**Universidad de la Cuenca del Plata**

****

**Ingeniería en Sistemas de Información**

**Ingeniería en Software II**

**Profesor: Kutz, Rene Gabriel**

**Trabajo Práctico Nro 2**

**Integrantes**

Ø Lulusis Mauricio

Ø Martinez Alejandro

índice

[Introducción 3](#_Toc166185101)

[Desarrollo 4](#_Toc166185102)

[Patron builder 4](#_Toc166185103)

[¿Cuándo usar el patrón Builder? 4](#_Toc166185104)

[Diagrama de clases 5](#_Toc166185105)

[Participantes 5](#_Toc166185106)

[Colaboraciones 6](#_Toc166185107)

[Diagrama de secuencia 6](#_Toc166185108)

[Ventajas y desventajas 7](#_Toc166185109)

[Ventajas 7](#_Toc166185110)

[Desventajas 7](#_Toc166185111)

[Usos conocidos en C# con .NET 9](#_Toc166185112)

[Patron Mediator con arquitectura CQRS 9](#_Toc166185113)

[¿Qué es CQRS? 9](#_Toc166185114)

[¿Qué problema se intenta resolver? 10](#_Toc166185115)

[Patrón mediator 11](#_Toc166185116)

[MediatR: Facilitando la Implementación de CQRS y el Patrón del Mediador 12](#_Toc166185117)

[Implementando CQRS en ASP.NET Core 12](#_Toc166185118)

[Domain 12](#_Toc166185119)

[Infrastructure → Persistence 12](#_Toc166185120)

[Features → Products → Queries 13](#_Toc166185121)

[Features → Products → Commands 15](#_Toc166185122)

[Controladores 15](#_Toc166185123)

[Dimensiones y Desafíos de la Confiabilidad en Sistemas de Software 18](#_Toc166185124)

[Disponibilidad y fiabilidad 19](#_Toc166185125)

[Protección 19](#_Toc166185126)

[Seguridad 20](#_Toc166185127)

[Conclusión 22](#_Toc166185128)

[Bibliografía 22](#_Toc166185129)

**Ejes Temáticos :** Nro 2 Diseño Orientado a Objetos (Patrones de Software)

Nro 3 Diseño Ul/interfaz Centrado en el Usuario

**Tema**: Patrones de Diseño Orientado de Objetos

**Objetivos:** investigar, analizar, resumir, ejemplificar y presentar dos patrones de diseño de software.

**Actividades:** Estableceremos en clases una lista de Patrones de Diseño de Software.

Los alumnos se dividirán en grupos de dos alumnos con dos patrones de diseño. Seleccionar el material bibliográfico de la cátedra, o presentados por los alumnos como referencia.

* En un segundo punto investigar sobre la Especificación de confiabilidad y seguridad en el Capítulo 11 del Libro Ingeniería el software (Ian Sommerville o PRESSMAN) novena Edición teniendo en cuenta los siguientes ítems:

12.1 Especificación de requerimientos dirigida por riesgos

12.2 Especificación de protección

12.3 Especificación de fiabilidad

12.4 Especificación de seguridad

12.5 Especificación formal

Leer comprensiva del material seleccionado. Elaborar la presentación y generar un informe escrito en formato presentación y ejemplo aplicación sobre los conceptos. Socialización y presentación de la producción realizada.

**Tipo de actividad:** Grupal (grupos de dos alumnos)

**Modalidad:** Investigación grupal, desarrollo de ejemplo y presentación oral. Entrega de los artefactos de la investigación en un repositorio de la materia en GitHub.

Tiempo: 2 Semanas

# **Introducción**

En este trabajo práctico, exploraremos dos patrones de diseño fundamentales: el patrón Mediator y el patrón Builder. Estos patrones son esenciales en el desarrollo de software, ofreciendo soluciones efectivas para problemas comunes que enfrentan los desarrolladores.

El patrón **Mediator** actúa como intermediario entre objetos, facilitando su comunicación y promoviendo la modularidad en el diseño al centralizar el control de las interacciones. Permite que los objetos se comuniquen de manera indirecta a través de un mediador, reduciendo las dependencias entre ellos y simplificando la lógica de interacción.

