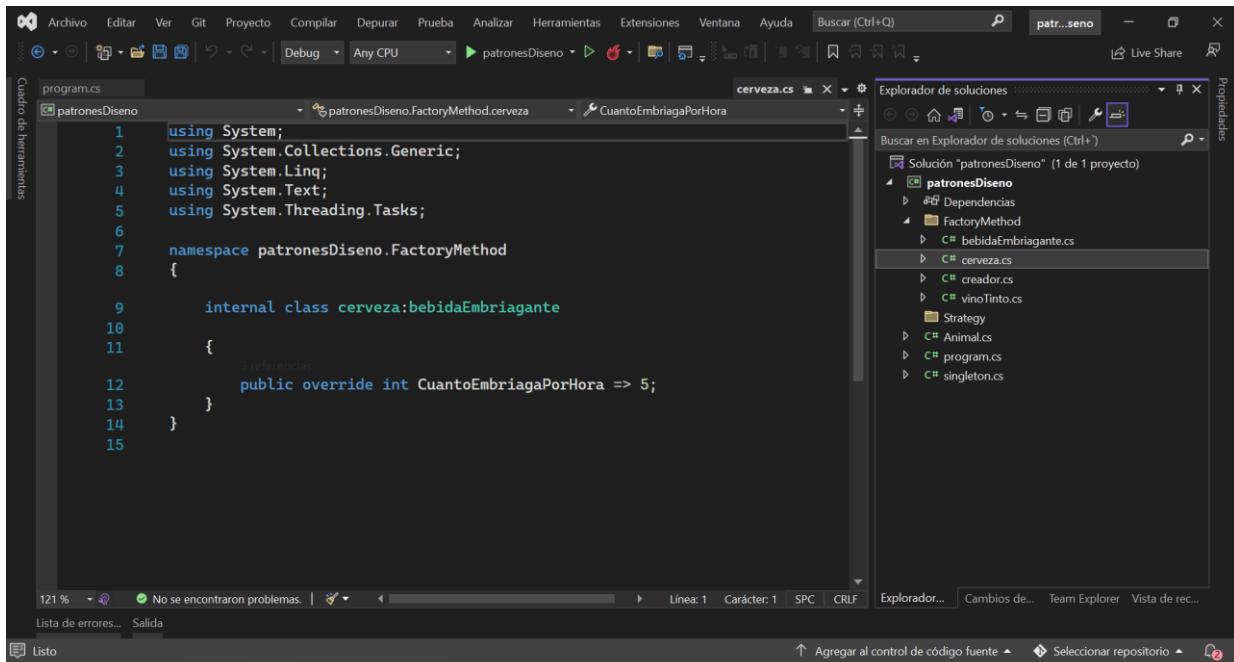
## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID



The screenshot shows the Visual Studio IDE interface with the following details:

- Menu Bar:** Archivo, Editar, Ver, Git, Proyecto, Compilar, Depurar, Prueba, Analizar, Herramientas, Extensiones, Ventana, Ayuda.
- Toolbar:** Buscar (Ctrl+Q), Live Share.
- Solution Explorer:** Muestra la solución "patronesDiseno" (1 de 1 proyecto) que contiene el proyecto "patronesDiseno". El proyecto "FactoryMethod" incluye los archivos: bebidaEmbriagante.cs, cerveza.cs, creador.cs, vinoTinto.cs, y Strategy.
- Code Editor:** Abierto el archivo "vinoTinto.cs" que contiene el siguiente código C#:using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace patronesDiseno.FactoryMethod
{
 internal class vinoTinto : bebidaEmbriagante
 {
 public override int CuantoEmbriagaPorHora => 20;
 }
}
- Status Bar:** 121%, No se encontraron problemas., Línea: 1, Carácter: 1, SPC, CRLF.
- Task List:** Lista de errores..., Salida.
- System Tray:** Listo, Agregar al control de código fuente, Seleccionar repositorio, 20:36, 04/07/2022, 6.

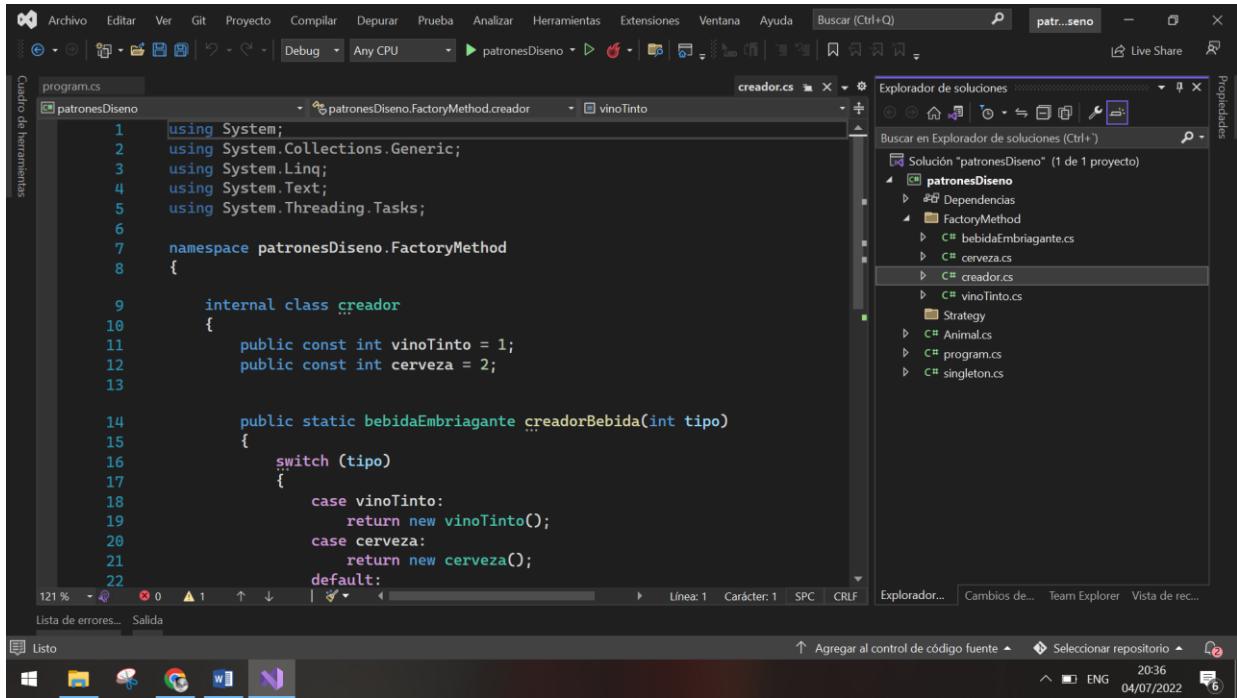
  


Este es otro screenshot del Visual Studio IDE con las siguientes características:

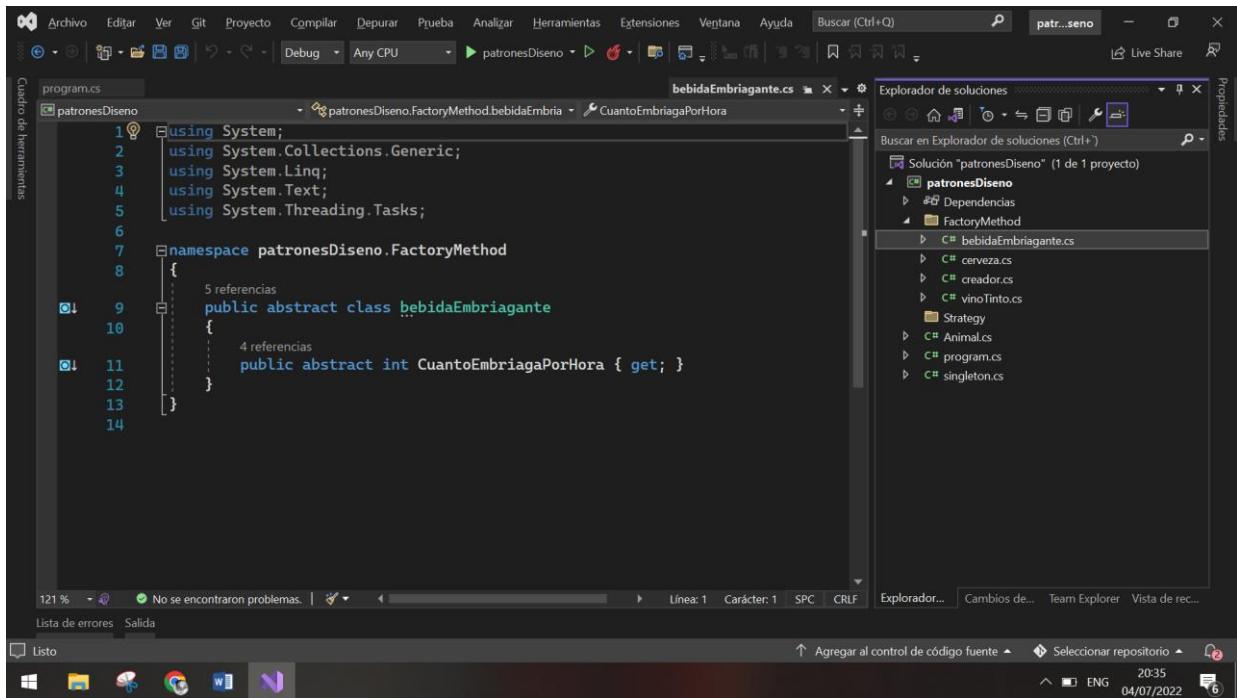
- Code Editor:** Abierto el archivo "cerveza.cs" que contiene el siguiente código C#:using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace patronesDiseno.FactoryMethod
{
 internal class cerveza:bebidaEmbriagante
 {
 public override int CuantoEmbriagaPorHora => 5;
 }
}
- Solution Explorer:** Muestra la solución "patronesDiseno" (1 de 1 proyecto) que contiene el proyecto "patronesDiseno". El proyecto "FactoryMethod" incluye los archivos: bebidaEmbriagante.cs, cerveza.cs, creador.cs, vinoTinto.cs, y Strategy.
- Status Bar:** 121%, No se encontraron problemas., Línea: 1, Carácter: 1, SPC, CRLF.
- Task List:** Lista de errores..., Salida.
- System Tray:** Listo, Agregar al control de código fuente, Seleccionar repositorio, 20:36, 04/07/2022, 6.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID



The screenshot shows the Visual Studio IDE interface with the following details:

- Toolbar:** Archivo, Editar, Ver, Git, Proyecto, Compilar, Depurar, Prueba, Analizar, Herramientas, Extensiones, Ventana, Ayuda.
- Search Bar:** Buscar (Ctrl+Q)
- Solution Explorer:** Muestra la solución "patronesDiseno" con un solo proyecto "patronesDiseno". El proyecto contiene los siguientes archivos:
  - Dependencias
  - FactoryMethod
    - bebidaEmbriagante.cs
    - cerveza.cs
    - creador.cs
    - vinoTinto.cs
  - Strategy
    - Animal.cs
    - program.cs
  - singleton.cs
- Code Editor:** Abierto el archivo "creador.cs" que implementa el Factory Method pattern para crear bebidas embriagantes.
- Status Bar:** Muestra el porcentaje de código escrito (121%), la línea y carácter actual (Línea: 1, Carácter: 1), y la hora (20:36).
  


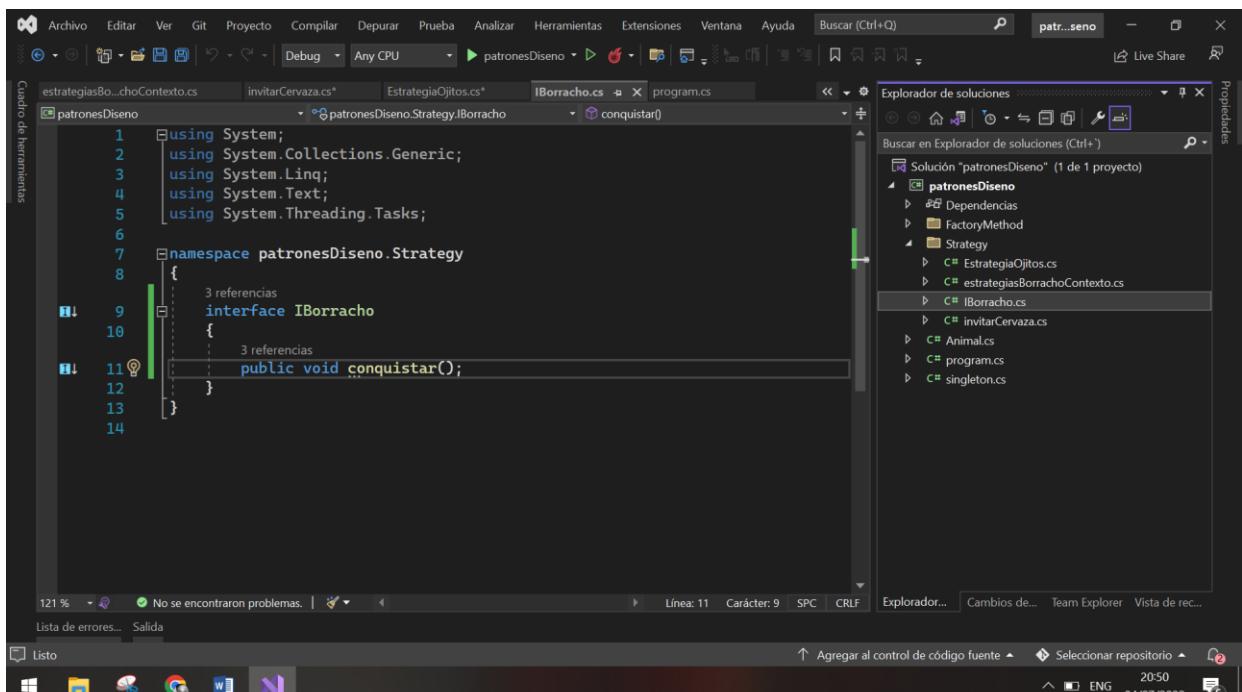
Este segundo screenshot muestra la misma configuración de Visual Studio, pero con el archivo "bebidaEmbriagante.cs" abierto en el editor. El código define una clase abstracta "bebidaEmbriagante" con un método abstracto "CuantoEmбриagaPorHora".