Por otro lado, el patrón **Builder** separa la construcción de objetos complejos de su representación, permitiendo la creación paso a paso de diferentes tipos y configuraciones de objetos. Este enfoque modular facilita la creación de objetos complejos, proporcionando un control preciso sobre cada aspecto de su construcción.

Además, en este trabajo, vamos a incorporar conceptos clave de confiabilidad y seguridad del libro de Summerville. La confiabilidad y la seguridad son atributos fundamentales en el desarrollo de software, que abordan la necesidad de que los sistemas sean consistentes, estables y protegidos contra posibles amenazas externas.

Al analizar en detalle la estructura, el propósito y la implementación de cada patrón, también exploraremos cómo estos conceptos se relacionan con la confiabilidad y la seguridad del software. Examincaremos las ventajas y desventajas de cada patrón en términos de confiabilidad y seguridad, así como sus aplicaciones prácticas en el desarrollo de software moderno.

Con una comprensión más profunda de estos patrones y conceptos, los desarrolladores estarán mejor equipados para abordar los desafíos del diseño de software, creando sistemas más robustos, seguros y confiables que satisfagan las necesidades del usuario y del negocio.

# **Desarrollo**

# **Patron builder**

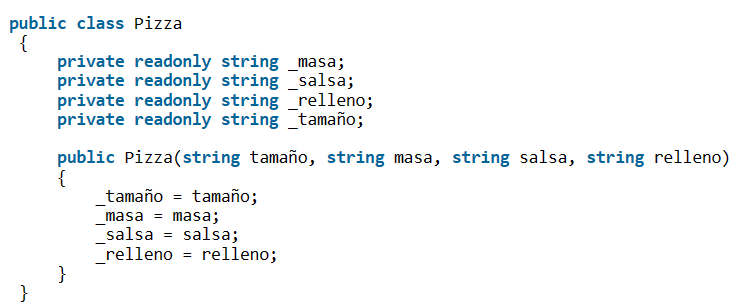
El patrón de diseño Builder separa la creación de un objeto complejo de su representación de modo que el mismo proceso de construcción pueda crear representaciones diferentes.

Básicamente significa que, en lugar de implementar la creación de un objeto en el constructor de la clase, va a haber otras clases encargadas de crear el objeto y asignarles las propiedades iniciales. Cada una de estas "otras" clases será un builder. Y cada uno de estos builders será capaz de crear un objeto nuevo con ciertas características.

Dentro de la clasificación de los patrones, Builder es considerado un patrón creacional.

# **¿Cuándo usar el patrón Builder?**

Supongamos que necesitas construir un mismo objeto complejo muchas veces y además con diferentes configuraciones, por ejemplo, una pizza. Cada pizza de la carta va a tener características concretas y diferenciadas de las demás:



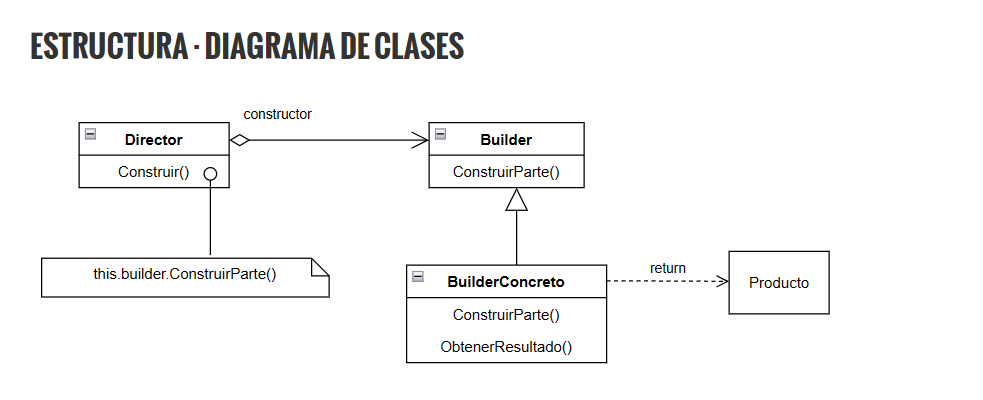
Si no queremos tener que recordar cada vez que creemos una pizza "CuatroQuesos" su tamaño, masa, salsa y relleno, podríamos utilizar el patrón Builder y olvidarnos de sus parámetros.

Para cada pizza del menú necesitaríamos un builder.