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace patronesDiseno.FactoryMethod
8 {
9     public abstract class bebidaEmbriagante
10    {
11        public abstract int CuantoEmбриagaPorHora { get; }
12    }
13}
14
```

El Explorador de soluciones muestra la misma estructura de proyecto y contenido que en la primera captura de pantalla.

### Ejemplo Strategy

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

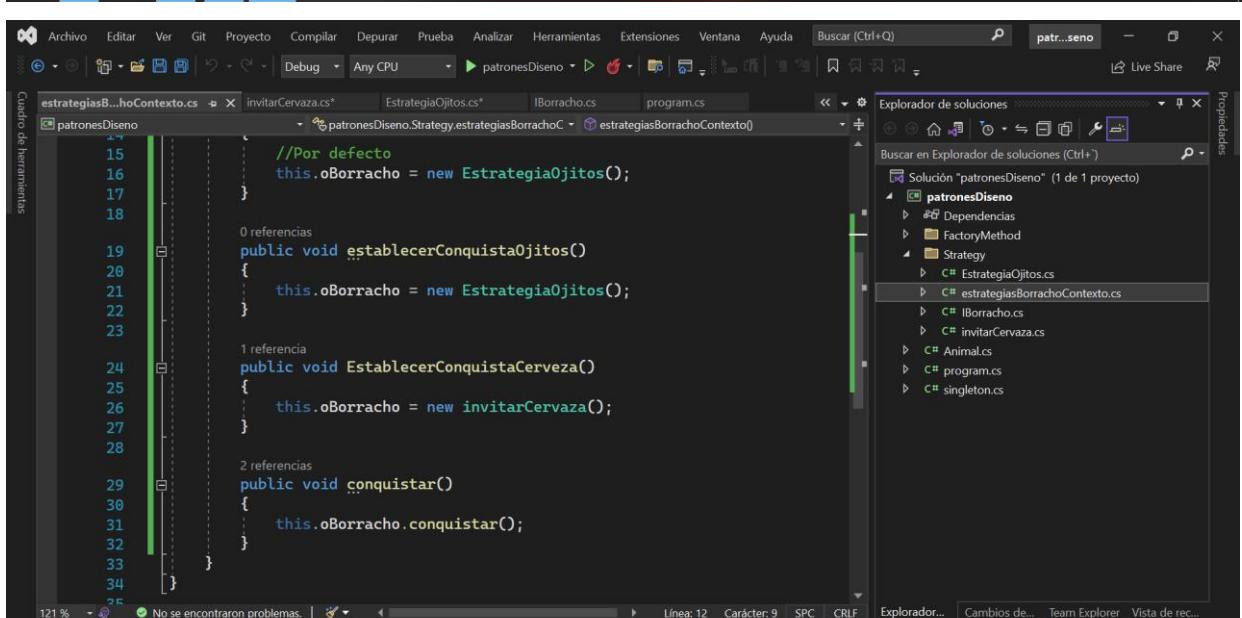


The screenshot shows the Visual Studio IDE interface with the following details:

- Menu Bar:** Archivo, Editar, Ver, Git, Proyecto, Compilar, Depurar, Prueba, Analizar, Herramientas, Extensiones, Ventana, Ayuda.
- Toolbar:** Buscar (Ctrl+Q), Live Share.
- Solution Explorer:** Muestra la solución "patronesDiseno" con un solo proyecto "patronesDiseno". El proyecto contiene los siguientes archivos:
  - Dependencias
  - FactoryMethod
  - Strategy
    - EstrategiaOjitos.cs
    - estrategiasBorrachoContexto.cs
    - IBorracho.cs
    - invitarCerveza.cs
    - Animal.cs
    - program.cs
    - singleton.cs
- Code Editor:** Abre el archivo "IBorracho.cs" que contiene el siguiente código:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace patronesDiseno.Strategy
8  {
9      interface IBorracho
10     {
11         public void conquistar();
12     }
13 }
```

El editor muestra una advertencia de tipo en la línea 11: "3 referencias".



Este es otro captura de pantalla similar a la anterior, pero muestra la ejecución de la función "conquistar()". La barra de status indica "Línea: 11, Carácter: 9".

El código en el editor es:

```
14
15     //Por defecto
16     this.oBorracho = new EstrategiaOjitos();
17 }
18
19     0 referencias
20     public void establecerConquistaOjitos()
21     {
22         this.oBorracho = new EstrategiaOjitos();
23     }
24
25     1 referencia
26     public void EstablecerConquistaCerveza()
27     {
28         this.oBorracho = new invitarCerveza();
29     }
30
31     2 referencias
32     public void conquistar()
33     {
34         this.oBorracho.conquistar();
35     }
36 }
```

La barra de status indica "Línea: 12, Carácter: 9".



Este es otro captura de pantalla similar a las anteriores, pero muestra la ejecución de la función "conquistar()". La barra de status indica "Línea: 12, Carácter: 9".

El código en el editor es:

```
17
18     //Por defecto
19     this.oBorracho = new EstrategiaOjitos();
20 }
21
22     0 referencias
23     public void establecerConquistaOjitos()
24     {
25         this.oBorracho = new EstrategiaOjitos();
26     }
27
28     1 referencia
29     public void EstablecerConquistaCerveza()
30     {
31         this.oBorracho = new invitarCerveza();
32     }
33
34     2 referencias
35     public void conquistar()
36     {
37         this.oBorracho.conquistar();
38     }
39 }
```

La barra de status indica "Línea: 13, Carácter: 9".

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

The screenshot shows the Visual Studio IDE interface with the following details:

- Project Explorer:** Shows a solution named "patronesDiseno" containing one project "patronesDiseno". The project includes files: Dependencias, FactoryMethod, Strategy, EstrategiaOjitos.cs, estrategiasBorrachoContexto.cs, IBorracho.cs, invitarCervaza.cs, Animal.cs, program.cs, and singleton.cs.
- Code Editor:** Displays the file "estrategiasBorrachoContexto.cs" with the following code:namespace patronesDiseno.Strategy
{
 internal class estrategiasBorrachoContexto
 {
 private IBorracho oBorracho;

 public estrategiasBorrachoContexto()
 {
 //Por defecto
 this.oBorracho = new EstrategiaOjitos();
 }

 public void establecerConquistaOjitos()
 {
 this.oBorracho = new EstrategiaOjitos();
 }

 public void EstablecerConquistaCerveza()
 {
 this.oBorracho = new invitarCervaza();
 }
 }
}
- Status Bar:** Shows "121 %", "No se encontraron problemas.", "Línea: 12", "Carácter: 9", "SPC", "CRLF".
- Taskbar:** Shows icons for File, Edit, View, Project, Build, Debug, Run, Analyze, Tools, Window, Help, and Visual Studio.
- System Tray:** Shows icons for Network, Battery, and Date/Time (04/07/2022, 20:51).

The second screenshot shows the Visual Studio IDE interface with the following details:

- Project Explorer:** Shows the same solution and project structure as the first screenshot.
- Code Editor:** Displays the file "EstrategiaOjitos.cs" with the following code:using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace patronesDiseno.Strategy
{
 internal class EstrategiaOjitos : IBorracho
 {
 public void conquistar()
 {
 Console.WriteLine("Le hago ojitos a la muchacha");
 }
 }
}
- Status Bar:** Shows "121 %", "No se encontraron problemas.", "Línea: 11", "Carácter: 33", "SPC", "CRLF".
- Taskbar:** Shows icons for File, Edit, View, Project, Build, Debug, Run, Analyze, Tools, Window, Help, and Visual Studio.
- System Tray:** Shows icons for Network, Battery, and Date/Time (04/07/2022, 20:51).

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

The screenshot shows the Microsoft Visual Studio IDE interface. The top menu bar includes Archivo, Editar, Ver, Git, Proyecto, Compilar, Depurar, Prueba, Analizar, Herramientas, Extensiones, Ventana, Ayuda, and Buscar (Ctrl+Q). The toolbar has icons for file operations like Open, Save, and Print. The solution explorer on the right shows a project named "patronesDiseno" with files like EstrategiaOjitos.cs, FactoryMethod.cs, Strategy.cs, and Animal.cs. The code editor displays a C# class named "invitarCervaza" that implements the "IBorracho" interface. The class contains a single method "conquistar()" which prints a message to the console.

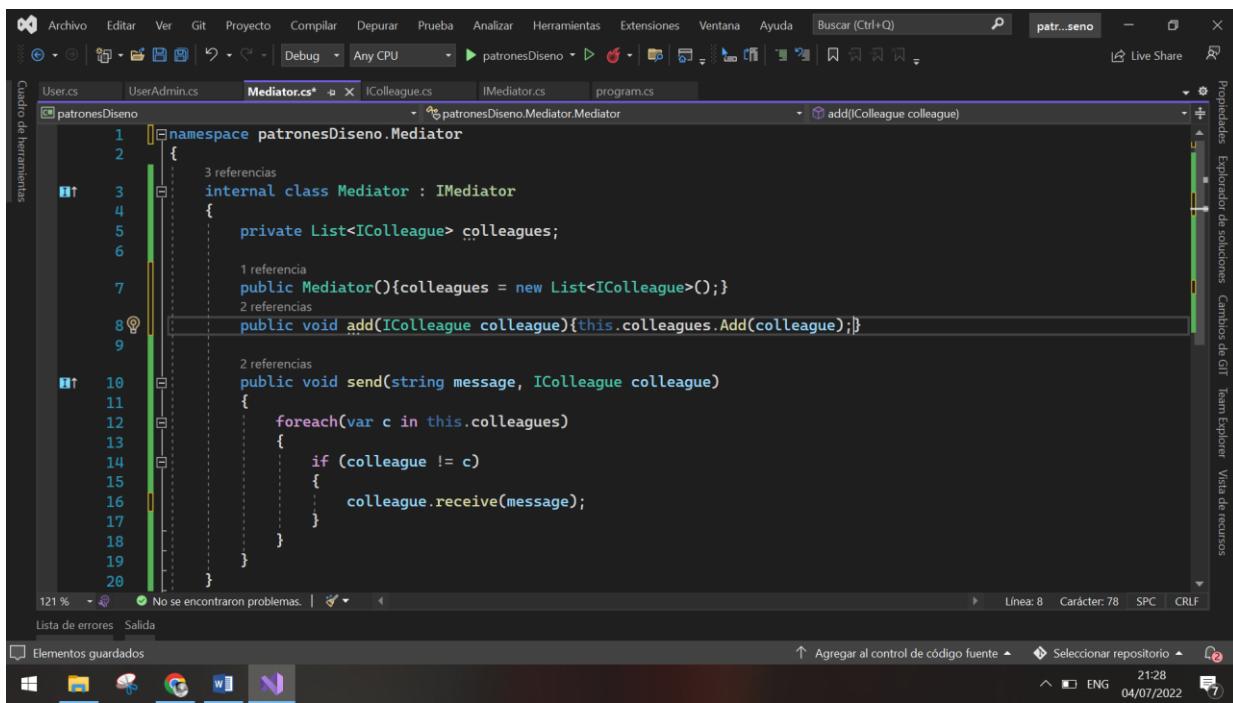
```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace patronesDiseno.Strategy
8 {
9     internal class invitarCervaza : IBorracho
10    {
11        public void conquistar()
12        {
13            Console.WriteLine("Le invito una cerveza");
14        }
15    }
16}
17
```