A modo orientativo, si tienes dudas sobre cuándo usar el patrón Builder fíjate en el número de parámetros que necesitas para crear el producto final. Si es cuatro o superior te podría interesar, por debajo no merece la pena.

Nota: he puesto un ejemplo sencillo, hay que suponer que las propiedades de la Pizza en lugar de cuatro strings, podrían ser cuatro clases complejas como Masa(), Salsa(), Relleno() y Tamaño().

# **Diagrama de clases**



# **Participantes**

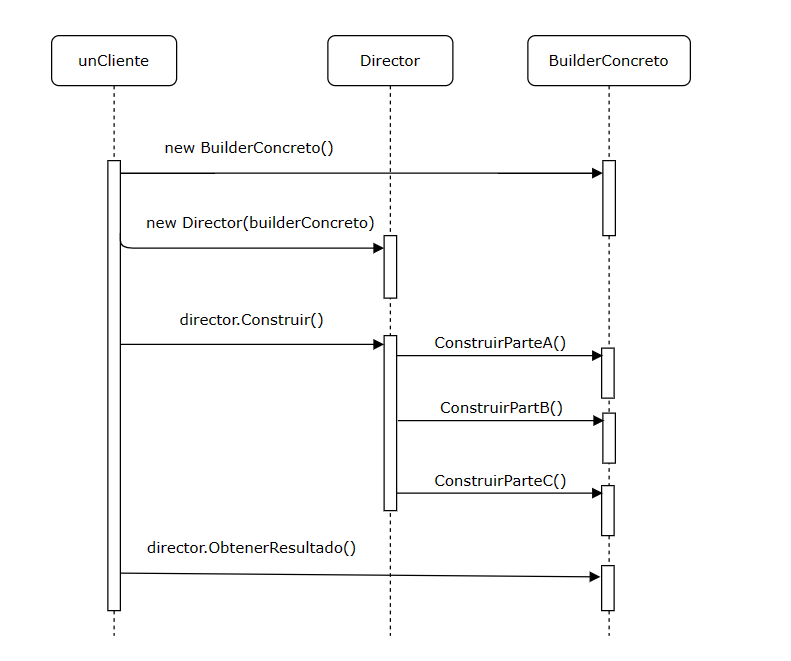
La estructura y los participantes empezarán a tener sentido cuando los veamos aplicados más adelante en el código de ejemplo:

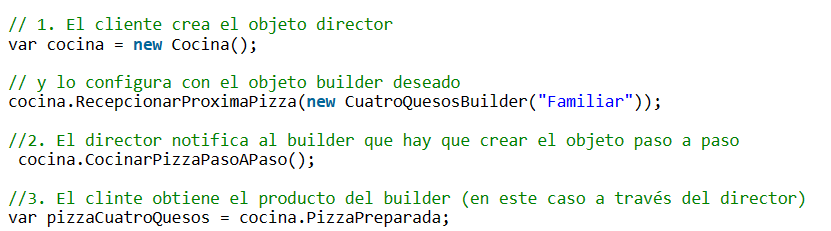
* **Builder**
  + clase abstracta para crear los objetos finales producto.
  + ejemplo en c#: crear una clase abstracta**BuilderPizza**
* **BuilderConcreto**
  + selecciona un nombre que defina la representación del objeto a crear.
  + implementa la clase abstracta *Builder*
  + cconstruye y ensambla las partes del producto.
  + ejemplo en c#: clase **BuilderPizzaHawaiana**: BuilderPizza
* **Director**
  + construye un objeto usando la claseBuilder.
  + ejemplo en c#: clase **PreparadorPizza o Cocinero o Cocina**
* **Producto**
  + representa el objeto complejo en construcción. El ConstructorConcreto construye la representación interna del producto y define el proceso de ensamblaje.
  + incluye las clases que definen sus partes constituyentes, incluyendo interfaces para ensamblar partes en el resultado final.
  + ejemplo en c#: objeto creado **PizzaCuatroQuesos** y **PizzaHawaiana**

# **Colaboraciones**

1. El cliente crea el objeto Director (Cocina o PreparadorPizzas) y lo configura con el objeto builder deseado
2. El Director notifica al builder cada vez que hay que construir una parte de un producto.
3. El Builder maneja las peticiones del director y las añade al producto.
4. El cliente obtiene el producto del builder.