## Ejemplo Mediator

The screenshot shows the Microsoft Visual Studio IDE interface. The top menu bar includes Archivo, Editar, Ver, Git, Proyecto, Compilar, Depurar, Prueba, Analizar, Herramientas, Extensiones, Ventana, Ayuda, and Buscar (Ctrl+Q). The toolbar has icons for file operations like Open, Save, and Print. The solution explorer on the right shows a project named "patronesDiseno" with files like User.cs, UserAdmin.cs, Mediator.cs, IColleague.cs, IMediator.cs, and program.cs. The code editor displays a C# abstract class "IColleague" with methods "communicate(string message)" and "receive(string message)". It also includes a constructor that takes an "IMediator" parameter and sets it to the "mediator" field.

```
1 namespace patronesDiseno.Mediator
2 {
3     public abstract class IColleague
4     {
5         private IMediator mediator;
6
7         public IColleague(IMediator mediator)
8         {
9             this.mediator = mediator;
10        }
11
12        public void communicate(string message)
13        {
14            this.mediator.send(message, this);
15        }
16
17        public abstract void receive(string message);
18    }
19}
20
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

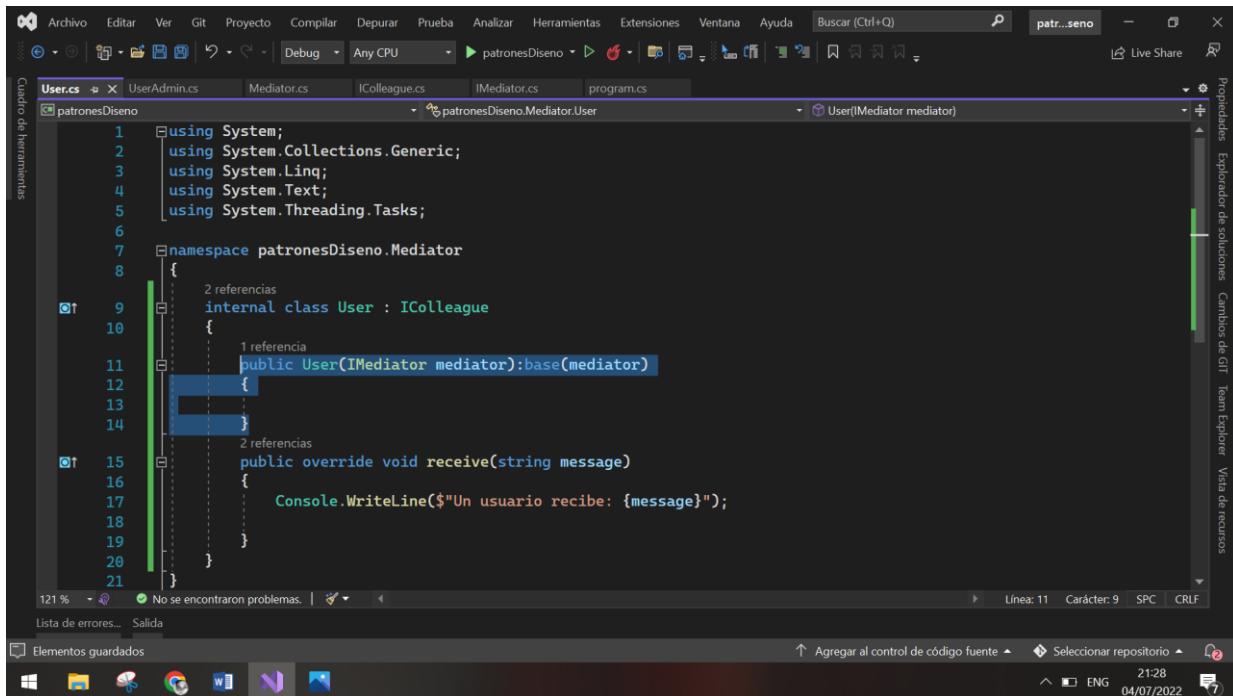


The screenshot shows the Microsoft Visual Studio IDE interface with the following details:

- Project:** patronesDiseno
- Open File:** Mediator.cs
- Code Content:**

```
1  namespace patronesDiseno.Mediator
2  {
3      3 referencias
4      internal class Mediator : IMediator
5      {
6          private List<IColleague> colleagues;
7
8          1 referencia
9          public Mediator(){colleagues = new List<IColleague>();}
10         2 referencias
11         public void add(IColleague colleague){this.colleagues.Add(colleague);}
12
13         2 referencias
14         public void send(string message, IColleague colleague)
15         {
16             foreach(var c in this.colleagues)
17             {
18                 if (colleague != c)
19                 {
20                     colleague.receive(message);
21                 }
22             }
23         }
24     }
25 }
```

- Toolbars and Menus:** Archivo, Editar, Ver, Git, Proyecto, Compilar, Depurar, Prueba, Analizar, Herramientas, Extensiones, Ventana, Ayuda, Buscar (Ctrl+Q), Live Share.
- Status Bar:** Línea: 8, Carácter: 78, SPC, CRLF.
- Bottom Bar:** Agregar al control de código fuente, Seleccionar repositorio, 21:28, ENG, 04/07/2022.

The screenshot shows the Microsoft Visual Studio IDE interface with the following details:

- Project:** patronesDiseno
- Open File:** User.cs
- Code Content:**

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace patronesDiseno.Mediator
8  {
9      2 referencias
10     internal class User : IColleague
11     {
12         1 referencia
13         public User(IMediator mediator):base(mediator)
14         {
15
16         }
17
18         2 referencias
19         public override void receive(string message)
20         {
21             Console.WriteLine($"Un usuario recibe: {message}");
22         }
23     }
24 }
```

- Toolbars and Menus:** Archivo, Editar, Ver, Git, Proyecto, Compilar, Depurar, Prueba, Analizar, Herramientas, Extensiones, Ventana, Ayuda, Buscar (Ctrl+Q), Live Share.
- Status Bar:** Línea: 11, Carácter: 9, SPC, CRLF.
- Bottom Bar:** Agregar al control de código fuente, Seleccionar repositorio, 21:28, ENG, 04/07/2022.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

The screenshot shows the Microsoft Visual Studio IDE interface with the following details:

- Project:** patronesDiseno
- Open Files:** UserAdmin.cs, Mediator.cs, IColleague.cs, IMediator.cs, program.cs
- UserAdmin.cs Content:**

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace patronesDiseno.Mediator
8 {
9     public class UserAdmin : IColleague
10    {
11        public UserAdmin(IMediator mediator) : base(mediator)
12        {
13        }
14
15        public override void receive(string message)
16        {
17            Console.WriteLine($"Un administrador recibe: {message}");
18        }
19    }
20
21 }
```

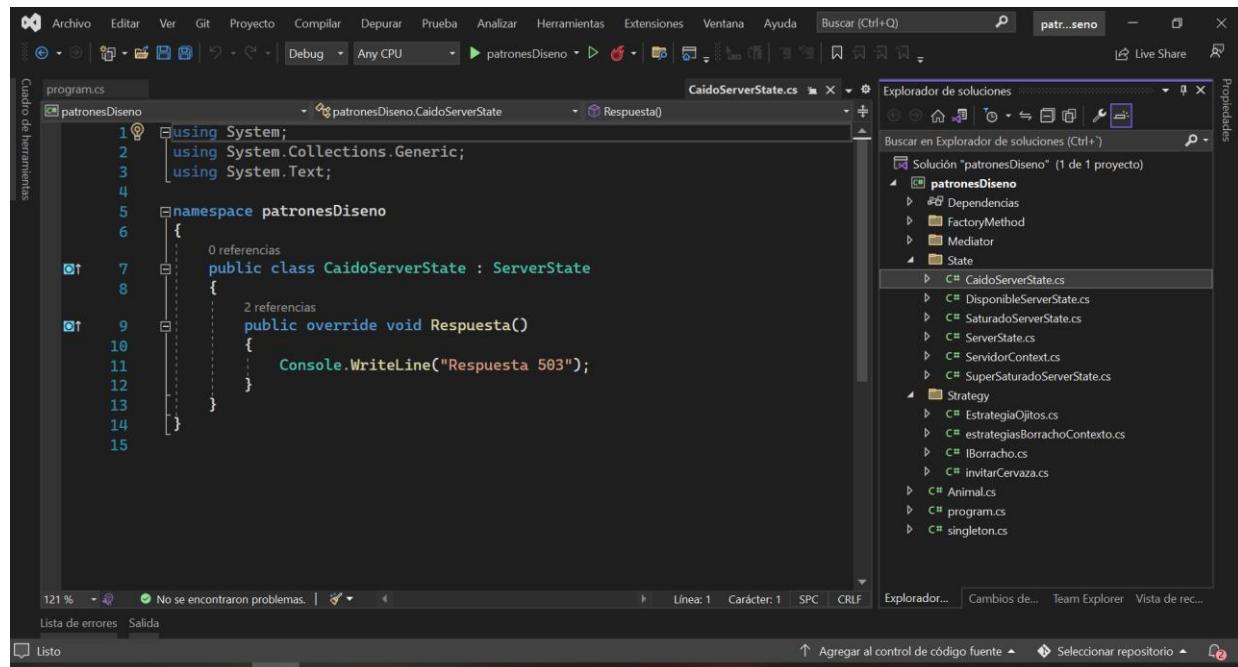
- Mediator.cs Content:**

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace patronesDiseno.Mediator
8 {
9     internal class Mediator : IMediator
10    {
11        private List<IColleague> colleagues;
12
13        public Mediator()
14        {
15            colleagues = new List<IColleague>();
16        }
17
18        public void add(IColleague colleague){this.colleagues.Add(colleague);}
19
20        public void send(string message, IColleague colleague)
21        {
22            foreach(var c in this.colleagues)
23            {
24                if (colleague != c)
25                {
26                    colleague.receive(message);
27                }
28            }
29        }
30    }
31 }
```

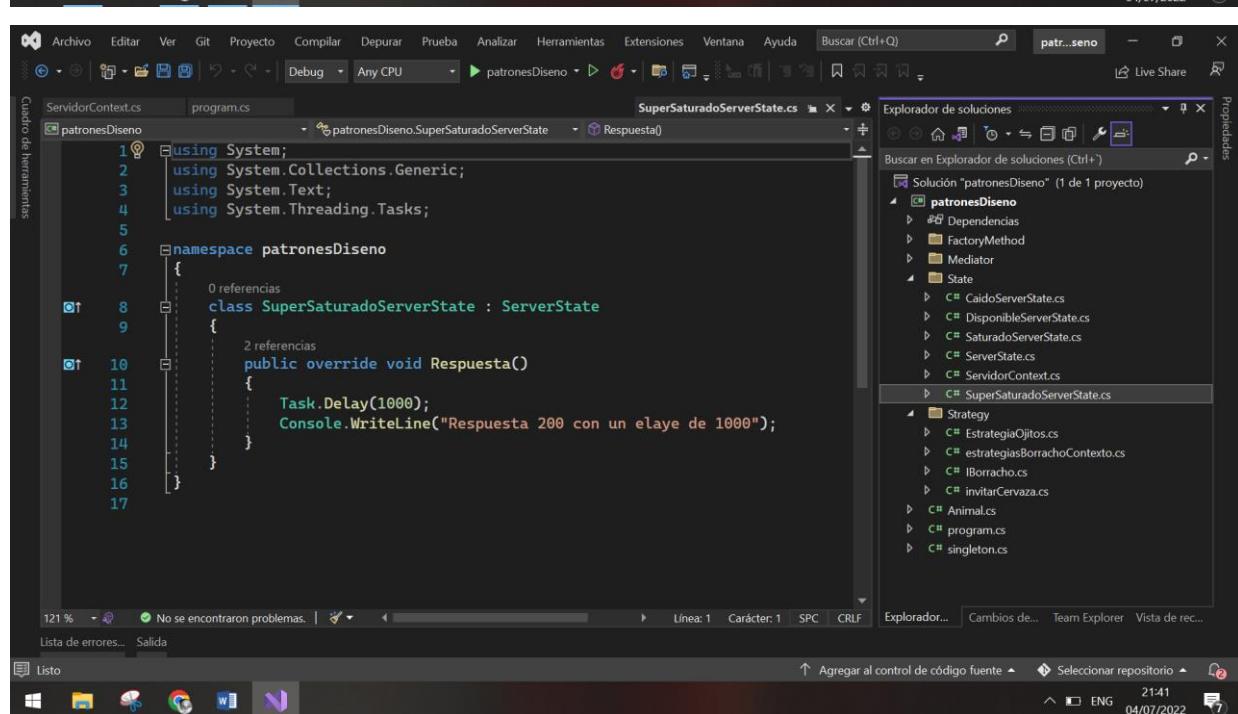
- Toolbars and Status Bar:** The status bar shows "Línea: 12" and "Carácter: 10". The bottom right corner displays "21:28 04/07/2022".