# **Diagrama de secuencia**





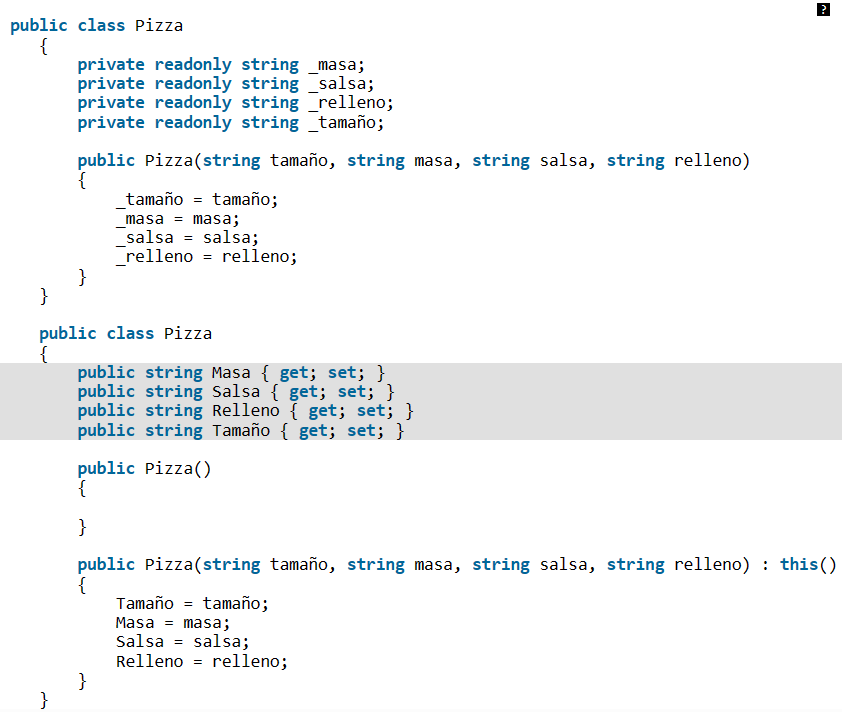
# **Ventajas y desventajas**

# **Ventajas**

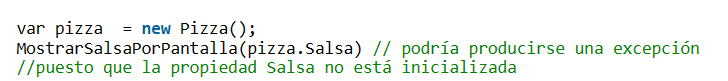
* **Permite variar la representación interna de un producto**. Esconde los detalles de la construcción del producto y cómo se ensambla. Lo que hay que hacer para cambiar la representación interna del producto es definir un nuevo tipo de builder.
* **Encapsula el código de construcción y de representación**. Aumenta la modularidad. Los clientes no necesitan saber nada sobre las clases que definen la estructura interna del producto.
* **Proporciona un control más explícito sobre el proceso de construcción**. A diferencia de los demás patrones de creación, que construyen los productos una sola vez, el patrón Builder construye el producto paso a paso, bajo el control del Director.
* **El código es más mantenible si el número de parametros para crear el objeto es mayor que cuatro.**

# **Desventajas**

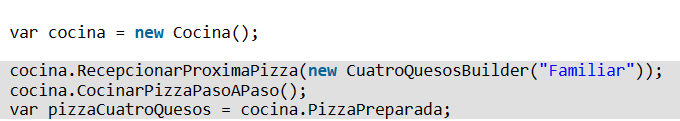
* Hay que crear un BuilderConrecto para cada representación de un producto, lo que **puede acabar con multitud de clases**.
* **Las clases productos que construye el Builder deben ser mutables**.  
    
  Por ejemplo, la clase pizza que habíamos definido era inmutable puesto que una vez se ha creado ya no se puede modificar. Pues si  vas a utilizar un builder esto ya no va a ser posible.



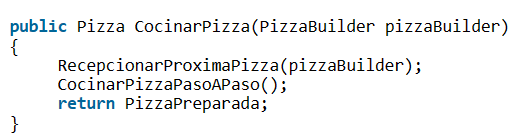
**No garantiza que los campos de la clase producto estén inicializados**.   
  
Efectivamente, en la nueva versión de pizza necesitamos un constructor vacío, con lo que ya no se garantiza que al tener un objeto Pizza, todos sus campos estén inicializados.  
  