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

### Ejemplo State



```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace patronesDiseno
6 {
7     public class CaidoServerState : ServerState
8     {
9         public override void Respuesta()
10        {
11            Console.WriteLine("Respuesta 503");
12        }
13    }
14}
15
```



```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4 using System.Threading.Tasks;
5
6 namespace patronesDiseno
7 {
8     class SuperSaturadoServerState : ServerState
9     {
10        public override void Respuesta()
11        {
12            Task.Delay(1000);
13            Console.WriteLine("Respuesta 200 con un elaye de 1000");
14        }
15    }
16}
17
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

The screenshot shows the Microsoft Visual Studio IDE interface. The top menu bar includes Archivo, Editar, Ver, Git, Proyecto, Compilar, Depurar, Prueba, Analizar, Herramientas, Extensiones, Ventana, Ayuda, and Buscar (Ctrl+Q). The toolbar contains icons for file operations like New, Open, Save, Print, and others. The status bar at the bottom shows 121%, Linea: 14, Carácter: 31, SPC, CRLF, and a date/time stamp of 04/07/2022 21:41.

The main code editor window displays the file `ServidorContext.cs` with the following code:

```
4
5     namespace patronesDiseno
6     {
7         2 referencias
8         public class ServidorContext
9         {
10             private ServerState state;
11
12             1 referencia
13             public ServerState State
14             {
15                 get{return state;}
16                 set{state = value;}
17             }
18             2 referencias
19             public void AtenderSolicitud()
20             {
21                 state.Respuesta();
22             }
23         }
24     }
```

The Solution Explorer on the right shows the project structure for "patronesDiseno":

- Solución "patronesDiseno" (1 de 1 proyecto)
  - patronesDiseno
    - Dependencias
    - FactoryMethod
    - Mediator
    - State
      - C# CaidoServerState.cs
      - C# DisponibleServerState.cs
      - C# SaturadoServerState.cs
      - C# ServerState.cs
      - C# ServidorContext.cs
      - C# SuperSaturadoServerState.cs
    - Strategy
      - C# EstrategiaOjitos.cs
      - C# estrategiasBorrachoContexto.cs
      - C# IBorracho.cs
      - C# invitarCerveza.cs
    - C# Animal.cs
    - C# program.cs
    - C# singleton.cs

The bottom status bar shows 121%, Linea: 14, Carácter: 31, SPC, CRLF, and a date/time stamp of 04/07/2022 21:41.

The second screenshot shows the Microsoft Visual Studio IDE interface. The top menu bar includes Archivo, Editar, Ver, Git, Proyecto, Compilar, Depurar, Prueba, Analizar, Herramientas, Extensiones, Ventana, Ayuda, and Buscar (Ctrl+Q). The toolbar contains icons for file operations like New, Open, Save, Print, and others. The status bar at the bottom shows 121%, Linea: 1, Carácter: 1, SPC, CRLF, and a date/time stamp of 04/07/2022 21:40.

The main code editor window displays the file `program.cs` with the following code:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace patronesDiseno
6  {
7      public abstract class ServerState
8      {
9          public abstract void Respuesta();
10     }
11 }
```

The Solution Explorer on the right shows the project structure for "patronesDiseno":

- Solución "patronesDiseno" (1 de 1 proyecto)
  - patronesDiseno
    - Dependencias
    - FactoryMethod
    - Mediator
    - State
      - C# CaidoServerState.cs
      - C# DisponibleServerState.cs
      - C# SaturadoServerState.cs
      - C# ServerState.cs
      - C# ServidorContext.cs
      - C# SuperSaturadoServerState.cs
    - Strategy
      - C# EstrategiaOjitos.cs
      - C# estrategiasBorrachoContexto.cs
      - C# IBorracho.cs
      - C# invitarCerveza.cs
    - C# Animal.cs
    - C# program.cs
    - C# singleton.cs

The bottom status bar shows 121%, Linea: 1, Carácter: 1, SPC, CRLF, and a date/time stamp of 04/07/2022 21:40.

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

Screenshot of Visual Studio showing the SaturadoServerState.cs code. The code defines a class SaturadoServerState that inherits from ServerState. It overrides the Respuesta() method to delay the response by 500ms and print a message to the console.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4 using System.Threading.Tasks;
5
6 namespace patronesDiseno
7 {
8     class SaturadoServerState : ServerState
9     {
10         public override void Respuesta()
11         {
12             Task.Delay(500);
13             Console.WriteLine("Respuesta 200 con un delay de 500");
14         }
15     }
16 }
```

The Solution Explorer shows the project structure with files like CaidoServerState.cs, DisponibleServerState.cs, SaturadoServerState.cs, and others.

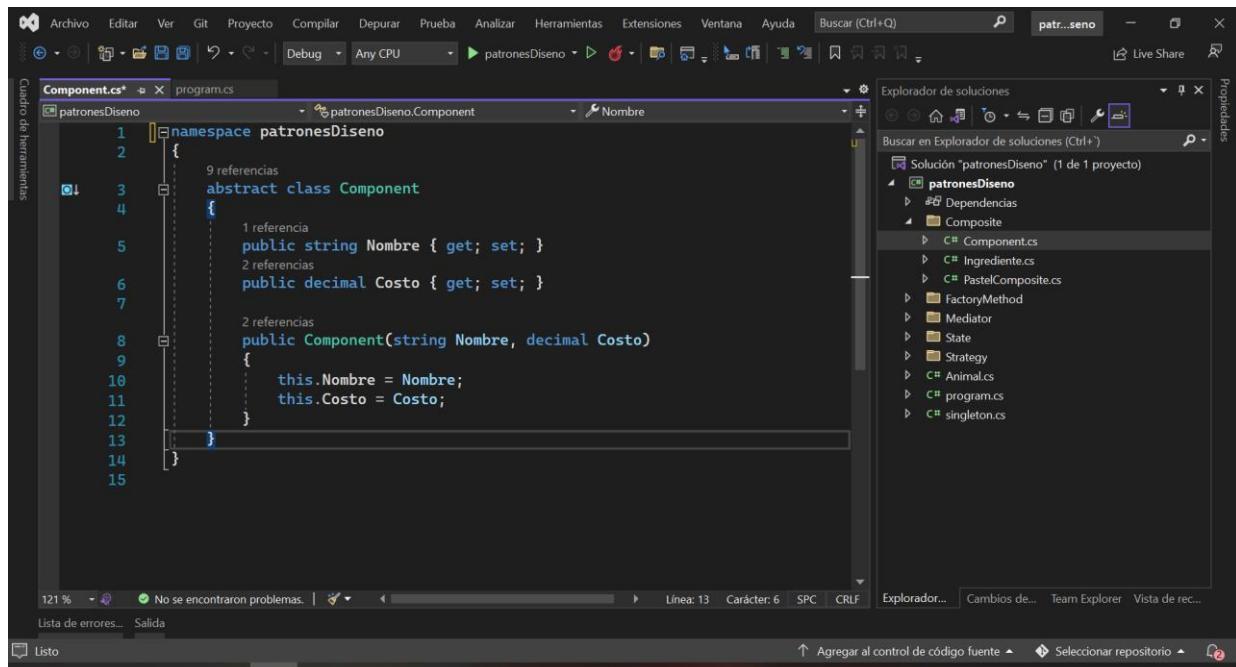
Screenshot of Visual Studio showing the DisponibleServerState.cs code. The code defines a class DisponibleServerState that inherits from ServerState. It overrides the Respuesta() method to print a message to the console.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace patronesDiseno
6 {
7     class DisponibleServerState : ServerState
8     {
9         public override void Respuesta()
10        {
11            Console.WriteLine("Respuesta 200");
12        }
13    }
14 }
```

The Solution Explorer shows the project structure with files like CaidoServerState.cs, DisponibleServerState.cs, SaturadoServerState.cs, and others.

# INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

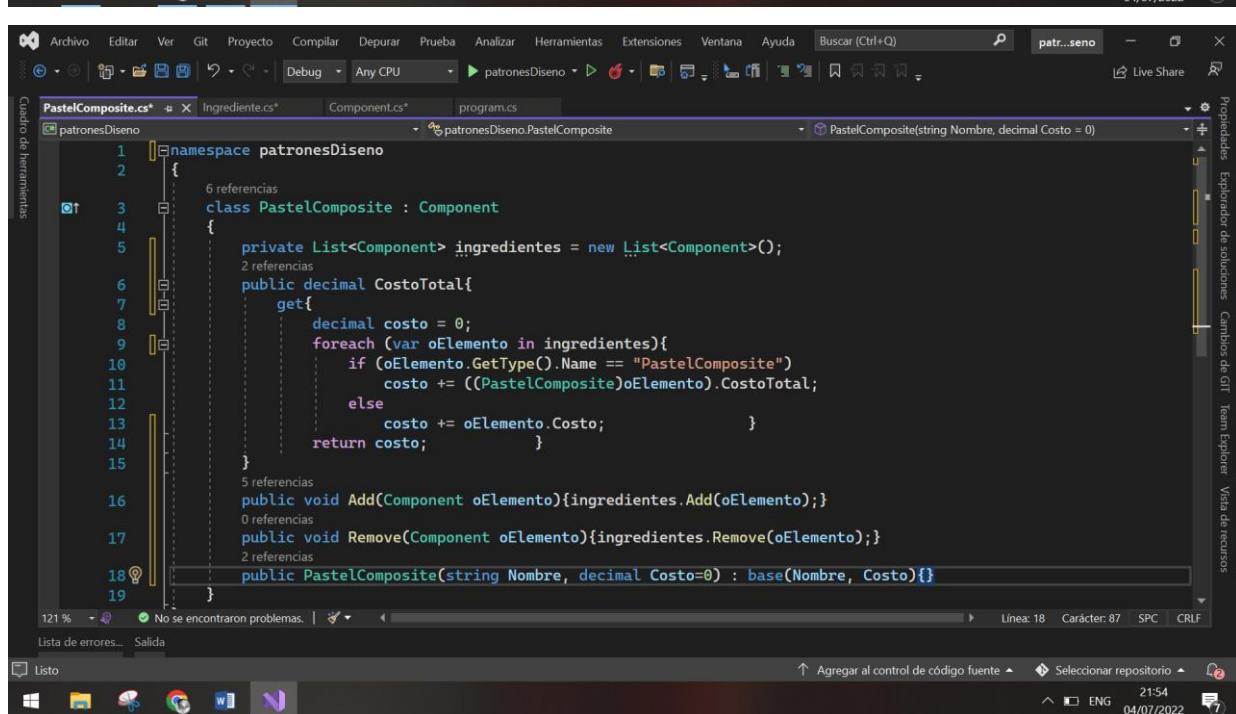
## Ejemplo Composite



Screenshot of Visual Studio showing the Component.cs file in the Component class. The code defines an abstract class Component with properties Nombre and Costo, and a constructor that initializes them. The Component class has 9 references and is part of the patronesDiseno.Component namespace.

```
namespace patronesDiseno.Component
{
    abstract class Component
    {
        public string Nombre { get; set; }
        public decimal Costo { get; set; }

        public Component(string Nombre, decimal Costo)
        {
            this.Nombre = Nombre;
            this.Costo = Costo;
        }
    }
}
```



Screenshot of Visual Studio showing the PastelComposite.cs file in the PastelComposite class. The code defines a class PastelComposite that inherits from Component. It contains a private list of components (ingredientes) and a public property CostoTotal that calculates the total cost of all ingredients. The PastelComposite class has 6 references and is part of the patronesDiseno.PastelComposite namespace.

```
namespace patronesDiseno.PastelComposite
{
    class PastelComposite : Component
    {
        private List<Component> ingredientes = new List<Component>();

        public decimal CostoTotal{
            get{
                decimal costo = 0;
                foreach (var oElemento in ingredientes){
                    if (oElemento.GetType().Name == "PastelComposite")
                        costo += ((PastelComposite)oElemento).CostoTotal;
                    else
                        costo += oElemento.Costo;
                }
                return costo;
            }
        }

        public void Add(Component oElemento){ingredientes.Add(oElemento);}

        public void Remove(Component oElemento){ingredientes.Remove(oElemento);}

        public PastelComposite(string Nombre, decimal Costo=0) : base(Nombre, Costo){}
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

The screenshot shows the Microsoft Visual Studio IDE interface. In the center, there is a code editor window displaying the `Ingrediente.cs` file. The code defines a class `Ingrediente` that implements the `Component` interface. The class has properties `Cantidad` and `Unidad`, and a constructor that initializes them. The Solution Explorer on the right shows a project named "patronesDiseno" containing several files under the "Composite" folder, including `Component.cs`, `Ingrediente.cs`, and `PastelComposite.cs`. The status bar at the bottom indicates "121 %", "No se encontraron problemas.", and the date "04/07/2022".

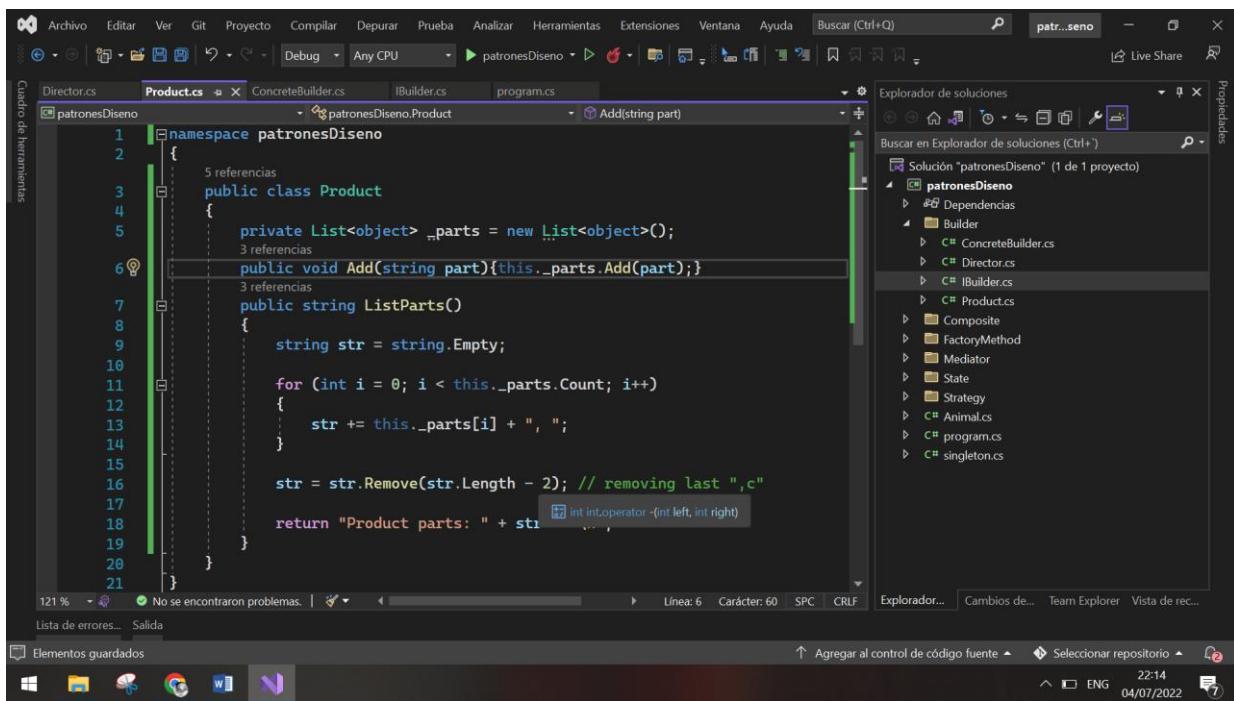
```
1 namespace patronesDiseno
2 {
3     class Ingrediente : Component
4     {
5         public int Cantidad { get; set; }
6         public string Unidad { get; set; }
7
8         public Ingrediente(string Nombre, decimal Costo, int Cantidad, string
9             : base(Nombre, Costo)
10            {
11                this.Cantidad = Cantidad;
12                this.Unidad = Unidad;
13            }
14        }
15    }
16}
17
```

### Ejemplo Builder

The screenshot shows the Microsoft Visual Studio IDE interface. In the center, there is a code editor window displaying the `Director.cs` file. The code defines an internal class `Director` that manages a `IBuilder` instance. It has methods `BuildMinimalViableProduct()` and `BuildFullFeaturedProduct()`, each calling the `BuildPartA()`, `BuildPartB()`, and `BuildPartC()` methods of the builder. The Solution Explorer on the right shows a project named "patronesDiseno" containing files under the "Builder" folder, including `ConcreteBuilder.cs`, `Director.cs`, `IBuilder.cs`, and `Product.cs`. The status bar at the bottom indicates "121 %", "0", "1", "1", "SPC", "CRLF", and the date "04/07/2022".

```
1 namespace patronesDiseno
2 {
3     internal class Director
4     {
5         private IBuilder _builder;
6         public IBuilder Builder{set { _builder = value; }}
7         public void BuildMinimalViableProduct()
8         {
9             this._builder.BuildPartA();
10        }
11
12        public void BuildFullFeaturedProduct()
13        {
14            this._builder.BuildPartA();
15            this._builder.BuildPartB();
16            this._builder.BuildPartC();
17        }
18    }
19}
20
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

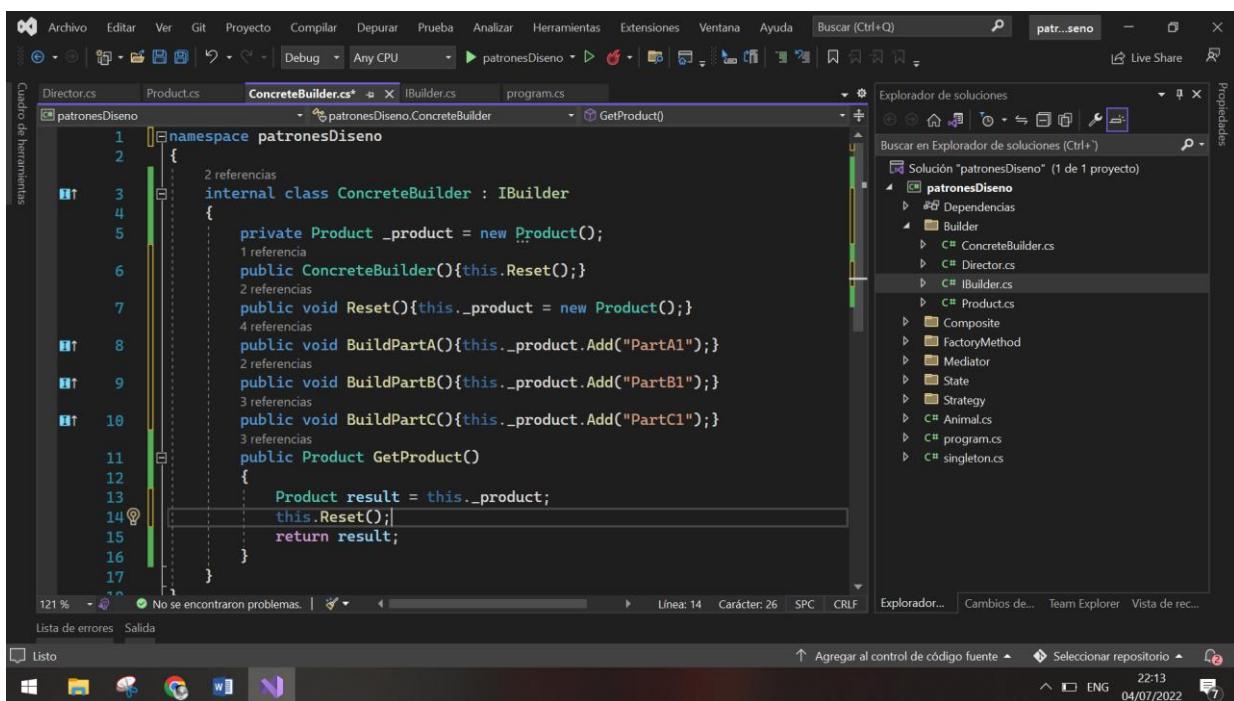


Screenshot of Microsoft Visual Studio showing the Product class code. The code implements the Composite pattern, allowing parts to be added to a product object.

```

namespace patronesDiseno
{
    public class Product
    {
        private List<object> _parts = new List<object>();
        public void Add(string part){this._parts.Add(part);}
        public string ListParts()
        {
            string str = string.Empty;
            for (int i = 0; i < this._parts.Count; i++)
            {
                str += this._parts[i] + ", ";
            }
            str = str.Remove(str.Length - 2); // removing last ","
            return "Product parts: " + str;
        }
    }
}

```



Screenshot of Microsoft Visual Studio showing the ConcreteBuilder class code. This class implements the Builder pattern, providing methods to build different parts of a product.

```

namespace patronesDiseno
{
    internal class ConcreteBuilder : IBuilder
    {
        private Product _product = new Product();
        public ConcreteBuilder(){this.Reset();}
        public void Reset(){this._product = new Product();}
        public void BuildPartA(){this._product.Add("PartA1");}
        public void BuildPartB(){this._product.Add("PartB1");}
        public void BuildPartC(){this._product.Add("PartC1");}
        public Product GetProduct()
        {
            Product result = this._product;
            this.Reset();
            return result;
        }
    }
}

```

```

namespace patronesDiseno
{
    internal class program
    {
        static void Main(string[] args)
        {
            //singleton
            Console.WriteLine(singleton.Instance.mensaje);
            Console.ReadLine();

            //prototype
        }
    }
}

```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
Animal oAnimal = new Animal() { Nombre = "Oveja Dolly", Patas = 4 };
Animal oAnimalClonado = oAnimal.Clone() as Animal;
oAnimalClonado.Patas = 5;

Console.WriteLine(oAnimal.Patas);

//FactoryMethod
FactoryMethod.bebidaEmbriagante oBebida =
FactoryMethod.creador.creadorBebida(FactoryMethod.creador.cerveza);
Console.WriteLine(oBebida.CuentoEmbriagaPorHora);
FactoryMethod.bebidaEmbriagante oBebida1 =
FactoryMethod.creador.creadorBebida(FactoryMethod.creador.cerveza);
Console.WriteLine(oBebida1.CuentoEmbriagaPorHora);

//strategy
Strategy.estrategiasBorrachoContexto oBorracho = new
Strategy.estrategiasBorrachoContexto();
oBorracho.conquistar();
oBorracho.EstablecerConquistaCerveza();
oBorracho.conquistar();

//Mediator

Mediator.Mediator mediador = new Mediator.Mediator();
Mediator.ICollection oPepe = new Mediator.User(mediador);
Mediator.ICollection oAdmin = new Mediator.UserAdmin(mediador);

mediador.add(oPepe);
mediador.add(oAdmin);

oPepe.communicate("oye admin tengo un problema");
oAdmin.communicate("Dime cual es el problema");

//State
ServidorContext oServior = new ServidorContext();
oServior.State = new DisponibleServerState();

oServior.AtenderSolicitud();
oServior.AtenderSolicitud();

//Composite
Ingrediente oIngrediente1 = new Ingrediente("Harina", 100, 200,
"gramos");
Ingrediente oIngrediente2 = new Ingrediente("Leche", 20, 1,
"litro");
Ingrediente oIngrediente3 = new Ingrediente("Huevo", 20, 1, "kilo");

PastelComposite oPastel = new PastelComposite("Pastel de leche");
oPastel.Add(oIngrediente1);
oPastel.Add(oIngrediente2);
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
oPastel.Add(oIngrediente3);

// Console.WriteLine(oPastel.Costo);

Ingrediente oIngrediente4 = new Ingrediente("Chocolate", 200, 1,
"litro");
PastelComposite oPastelChocolateYLeche = new PastelComposite("Pastel
compuesto");

oPastelChocolateYLeche.Add(oIngrediente4);
oPastelChocolateYLeche.Add(oPastel);

Console.WriteLine(oPastelChocolateYLeche.CostoTotal);

//Builder
var director = new Director();
var builder = new ConcreteBuilder();
director.Builder = builder;

Console.WriteLine("Standard basic product:");
director.BuildMinimalViableProduct();
Console.WriteLine(builder.GetProduct().ListParts());

Console.WriteLine("Standard full featured product:");
director.BuildFullFeaturedProduct();
Console.WriteLine(builder.GetProduct().ListParts());

// Remember, the Builder pattern can be used without a Director
// class.
Console.WriteLine("Custom product:");
builder.BuildPartA();
builder.BuildPartC();
Console.WriteLine(builder.GetProduct().ListParts());

//AbstractFactory
new AbstractFactory.Client().Main();

}

}

}
```

### Ejemplo Abstract Factory

```
using System;

namespace RefactoringGuru.DesignPatterns.AbstractFactory.Conceptual
{
    // The Abstract Factory interface declares a set of methods that return
    // different abstract products. These products are called a family and are
    // related by a high-level theme or concept. Products of one family are
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
// usually able to collaborate among themselves. A family of products may
// have several variants, but the products of one variant are incompatible
// with products of another.
public interface IAbstractFactory
{
    IAbstractProductA CreateProductA();

    IAbstractProductB CreateProductB();
}

// Concrete Factories produce a family of products that belong to a single
// variant. The factory guarantees that resulting products are compatible.
// Note that signatures of the Concrete Factory's methods return an abstract
// product, while inside the method a concrete product is instantiated.
class ConcreteFactory1 : IAbstractFactory
{
    public IAbstractProductA CreateProductA()
    {
        return new ConcreteProductA1();
    }

    public IAbstractProductB CreateProductB()
    {
        return new ConcreteProductB1();
    }
}

// Each Concrete Factory has a corresponding product variant.
class ConcreteFactory2 : IAbstractFactory
{
    public IAbstractProductA CreateProductA()
    {
        return new ConcreteProductA2();
    }

    public IAbstractProductB CreateProductB()
    {
        return new ConcreteProductB2();
    }
}

// Each distinct product of a product family should have a base interface.
// All variants of the product must implement this interface.
public interface IAbstractProductA
{
    string UsefulFunctionA();
}

// Concrete Products are created by corresponding Concrete Factories.
class ConcreteProductA1 : IAbstractProductA
{
    public string UsefulFunctionA()
    {
        return "The result of the product A1.";
    }
}

class ConcreteProductA2 : IAbstractProductA
{
    public string UsefulFunctionA()
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
{  
    return "The result of the product A2.";  
}  
  
// Here's the base interface of another product. All products can  
// interact with each other, but proper interaction is possible only between  
// products of the same concrete variant.  
public interface IAbstractProductB  
{  
    // Product B is able to do its own thing...  
    string UsefulFunctionB();  
  
    // ...but it also can collaborate with the ProductA.  
    //  
    // The Abstract Factory makes sure that all products it creates are of  
    // the same variant and thus, compatible.  
    string AnotherUsefulFunctionB(IAbstractProductA collaborator);  
}  
  
// Concrete Products are created by corresponding Concrete Factories.  
class ConcreteProductB1 : IAbstractProductB  
{  
    public string UsefulFunctionB()  
    {  
        return "The result of the product B1.";  
    }  
  
    // The variant, Product B1, is only able to work correctly with the  
    // variant, Product A1. Nevertheless, it accepts any instance of  
    // AbstractProductA as an argument.  
    public string AnotherUsefulFunctionB(IAbstractProductA collaborator)  
    {  
        var result = collaborator.UsefulFunctionA();  
  
        return $"The result of the B1 collaborating with the ({result})";  
    }  
}  
  
class ConcreteProductB2 : IAbstractProductB  
{  
    public string UsefulFunctionB()  
    {  
        return "The result of the product B2.";  
    }  
  
    // The variant, Product B2, is only able to work correctly with the  
    // variant, Product A2. Nevertheless, it accepts any instance of  
    // AbstractProductA as an argument.  
    public string AnotherUsefulFunctionB(IAbstractProductA collaborator)  
    {  
        var result = collaborator.UsefulFunctionA();  
  
        return $"The result of the B2 collaborating with the ({result})";  
    }  
}  
  
// The client code works with factories and products only through abstract  
// types: AbstractFactory and AbstractProduct. This lets you pass any  
// factory or product subclass to the client code without breaking it.
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
class Client
{
    public void Main()
    {
        // The client code can work with any concrete factory class.
        Console.WriteLine("Client: Testing client code with the first factory
type...");
        ClientMethod(new ConcreteFactory1());
        Console.WriteLine();