Un cliente podría hacer lo siguiente:



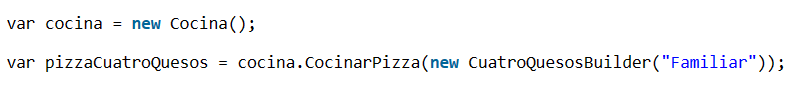
* **No es muy compatible con la inyección de dependencias**. A mi entender, y que alguien me corrija, esto no es una desventaja. No creo que sea una buena práctica inyectar este patrón en clases servicio o de aplicación. Es mejor utilizarlo tal cual para crear objetos de dominio.
* **Requiere emplear más código de lo que sería deseable**. En el ejemplo que he puesto no se da, pero si el objeto final tiene muchas propiedades es posible que acabes duplicando parte del código. No todos los builder necesitarán aplicar cada uno de los pasos, y si eso pasa duplicaremos métodos vacíos.  
    
  Por ejemplo, la PizzaCalzone requerirá un paso más que podríamos llamar PasoDoblarPizza(). Este nuevo paso lo deberán implementar todos los builders, pero solo tendrá sentido en el CalzoneBuilder.
* **Aplicado al pie de la letra el patrón está "acoplado temporalmente"**. Es decir, los métodos se deben ejecutar en un orden concreto y si no es así, el resultado no será el deseado:



Si primero ejecutase CocinarPizzaPasoAPaso() en lugar de RecepcionarProximaPizza() el código sería incorrecto.  
  
Pero esto es fácilmente solucionable se encapsulamos estas tres sentencias dentro de una única función de Cocina



Y llamamos directamente a este método así:



# **Usos conocidos en C# con .NET**

En .net framework hay pocos usos conocidos del patrón Builder.

* [SqlConnectionStringBuilder](https://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnectionstringbuilder(v=vs.90).aspx): se usa para poder añadir partes de la cadena de conexion por separado.
* [UriBuilder](https://msdn.microsoft.com/es-es/library/system.uribuilder(v=vs.110).aspx): parecido al anterior, se usa para crear url's indicando partes de la misma, puerto, *host*, etc.

Y con .net core hay un ejemplo de builder que siempre nos vamos a encontrar en una aplicación web

* [IWebHostBuilder](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.hosting.iwebhostbuilder?view=aspnetcore-2.1): la manera de utilizar el builder no es definida del patrón original, pero el concepto es el mismo. A través del builder se puede obtener un [IWebHost](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.hosting.iwebhost?view=aspnetcore-2.1) configurado adecuadamente y que puede ser ejecutado.

# **Patron Mediator con arquitectura CQRS**

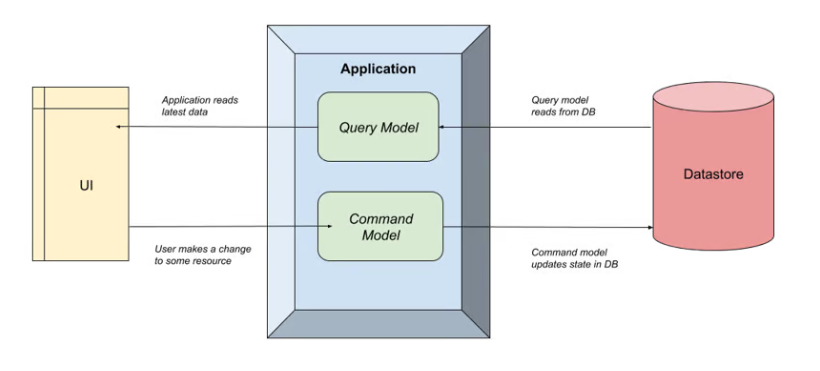
# **¿Qué es CQRS?**

**Command Query Responsibility Segregation**, o CQRS en sus siglas, es un patrón de diseño que ha ganado popularidad en los últimos años. La idea fundamental detrás de CQRS es dividir lógicamente el flujo de nuestra aplicación en dos corrientes principales:

* **Comandos (Commands)**: Estos son responsables de modificar el estado del dominio y no son idempotentes.
* **Consultas (Queries)**: Se encargan de obtener información del estado del dominio y representan operaciones idempotentes.