        Console.WriteLine("Client: Testing the same client code with the second
factory type...");
        ClientMethod(new ConcreteFactory2());
    }

    public void ClientMethod(IAbstractFactory factory)
    {
        var productA = factory.CreateProductA();
        var productB = factory.CreateProductB();

        Console.WriteLine(productB.UsefulFunctionB());
        Console.WriteLine(productB.AnotherUsefulFunctionB(productA));
    }
}

class Program
{
    static void Main(string[] args)
    {
        new Client().Main();
    }
}
```

### Ejemplo Bridge

```
using System;

namespace RefactoringGuru.DesignPatterns.Bridge.Conceptual
{
    // The Abstraction defines the interface for the "control" part of the two
    // class hierarchies. It maintains a reference to an object of the
    // Implementation hierarchy and delegates all of the real work to this
    // object.
    class Abstraction
    {
        protected IImplementation _implementation;

        public Abstraction(IImplementation implementation)
        {
            this._implementation = implementation;
        }

        public virtual string Operation()
        {
            return "Abstract: Base operation with:\n" +
                _implementation.OperationImplementation();
        }
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
}

// You can extend the Abstraction without changing the Implementation
// classes.
class ExtendedAbstraction : Abstraction
{
    public ExtendedAbstraction(IIImplementation implementation) :
base(implementation)
    {

    }

    public override string Operation()
    {
        return "ExtendedAbstraction: Extended operation with:\n" +
            base._implementation.OperationImplementation();
    }
}

// The Implementation defines the interface for all implementation classes.
// It doesn't have to match the Abstraction's interface. In fact, the two
// interfaces can be entirely different. Typically the Implementation
// interface provides only primitive operations, while the Abstraction
// defines higher-level operations based on those primitives.
public interface IIImplementation
{
    string OperationImplementation();
}

// Each Concrete Implementation corresponds to a specific platform and
// implements the Implementation interface using that platform's API.
class ConcreteImplementationA : IIImplementation
{
    public string OperationImplementation()
    {
        return "ConcreteImplementationA: The result in platform A.\n";
    }
}

class ConcreteImplementationB : IIImplementation
{
    public string OperationImplementation()
    {
        return "ConcreteImplementationB: The result in platform B.\n";
    }
}

class Client
{
    // Except for the initialization phase, where an Abstraction object gets
    // linked with a specific Implementation object, the client code should
    // only depend on the Abstraction class. This way the client code can
    // support any abstraction-implementation combination.
    public void ClientCode(Abstraction abstraction)
    {
        Console.WriteLine(abstraction.Operation());
    }
}

class Program
{
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
static void Main(string[] args)
{
    Client client = new Client();

    Abstraction abstraction;
    // The client code should be able to work with any pre-configured
    // abstraction-implementation combination.
    abstraction = new Abstraction(new ConcreteImplementationA());
    client.ClientCode(abstraction);

    Console.WriteLine();

    abstraction = new ExtendedAbstraction(new ConcreteImplementationB());
    client.ClientCode(abstraction);
}
}
```

### Ejemplo Adapter

```
using System;

namespace RefactoringGuru.DesignPatterns.Adapter.Conceptual
{
    // The Target defines the domain-specific interface used by the client code.
    public interface ITarget
    {
        string GetRequest();
    }

    // The Adaptee contains some useful behavior, but its interface is
    // incompatible with the existing client code. The Adaptee needs some
    // adaptation before the client code can use it.
    class Adaptee
    {
        public string GetSpecificRequest()
        {
            return "Specific request.";
        }
    }

    // The Adapter makes the Adaptee's interface compatible with the Target's
    // interface.
    class Adapter : ITarget
    {
        private readonly Adaptee _adaptee;

        public Adapter(Adaptee adaptee)
        {
            this._adaptee = adaptee;
        }
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
public string GetRequest()
{
    return $"This is '{this._adaptee.GetSpecificRequest()}'";
}

class Program
{
    static void Main(string[] args)
    {
        Adaptee adaptee = new Adaptee();
        ITarget target = new Adapter(adaptee);

        Console.WriteLine("Adaptee interface is incompatible with the
client.");
        Console.WriteLine("But with adapter client can call it's method.");

        Console.WriteLine(target.GetRequest());
    }
}
```

### Ejemplo Decorator

```
using System;

namespace RefactoringGuru.DesignPatterns.Composite.Conceptual
{
    // The base Component interface defines operations that can be altered by
    // decorators.
    public abstract class Component
    {
        public abstract string Operation();
    }

    // Concrete Components provide default implementations of the operations.
    // There might be several variations of these classes.
    class ConcreteComponent : Component
    {
        public override string Operation()
        {
            return "ConcreteComponent";
        }
    }

    // The base Decorator class follows the same interface as the other
    // components. The primary purpose of this class is to define the wrapping
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
// interface for all concrete decorators. The default implementation of the
// wrapping code might include a field for storing a wrapped component and
// the means to initialize it.
abstract class Decorator : Component
{
    protected Component _component;

    public Decorator(Component component)
    {
        this._component = component;
    }

    public void SetComponent(Component component)
    {
        this._component = component;
    }

    // The Decorator delegates all work to the wrapped component.
    public override string Operation()
    {
        if (this._component != null)
        {
            return this._component.Operation();
        }
        else
        {
            return string.Empty;
        }
    }
}

// Concrete Decorators call the wrapped object and alter its result in some
// way.
class ConcreteDecoratorA : Decorator
{
    public ConcreteDecoratorA(Component comp) : base(comp)
    {

        // Decorators may call parent implementation of the operation, instead
        // of calling the wrapped object directly. This approach simplifies
        // extension of decorator classes.
        public override string Operation()
        {
            return $"ConcreteDecoratorA({base.Operation()})";
        }
    }

    // Decorators can execute their behavior either before or after the call to
    // a wrapped object.
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
class ConcreteDecoratorB : Decorator
{
    public ConcreteDecoratorB(Component comp) : base(comp)
    {

    }

    public override string Operation()
    {
        return $"ConcreteDecoratorB({base.Operation()})";
    }
}

public class Client
{
    // The client code works with all objects using the Component interface.
    // This way it can stay independent of the concrete classes of
    // components it works with.
    public void ClientCode(Component component)
    {
        Console.WriteLine("RESULT: " + component.Operation());
    }
}

class Program
{
    static void Main(string[] args)
    {
        Client client = new Client();

        var simple = new ConcreteComponent();
        Console.WriteLine("Client: I get a simple component:");
        client.ClientCode(simple);
        Console.WriteLine();

        // ...as well as decorated ones.
        //
        // Note how decorators can wrap not only simple components but the
        // other decorators as well.
        ConcreteDecoratorA decorator1 = new ConcreteDecoratorA(simple);
        ConcreteDecoratorB decorator2 = new ConcreteDecoratorB(decorator1);
        Console.WriteLine("Client: Now I've got a decorated component:");
        client.ClientCode(decorator2);
    }
}
```

**Ejemplo Facade**

```

using System;

namespace RefactoringGuru.DesignPatterns.Facade.Conceptual
{
    // The Facade class provides a simple interface to the complex logic of one
    // or several subsystems. The Facade delegates the client requests to the
    // appropriate objects within the subsystem. The Facade is also responsible
    // for managing their lifecycle. All of this shields the client from the
    // undesired complexity of the subsystem.
    public class Facade
    {
        protected Subsystem1 _subsystem1;

        protected Subsystem2 _subsystem2;

        public Facade(Subsystem1 subsystem1, Subsystem2 subsystem2)
        {
            this._subsystem1 = subsystem1;
            this._subsystem2 = subsystem2;
        }

        // The Facade's methods are convenient shortcuts to the sophisticated
        // functionality of the subsystems. However, clients get only to a
        // fraction of a subsystem's capabilities.
        public string Operation()
        {
            string result = "Facade initializes subsystems:\n";
            result += this._subsystem1.operation1();
            result += this._subsystem2.operation1();
            result += "Facade orders subsystems to perform the action:\n";
            result += this._subsystem1.operationN();
            result += this._subsystem2.operationZ();
            return result;
        }
    }

    // The Subsystem can accept requests either from the facade or client
    // directly. In any case, to the Subsystem, the Facade is yet another
    // client, and it's not a part of the Subsystem.
    public class Subsystem1
    {
        public string operation1()
        {
            return "Subsystem1: Ready!\n";
        }

        public string operationN()
        {
    
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
        return "Subsystem1: Go!\n";
    }
}

// Some facades can work with multiple subsystems at the same time.
public class Subsystem2
{
    public string operation1()
    {
        return "Subsystem2: Get ready!\n";
    }

    public string operation2()
    {
        return "Subsystem2: Fire!\n";
    }
}

class Client
{
    // The client code works with complex subsystems through a simple
    // interface provided by the Facade. When a facade manages the lifecycle
    // of the subsystem, the client might not even know about the existence
    // of the subsystem. This approach lets you keep the complexity under
    // control.
    public static void ClientCode(Facade facade)
    {
        Console.WriteLine(facade.Operation());
    }
}

class Program
{
    static void Main(string[] args)
    {
        // The client code may have some of the subsystem's objects already
        // created. In this case, it might be worthwhile to initialize the
        // Facade with these objects instead of letting the Facade create
        // new instances.
        Subsystem1 subsystem1 = new Subsystem1();
        Subsystem2 subsystem2 = new Subsystem2();
        Facade facade = new Facade(subsystem1, subsystem2);
        Client.ClientCode(facade);
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

## Ejemplo Flyweight

```

using System;
using System.Collections.Generic;
using System.Linq;
// Use Json.NET library, you can download it from NuGet Package Manager
using Newtonsoft.Json;

namespace RefactoringGuru.DesignPatterns.Flyweight.Conceptual
{
    // The Flyweight stores a common portion of the state (also called intrinsic
    // state) that belongs to multiple real business entities. The Flyweight
    // accepts the rest of the state (extrinsic state, unique for each entity)
    // via its method parameters.
    public class Flyweight
    {
        private Car _sharedState;

        public Flyweight(Car car)
        {
            this._sharedState = car;
        }

        public void Operation(Car uniqueState)
        {
            string s = JsonConvert.SerializeObject(this._sharedState);
            string u = JsonConvert.SerializeObject(uniqueState);
            Console.WriteLine($"Flyweight: Displaying shared {s} and unique {u}
state.");
        }
    }

    // The Flyweight Factory creates and manages the Flyweight objects. It
    // ensures that flyweights are shared correctly. When the client requests a
    // flyweight, the factory either returns an existing instance or creates a
    // new one, if it doesn't exist yet.
    public class FlyweightFactory
    {
        private List<Tuple<Flyweight, string>> flyweights = new
List<Tuple<Flyweight, string>>();

        public FlyweightFactory(params Car[] args)
        {
            foreach (var elem in args)
            {
                flyweights.Add(new Tuple<Flyweight, string>(new Flyweight(elem),
this.getKey(elem)));
            }
        }
    }
}

```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
// Returns a Flyweight's string hash for a given state.
public string getKey(Car key)
{
    List<string> elements = new List<string>();

    elements.Add(key.Model);
    elements.Add(key.Color);
    elements.Add(key.Company);

    if (key.Owner != null && key.Number != null)
    {
        elements.Add(key.Number);
        elements.Add(key.Owner);
    }

    elements.Sort();

    return string.Join("_", elements);
}

// Returns an existing Flyweight with a given state or creates a new
// one.
public Flyweight GetFlyweight(Car sharedState)
{
    string key = this.getKey(sharedState);

    if (flyweights.Where(t => t.Item2 == key).Count() == 0)
    {
        Console.WriteLine("FlyweightFactory: Can't find a flyweight,
creating new one.");
        this.flyweights.Add(new Tuple<Flyweight, string>(new
Flyweight(sharedState), key));
    }
    else
    {
        Console.WriteLine("FlyweightFactory: Reusing existing
flyweight.");
    }
    return this.flyweights.Where(t => t.Item2 ==
key).FirstOrDefault().Item1;
}

public void listFlyweights()
{
    var count = flyweights.Count;
    Console.WriteLine($"\\nFlyweightFactory: I have {count}
flyweights:");
    foreach (var flyweight in flyweights)
    {
        Console.WriteLine(flyweight.Item2);
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
        }
    }
}

public class Car
{
    public string Owner { get; set; }

    public string Number { get; set; }

    public string Company { get; set; }

    public string Model { get; set; }

    public string Color { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        // The client code usually creates a bunch of pre-populated
        // flyweights in the initialization stage of the application.
        var factory = new FlyweightFactory(
            new Car { Company = "Chevrolet", Model = "Camaro2018", Color =
"pink" },
            new Car { Company = "Mercedes Benz", Model = "C300", Color =
"black" },
            new Car { Company = "Mercedes Benz", Model = "C500", Color =
"red" },
            new Car { Company = "BMW", Model = "M5", Color = "red" },
            new Car { Company = "BMW", Model = "X6", Color = "white" }
        );
        factory.listFlyweights();

        addCarToPoliceDatabase(factory, new Car {
            Number = "CL234IR",
            Owner = "James Doe",
            Company = "BMW",
            Model = "M5",
            Color = "red"
        });

        addCarToPoliceDatabase(factory, new Car {
            Number = "CL234IR",
            Owner = "James Doe",
            Company = "BMW",
            Model = "X1",
            Color = "red"
        });
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
        factory.listFlyweights();
    }

    public static void addCarToPoliceDatabase(FlyweightFactory factory, Car car)
    {
        Console.WriteLine("\nClient: Adding a car to database.");

        var flyweight = factory.GetFlyweight(new Car {
            Color = car.Color,
            Model = car.Model,
            Company = car.Company
        });

        // The client code either stores or calculates extrinsic state and
        // passes it to the flyweight's methods.
        flyweight.Operation(car);
    }
}
```

### Ejemplo Proxy

```
using System;

namespace RefactoringGuru.DesignPatterns.Proxy.Conceptual
{
    // The Subject interface declares common operations for both RealSubject and
    // the Proxy. As long as the client works with RealSubject using this
    // interface, you'll be able to pass it a proxy instead of a real subject.
    public interface ISubject
    {
        void Request();
    }

    // The RealSubject contains some core business logic. Usually, RealSubjects
    // are capable of doing some useful work which may also be very slow or
    // sensitive - e.g. correcting input data. A Proxy can solve these issues
    // without any changes to the RealSubject's code.
    class RealSubject : ISubject
    {
        public void Request()
        {
            Console.WriteLine("RealSubject: Handling Request.");
        }
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
// The Proxy has an interface identical to the RealSubject.
class Proxy : ISubject
{
    private RealSubject _realSubject;

    public Proxy(RealSubject realSubject)
    {
        this._realSubject = realSubject;
    }

    // The most common applications of the Proxy pattern are lazy loading,
    // caching, controlling the access, logging, etc. A Proxy can perform
    // one of these things and then, depending on the result, pass the
    // execution to the same method in a linked RealSubject object.
    public void Request()
    {
        if (this.CheckAccess())
        {
            this._realSubject.Request();

            this.LogAccess();
        }
    }

    public bool CheckAccess()
    {
        // Some real checks should go here.
        Console.WriteLine("Proxy: Checking access prior to firing a real
request.");

        return true;
    }

    public void LogAccess()
    {
        Console.WriteLine("Proxy: Logging the time of request.");
    }
}

public class Client
{
    // The client code is supposed to work with all objects (both subjects
    // and proxies) via the Subject interface in order to support both real
    // subjects and proxies. In real life, however, clients mostly work with
    // their real subjects directly. In this case, to implement the pattern
    // more easily, you can extend your proxy from the real subject's class.
    public void ClientCode(ISubject subject)
    {
        // ...
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
        subject.Request();

        // ...
    }

}

class Program
{
    static void Main(string[] args)
    {
        Client client = new Client();

        Console.WriteLine("Client: Executing the client code with a real
subject:");
        RealSubject realSubject = new RealSubject();
        client.ClientCode(realSubject);

        Console.WriteLine();

        Console.WriteLine("Client: Executing the same client code with a
proxy:");
        Proxy proxy = new Proxy(realSubject);
        client.ClientCode(proxy);
    }
}
```

### Ejemplo Chain of Responsibility

```
using System;
using System.Collections.Generic;

namespace RefactoringGuru.DesignPatterns.ChainOfResponsibility.Conceptual
{
    // The Handler interface declares a method for building the chain of
    // handlers. It also declares a method for executing a request.
    public interface IHandler
    {
        IHandler SetNext(IHandler handler);

        object Handle(object request);
    }

    // The default chaining behavior can be implemented inside a base handler
    // class.
    abstract class AbstractHandler : IHandler
    {
        private IHandler _nextHandler;
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
public IHandler SetNext(IHandler handler)
{
    this._nextHandler = handler;

    // Returning a handler from here will let us link handlers in a
    // convenient way like this:
    // monkey.SetNext(squirrel).SetNext(dog);
    return handler;
}

public virtual object Handle(object request)
{
    if (this._nextHandler != null)
    {
        return this._nextHandler.Handle(request);
    }
    else
    {
        return null;
    }
}

class MonkeyHandler : AbstractHandler
{
    public override object Handle(object request)
    {
        if ((request as string) == "Banana")
        {
            return $"Monkey: I'll eat the {request.ToString()}.\\n";
        }
        else
        {
            return base.Handle(request);
        }
    }
}

class SquirrelHandler : AbstractHandler
{
    public override object Handle(object request)
    {
        if (request.ToString() == "Nut")
        {
            return $"Squirrel: I'll eat the {request.ToString()}.\\n";
        }
        else
        {
            return base.Handle(request);
        }
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
        }
    }
}

class DogHandler : AbstractHandler
{
    public override object Handle(object request)
    {
        if (request.ToString() == "MeatBall")
        {
            return $"Dog: I'll eat the {request.ToString()}.\n";
        }
        else
        {
            return base.Handle(request);
        }
    }
}

class Client
{
    // The client code is usually suited to work with a single handler. In
    // most cases, it is not even aware that the handler is part of a chain.
    public static void ClientCode(AbstractHandler handler)
    {
        foreach (var food in new List<string> { "Nut", "Banana", "Cup of
coffee" })
        {
            Console.WriteLine($"Client: Who wants a {food}?");

            var result = handler.Handle(food);

            if (result != null)
            {
                Console.Write($"    {result}");
            }
            else
            {
                Console.WriteLine($"    {food} was left untouched.");
            }
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        // The other part of the client code constructs the actual chain.
        var monkey = new MonkeyHandler();
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
var squirrel = new SquirrelHandler();
var dog = new DogHandler();

monkey.SetNext(squirrel).SetNext(dog);

// The client should be able to send a request to any handler, not
// just the first one in the chain.
Console.WriteLine("Chain: Monkey > Squirrel > Dog\n");
Client.ClientCode(monkey);
Console.WriteLine();

Console.WriteLine("Subchain: Squirrel > Dog\n");
Client.ClientCode(squirrel);
}

}

}
```

### Ejemplo Command

```
using System;

namespace RefactoringGuru.DesignPatterns.Command.Conceptual
{
    // The Command interface declares a method for executing a command.
    public interface ICommand
    {
        void Execute();
    }

    // Some commands can implement simple operations on their own.
    class SimpleCommand : ICommand
    {
        private string _payload = string.Empty;

        public SimpleCommand(string payload)
        {
            this._payload = payload;
        }

        public void Execute()
        {
            Console.WriteLine($"SimpleCommand: See, I can do simple things like
printing ({this._payload})");
        }
    }

    // However, some commands can delegate more complex operations to other
    // objects, called "receivers."
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
class ComplexCommand : ICommand
{
    private Receiver _receiver;

    // Context data, required for launching the receiver's methods.
    private string _a;

    private string _b;

    // Complex commands can accept one or several receiver objects along
    // with any context data via the constructor.
    public ComplexCommand(Receiver receiver, string a, string b)
    {
        this._receiver = receiver;
        this._a = a;
        this._b = b;
    }

    // Commands can delegate to any methods of a receiver.
    public void Execute()
    {
        Console.WriteLine("ComplexCommand: Complex stuff should be done by a
receiver object.");
        this._receiver.DoSomething(this._a);
        this._receiver.DoSomethingElse(this._b);
    }
}

// The Receiver classes contain some important business logic. They know how
// to perform all kinds of operations, associated with carrying out a
// request. In fact, any class may serve as a Receiver.
class Receiver
{
    public void DoSomething(string a)
    {
        Console.WriteLine($"Receiver: Working on ({a}.)");
    }

    public void DoSomethingElse(string b)
    {
        Console.WriteLine($"Receiver: Also working on ({b}.)");
    }
}

// The Invoker is associated with one or several commands. It sends a
// request to the command.
class Invoker
{
    private ICommand _onStart;
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
private ICommand _onFinish;

// Initialize commands.
public void SetOnStart(ICommand command)
{
    this._onStart = command;
}

public void SetOnFinish(ICommand command)
{
    this._onFinish = command;
}

// The Invoker does not depend on concrete command or receiver classes.
// The Invoker passes a request to a receiver indirectly, by executing a
// command.
public void DoSomethingImportant()
{
    Console.WriteLine("Invoker: Does anybody want something done before
I begin?");
    if (this._onStart is ICommand)
    {
        this._onStart.Execute();
    }

    Console.WriteLine("Invoker: ...doing something really
important...");

    Console.WriteLine("Invoker: Does anybody want something done after I
finish?");
    if (this._onFinish is ICommand)
    {
        this._onFinish.Execute();
    }
}

class Program
{
    static void Main(string[] args)
    {
        // The client code can parameterize an invoker with any commands.
        Invoker invoker = new Invoker();
        invoker.SetOnStart(new SimpleCommand("Say Hi!"));
        Receiver receiver = new Receiver();
        invoker.SetOnFinish(new ComplexCommand(receiver, "Send email", "Save
report"));

        invoker.DoSomethingImportant();
    }
}
```

```
}
```

## Ejemplo Iterator

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace RefactoringGuru.DesignPatterns.Iterator.Conceptual
{
    abstract class Iterator : IEnumerator
    {
        object IEnumerator.Current => Current();

        // Returns the key of the current element
        public abstract int Key();

        // Returns the current element
        public abstract object Current();

        // Move forward to next element
        public abstract bool MoveNext();

        // Rewinds the Iterator to the first element
        public abstract void Reset();
    }

    abstract class IteratorAggregate : IEnumerable
    {
        // Returns an Iterator or another IteratorAggregate for the implementing
        // object.
        public abstract IEnumerator GetEnumerator();
    }

    // Concrete Iterators implement various traversal algorithms. These classes
    // store the current traversal position at all times.
    class AlphabeticalOrderIterator : Iterator
    {
        private WordsCollection _collection;

        // Stores the current traversal position. An iterator may have a lot of
        // other fields for storing iteration state, especially when it is
        // supposed to work with a particular kind of collection.
        private int _position = -1;

        private bool _reverse = false;
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
public AlphabeticalOrderIterator(WordsCollection collection, bool reverse = false)
{
    this._collection = collection;
    this._reverse = reverse;

    if (reverse)
    {
        this._position = collection.getItems().Count;
    }
}

public override object Current()
{
    return this._collection.getItems()[_position];
}

public override int Key()
{
    return this._position;
}

public override bool MoveNext()
{
    int updatedPosition = this._position + (this._reverse ? -1 : 1);

    if (updatedPosition >= 0 && updatedPosition <
this._collection.getItems().Count)
    {
        this._position = updatedPosition;
        return true;
    }
    else
    {
        return false;
    }
}

public override void Reset()
{
    this._position = this._reverse ? this._collection.getItems().Count -
1 : 0;
}

// Concrete Collections provide one or several methods for retrieving fresh
// iterator instances, compatible with the collection class.
class WordsCollection : IteratorAggregate
{
    List<string> _collection = new List<string>();
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
bool _direction = false;

public void ReverseDirection()
{
    _direction = !_direction;
}

public List<string> getItems()
{
    return _collection;
}

public void AddItem(string item)
{
    this._collection.Add(item);
}

public override IEnumerator GetEnumerator()
{
    return new AlphabeticalOrderIterator(this, _direction);
}

class Program
{
    static void Main(string[] args)
    {
        // The client code may or may not know about the Concrete Iterator
        // or Collection classes, depending on the level of indirection you
        // want to keep in your program.
        var collection = new WordsCollection();
        collection.AddItem("First");
        collection.AddItem("Second");
        collection.AddItem("Third");

        Console.WriteLine("Straight traversal:");

        foreach (var element in collection)
        {
            Console.WriteLine(element);
        }

        Console.WriteLine("\nReverse traversal:");

        collection.ReverseDirection();

        foreach (var element in collection)
        {
            Console.WriteLine(element);
        }
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
        }
    }
}
```

### Ejemplo Memento

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace RefactoringGuru.DesignPatterns.Memento.Conceptual
{
    // The Originator holds some important state that may change over time. It
    // also defines a method for saving the state inside a memento and another
    // method for restoring the state from it.
    class Originator
    {
        // For the sake of simplicity, the originator's state is stored inside a
        // single variable.
        private string _state;

        public Originator(string state)
        {
            this._state = state;
            Console.WriteLine("Originator: My initial state is: " + state);
        }

        // The Originator's business logic may affect its internal state.
        // Therefore, the client should backup the state before launching
        // methods of the business logic via the save() method.
        public void DoSomething()
        {
            Console.WriteLine("Originator: I'm doing something important.");
            this._state = this.GenerateRandomString(30);
            Console.WriteLine($"Originator: and my state has changed to:
{_state}");
        }

        private string GenerateRandomString(int length = 10)
        {
            string allowedSymbols =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
            string result = string.Empty;

            while (length > 0)
            {
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
        result += allowedSymbols[new Random().Next(0,
allowedSymbols.Length)];  
  
        Thread.Sleep(12);  
  
        length--;  
    }  
  
    return result;  
}  
  
// Saves the current state inside a memento.  
public IMemento Save()  
{  
    return new ConcreteMemento(this._state);  
}  
  
// Restores the Originator's state from a memento object.  
public void Restore(IMemento memento)  
{  
    if (!(memento is ConcreteMemento))  
    {  
        throw new Exception("Unknown memento class " +  
memento.ToString());  
    }  
  
    this._state = memento.GetState();  
    Console.WriteLine($"Originator: My state has changed to: {_state}");  
}  
}  
  