Si pensamos en un CRUD (Crear, Leer, Actualizar y Borrar), los comandos corresponden a las operaciones Crear(Create), Actualizar (Update) y Borrar (Delete), mientras que las consultas se relacionan con la operación **Leer (Read).**

La siguiente imagen ilustra cómo funciona esta separación de responsabilidades:



Como se puede apreciar, la aplicación se divide en dos conceptos fundamentales: comandos (commands) y consultas (queries). Aunque la idea principal de CQRS también involucra dividir el almacén de datos en dos (uno maestro y otro replicado) para la lectura y la escritura, la noción de dividirlo de manera lógica funciona de manera eficiente en el diseño del sistema, incluso si se utiliza una única base de datos (aunque también es factible implementar el uso de bases de datos físicamente separadas).

# **¿Qué problema se intenta resolver?**

El enfoque tradicional para diseñar aplicaciones en "n-capas" generalmente implica dividirlas en tres capas: Interfaz de usuario, Lógica de Negocio y Almacenamiento de Datos.

Al principio, esto puede no parecer un problema, pero surgen dificultades en cuanto al mantenimiento y la falta de flexibilidad para agregar nuevas características, depurar el código y otros desafíos.

En sistemas de "n-capas," a menudo terminamos con enormes repositorios que contienen todas las operaciones que se pueden realizar en una entidad. También solemos tener servicios que se vuelven cada vez más grandes con el tiempo.

La clave aquí es la segregación de responsabilidades en el mantenimiento de un sistema. Modificar una función no debería afectar a áreas completamente diferentes. Imagina tener una clase llamada ProductosService que contiene todas las operaciones relacionadas con los productos. Esto puede convertirse en un problema cuando el sistema crece, se unen nuevos miembros al equipo y la curva de aprendizaje es empinada. Cuando un desarrollador junior necesita modificar una función, es natural tener miedo de romper algo, ya que toda la funcionalidad está fuertemente acoplada en el servicio o repositorio.

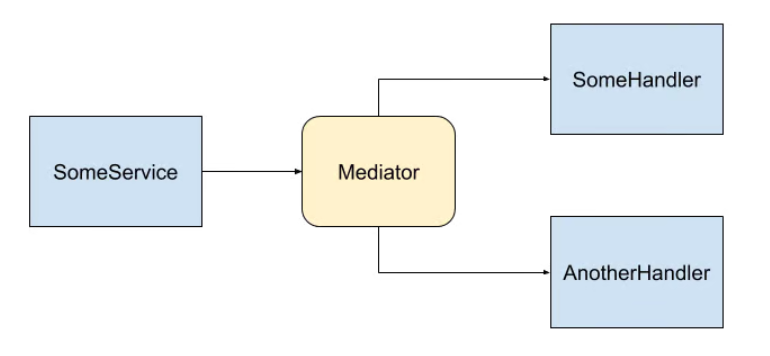
La separación en Queries y Commands y, aún mejor, en [Segmentos Verticales](https://dev.to/isaacojeda/vertical-slice-architecture-36ng) (Características) permite mantener un código organizado. Agregar nuevas funcionalidades simplemente significa agregar más Queries o Commands en lugar de modificar servicios o repositorios gigantes.

Además, esta estructura facilita las pruebas. Un servicio puede tener dependencias para diversas operaciones sobre una entidad, lo que significa que necesitarás una serie de "mocks" para probar una función específica. En cambio, un Command solo incluye lo que necesita para funcionar, sin afectar a otras funcionalidades. Cada Command se encuentra encapsulado, y modificar uno no debería afectar a otros.

Por supuesto, es importante saber cuándo refactorizar. Si un Command realiza una tarea que también es realizada por otro Command, es hora de considerar otros patrones como Strategy o decoradores y realizar una refactorización. Además, es esencial encontrar un equilibrio entre no repetir código (DRY - Don't Repeat Yourself) y cumplir con el principio de Responsabilidad Única (aunque puede ser un desafío, con el tiempo te acostumbrarás).