// The Memento interface provides a way to retrieve the memento's metadata,  
// such as creation date or name. However, it doesn't expose the  
// Originator's state.  
public interface IMemento  
{  
    string GetName();  
  
    string GetState();  
  
    DateTime GetDate();  
}  
  
// The Concrete Memento contains the infrastructure for storing the  
// Originator's state.  
class ConcreteMemento : IMemento  
{  
    private string _state;  
  
    private DateTime _date;
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
public ConcreteMemento(string state)
{
    this._state = state;
    this._date = DateTime.Now;
}

// The Originator uses this method when restoring its state.
public string GetState()
{
    return this._state;
}

// The rest of the methods are used by the Caretaker to display
// metadata.
public string GetName()
{
    return $"{this._date} / ({this._state.Substring(0, 9)})...";
}

public DateTime GetDate()
{
    return this._date;
}

// The Caretaker doesn't depend on the Concrete Memento class. Therefore, it
// doesn't have access to the originator's state, stored inside the memento.
// It works with all mementos via the base Memento interface.
class Caretaker
{
    private List<IMemento> _mementos = new List<IMemento>();

    private Originator _originator = null;

    public Caretaker(Originator originator)
    {
        this._originator = originator;
    }

    public void Backup()
    {
        Console.WriteLine("\nCaretaker: Saving Originator's state...");
        this._mementos.Add(this._originator.Save());
    }

    public void Undo()
    {
        if (this._mementos.Count == 0)
        {

```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
        return;
    }

    var memento = this._mementos.Last();
    this._mementos.Remove(memento);

    Console.WriteLine("Caretaker: Restoring state to: " +
memento.GetName());

    try
    {
        this._originator.Restore(memento);
    }
    catch (Exception)
    {
        this.Undo();
    }
}

public void ShowHistory()
{
    Console.WriteLine("Caretaker: Here's the list of mementos:");

    foreach (var memento in this._mementos)
    {
        Console.WriteLine(memento.GetName());
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Client code.
        Originator originator = new Originator("Super-duper-super-puper-
super.");
        Caretaker caretaker = new Caretaker(originator);

        caretaker.Backup();
        originator.DoSomething();

        caretaker.Backup();
        originator.DoSomething();

        caretaker.Backup();
        originator.DoSomething();

        Console.WriteLine();
        caretaker.ShowHistory();
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
Console.WriteLine("\nClient: Now, let's rollback!\n");
caretaker.Undo();

Console.WriteLine("\n\nClient: Once more!\n");
caretaker.Undo();

Console.WriteLine();
}

}

}
```

### Ejemplo Observer

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace RefactoringGuru.DesignPatterns.Observer.Conceptual
{
    public interface IObserver
    {
        // Receive update from subject
        void Update(ISubject subject);
    }

    public interface ISubject
    {
        // Attach an observer to the subject.
        void Attach(IObserver observer);

        // Detach an observer from the subject.
        void Detach(IObserver observer);

        // Notify all observers about an event.
        void Notify();
    }

    // The Subject owns some important state and notifies observers when the
    // state changes.
    public class Subject : ISubject
    {
        // For the sake of simplicity, the Subject's state, essential to all
        // subscribers, is stored in this variable.
        public int State { get; set; } = -0;

        // List of subscribers. In real life, the list of subscribers can be
        // implemented using a linked list or a tree structure.
        private List<IObserver> observers = new List<IObserver>();
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
// stored more comprehensively (categorized by event type, etc.).  
private List<IObserver> _observers = new List<IObserver>();  
  
// The subscription management methods.  
public void Attach(IObserver observer)  
{  
    Console.WriteLine("Subject: Attached an observer.");  
    this._observers.Add(observer);  
}  
  
public void Detach(IObserver observer)  
{  
    this._observers.Remove(observer);  
    Console.WriteLine("Subject: Detached an observer.");  
}  
  
// Trigger an update in each subscriber.  
public void Notify()  
{  
    Console.WriteLine("Subject: Notifying observers...");  
  
    foreach (var observer in _observers)  
    {  
        observer.Update(this);  
    }  
}  
  
// Usually, the subscription logic is only a fraction of what a Subject  
// can really do. Subjects commonly hold some important business logic,  
// that triggers a notification method whenever something important is  
// about to happen (or after it).  
public void SomeBusinessLogic()  
{  
    Console.WriteLine("\nSubject: I'm doing something important.");  
    this.State = new Random().Next(0, 10);  
  
    Thread.Sleep(15);  
  
    Console.WriteLine("Subject: My state has just changed to: " +  
this.State);  
    this.Notify();  
}  
}  
  
// Concrete Observers react to the updates issued by the Subject they had  
// been attached to.  
class ConcreteObserverA : IObserver  
{  
    public void Update(ISubject subject)  
    {
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
        if ((subject as Subject).State < 3)
    {
        Console.WriteLine("ConcreteObserverA: Reacted to the event.");
    }
}

class ConcreteObserverB : IObserver
{
    public void Update(ISubject subject)
    {
        if ((subject as Subject).State == 0 || (subject as Subject).State >=
2)
        {
            Console.WriteLine("ConcreteObserverB: Reacted to the event.");
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        // The client code.
        var subject = new Subject();
        var observerA = new ConcreteObserverA();
        subject.Attach(observerA);

        var observerB = new ConcreteObserverB();
        subject.Attach(observerB);

        subject.SomeBusinessLogic();
        subject.SomeBusinessLogic();

        subject.Detach(observerB);

        subject.SomeBusinessLogic();
    }
}
}
```

### Ejemplo Template Method

```
using System;

namespace RefactoringGuru.DesignPatterns.TemplateMethod.Conceptual
{
    // The Abstract Class defines a template method that contains a skeleton of
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
// some algorithm, composed of calls to (usually) abstract primitive
// operations.
//
// Concrete subclasses should implement these operations, but leave the
// template method itself intact.
abstract class AbstractClass
{
    // The template method defines the skeleton of an algorithm.
    public void TemplateMethod()
    {
        this.BaseOperation1();
        this.RequiredOperations1();
        this.BaseOperation2();
        this.Hook1();
        this.RequiredOperation2();
        this.BaseOperation3();
        this.Hook2();
    }

    // These operations already have implementations.
    protected void BaseOperation1()
    {
        Console.WriteLine("AbstractClass says: I am doing the bulk of the
work");
    }

    protected void BaseOperation2()
    {
        Console.WriteLine("AbstractClass says: But I let subclasses override
some operations");
    }

    protected void BaseOperation3()
    {
        Console.WriteLine("AbstractClass says: But I am doing the bulk of
the work anyway");
    }

    // These operations have to be implemented in subclasses.
    protected abstract void RequiredOperations1();

    protected abstract void RequiredOperation2();

    // These are "hooks." Subclasses may override them, but it's not
    // mandatory since the hooks already have default (but empty)
    // implementation. Hooks provide additional extension points in some
    // crucial places of the algorithm.
    protected virtual void Hook1() { }

    protected virtual void Hook2() { }
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
}

// Concrete classes have to implement all abstract operations of the base
// class. They can also override some operations with a default
// implementation.
class ConcreteClass1 : AbstractClass
{
    protected override void RequiredOperations1()
    {
        Console.WriteLine("ConcreteClass1 says: Implemented Operation1");
    }

    protected override void RequiredOperation2()
    {
        Console.WriteLine("ConcreteClass1 says: Implemented Operation2");
    }
}

// Usually, concrete classes override only a fraction of base class'
// operations.
class ConcreteClass2 : AbstractClass
{
    protected override void RequiredOperations1()
    {
        Console.WriteLine("ConcreteClass2 says: Implemented Operation1");
    }

    protected override void RequiredOperation2()
    {
        Console.WriteLine("ConcreteClass2 says: Implemented Operation2");
    }

    protected override void Hook1()
    {
        Console.WriteLine("ConcreteClass2 says: Overridden Hook1");
    }
}

class Client
{
    // The client code calls the template method to execute the algorithm.
    // Client code does not have to know the concrete class of an object it
    // works with, as long as it works with objects through the interface of
    // their base class.
    public static void ClientCode(AbstractClass abstractClass)
    {
        // ...
        abstractClass.TemplateMethod();
        // ...
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Same client code can work with different
subclasses:");

        Client.ClientCode(new ConcreteClass1());

        Console.Write("\n");

        Console.WriteLine("Same client code can work with different
subclasses:");
        Client.ClientCode(new ConcreteClass2());
    }
}
```

### Ejemplo Visitor

```
using System;
using System.Collections.Generic;

namespace RefactoringGuru.DesignPatterns.Visitor.Conceptual
{
    // The Component interface declares an `accept` method that should take the
    // base visitor interface as an argument.
    public interface IComponent
    {
        void Accept(IVisitor visitor);
    }

    // Each Concrete Component must implement the `Accept` method in such a way
    // that it calls the visitor's method corresponding to the component's
    // class.
    public class ConcreteComponentA : IComponent
    {
        // Note that we're calling `VisitConcreteComponentA`, which matches the
        // current class name. This way we let the visitor know the class of the
        // component it works with.
        public void Accept(IVisitor visitor)
        {
            visitor.VisitConcreteComponentA(this);
        }

        // Concrete Components may have special methods that don't exist in
    }
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
// their base class or interface. The Visitor is still able to use these
// methods since it's aware of the component's concrete class.
public string ExclusiveMethodOfConcreteComponentA()
{
    return "A";
}

public class ConcreteComponentB : IComponent
{
    // Same here: VisitConcreteComponentB => ConcreteComponentB
    public void Accept(IVisitor visitor)
    {
        visitor.VisitConcreteComponentB(this);
    }

    public string SpecialMethodOfConcreteComponentB()
    {
        return "B";
    }
}

// The Visitor Interface declares a set of visiting methods that correspond
// to component classes. The signature of a visiting method allows the
// visitor to identify the exact class of the component that it's dealing
// with.
public interface IVisitor
{
    void VisitConcreteComponentA(ConcreteComponentA element);

    void VisitConcreteComponentB(ConcreteComponentB element);
}

// Concrete Visitors implement several versions of the same algorithm, which
// can work with all concrete component classes.
//
// You can experience the biggest benefit of the Visitor pattern when using
// it with a complex object structure, such as a Composite tree. In this
// case, it might be helpful to store some intermediate state of the
// algorithm while executing visitor's methods over various objects of the
// structure.
class ConcreteVisitor1 : IVisitor
{
    public void VisitConcreteComponentA(ConcreteComponentA element)
    {
        Console.WriteLine(element.ExclusiveMethodOfConcreteComponentA() + "
+ ConcreteVisitor1");
    }

    public void VisitConcreteComponentB(ConcreteComponentB element)
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
{  
    Console.WriteLine(element.SpecialMethodOfConcreteComponentB() + " +  
ConcreteVisitor1");  
}  
}  
  
class ConcreteVisitor2 : IVisitor  
{  
    public void VisitConcreteComponentA(ConcreteComponentA element)  
    {  
        Console.WriteLine(element.ExclusiveMethodOfConcreteComponentA() + "  
+ ConcreteVisitor2");  
    }  
  
    public void VisitConcreteComponentB(ConcreteComponentB element)  
    {  
        Console.WriteLine(element.SpecialMethodOfConcreteComponentB() + " +  
ConcreteVisitor2");  
    }  
}  
  
public class Client  
{  
    // The client code can run visitor operations over any set of elements  
    // without figuring out their concrete classes. The accept operation  
    // directs a call to the appropriate operation in the visitor object.  
    public static void ClientCode(List<IComponent> components, IVisitor  
visitor)  
    {  
        foreach (var component in components)  
        {  
            component.Accept(visitor);  
        }  
    }  
}  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        List<IComponent> components = new List<IComponent>  
        {  
            new ConcreteComponentA(),  
            new ConcreteComponentB()  
        };  
  
        Console.WriteLine("The client code works with all visitors via the  
base Visitor interface:");  
        var visitor1 = new ConcreteVisitor1();  
        Client.ClientCode(components, visitor1);  
    }  
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

```
Console.WriteLine();

Console.WriteLine("It allows the same client code to work with
different types of visitors:");
    var visitor2 = new ConcreteVisitor2();
    Client.ClientCode(components, visitor2);
}
}
```

## INVESTIGACIÓN N° 2 PATRONES DE DISEÑO Y SOLID

### BIBLIOGRAFÍA

Refactoring Guru. (s. f.). *Patrones de diseño en C#*. Recuperado 30 de junio de 2022, de <https://refactoring.guru/es/design-patterns/csharp>

Naidu, D. (2021, 16 marzo). *SOLID Principles In C#*. C#Corner. Recuperado 4 de julio de 2022, de <https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/>

*Principios SOLID de Uncle Bob en ESPAÑOL y con C#!!! - Introducción*. (2021, 4 mayo).

YouTube. Recuperado 4 de julio de 2022, de

<https://www.youtube.com/watch?v=P9yOfeD8Gv8&list=PL-mmLKprTygvd6NdeMEo7tGF9lPz3dh8b>

*Singleton en C# .Net / Patrones de diseño / design patterns / #1*. (2018, 7 agosto). YouTube.

Recuperado 4 de julio de 2022, de

[https://www.youtube.com/watch?v=K902i\\_tsXI0&list=PLWYKfSbdsjJiaXNIW1OYhRnStsyGvr6t](https://www.youtube.com/watch?v=K902i_tsXI0&list=PLWYKfSbdsjJiaXNIW1OYhRnStsyGvr6t)