# **Patrón mediator**

El patrón del mediador se trata simplemente de definir un objeto que encapsula cómo otros objetos interactúan entre sí. En lugar de tener dos o más objetos que dependen directamente de otros objetos, estos objetos toman dependencia directa de un "mediador", y este mediador se encarga de gestionar las interacciones entre ellos:



Como se muestra en el diagrama, SomeService envía un mensaje al mediador, y el mediador a su vez llama a otros servicios para que realicen acciones basadas en el mensaje recibido. SomeService no necesita saber nada acerca de los otros servicios que actúan según su solicitud; solo comunica al mediador lo que necesita que se haga.

La razón por la que el patrón del mediador es tan útil es la misma razón por la que utilizamos patrones como la [Inversión de Control (IoC)](https://code-maze.com/dependency-injection-aspnet/). Nos permite desacoplar por completo componentes, pero aún así permite que interactúen entre sí. Cuanto menos tenga que preocuparse un componente para funcionar, más sencillo será desarrollarlo, mantenerlo y probarlo.

# **MediatR: Facilitando la Implementación de CQRS y el Patrón del Mediador**

MediatR es una implementación del patrón mediador que ocurre completamente en el mismo proceso de la aplicación (*in-process*), y es una herramienta fundamental para crear sistemas basados en CQRS. Toda la comunicación entre el usuario y la capa de persistencia se gestiona a través de MediatR.

Es importante destacar que MediatR se ejecuta dentro del mismo proceso (*in-process*), lo que es una limitación clave. Dado que .NET maneja todas las interacciones entre objetos en el mismo proceso, MediatR no es apropiado si deseamos separar los Queries y Commands en aplicaciones distintas (es decir, si buscamos sistemas completamente independientes).

# **Implementando CQRS en ASP.NET Core**

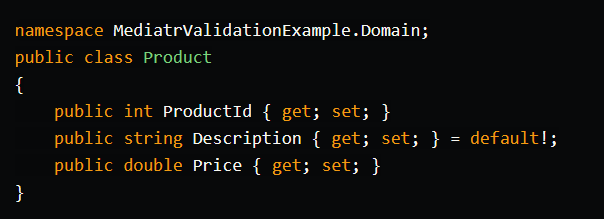
La idea detrás de la implementación de CQRS en ASP.NET Core, específicamente en una Web API, es delegar la responsabilidad de procesar cada solicitud (*Request*) a un "Manejador" (*Handler*) en lugar de hacerlo en el controlador, como se mencionó anteriormente.

¿Por qué hacer esto? Hay varias razones, pero una de las más importantes es asegurarse de que todo el procesamiento de las solicitudes en la API no dependa directamente de los controladores. En su lugar, delegamos esta responsabilidad a una capa en la "Aplicación Central" (siguiendo los principios de Clean Architecture o Vertical Slices).

En .NET 7, es posible que incluso comencemos a utilizar Minimal APIs debido a la mejora de rendimiento. Los controladores ya no realizan tareas de procesamiento, solo reciben la solicitud y pueden hacer estos cambios sin inconvenientes.

# **Domain**

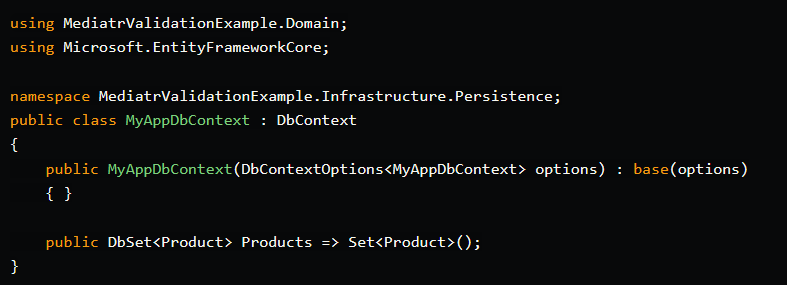
Utilizaremos una clase Product para este ejemplo:



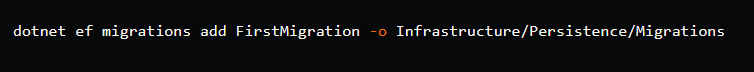
¡Es importante notar que aquí usamos el operador default! simplemente para inicializar un string con un valor predeterminado y decirle al compilador que nunca será null. Es importante mencionar que esto es técnicamente incorrecto, ya que el valor predeterminado de un string es null.

# **Infrastructure → Persistence**

Utilizamos Entity Framework Core para la persistencia



Para crear la base de datos y su migración inicial, ejecutamos los siguientes comandos:

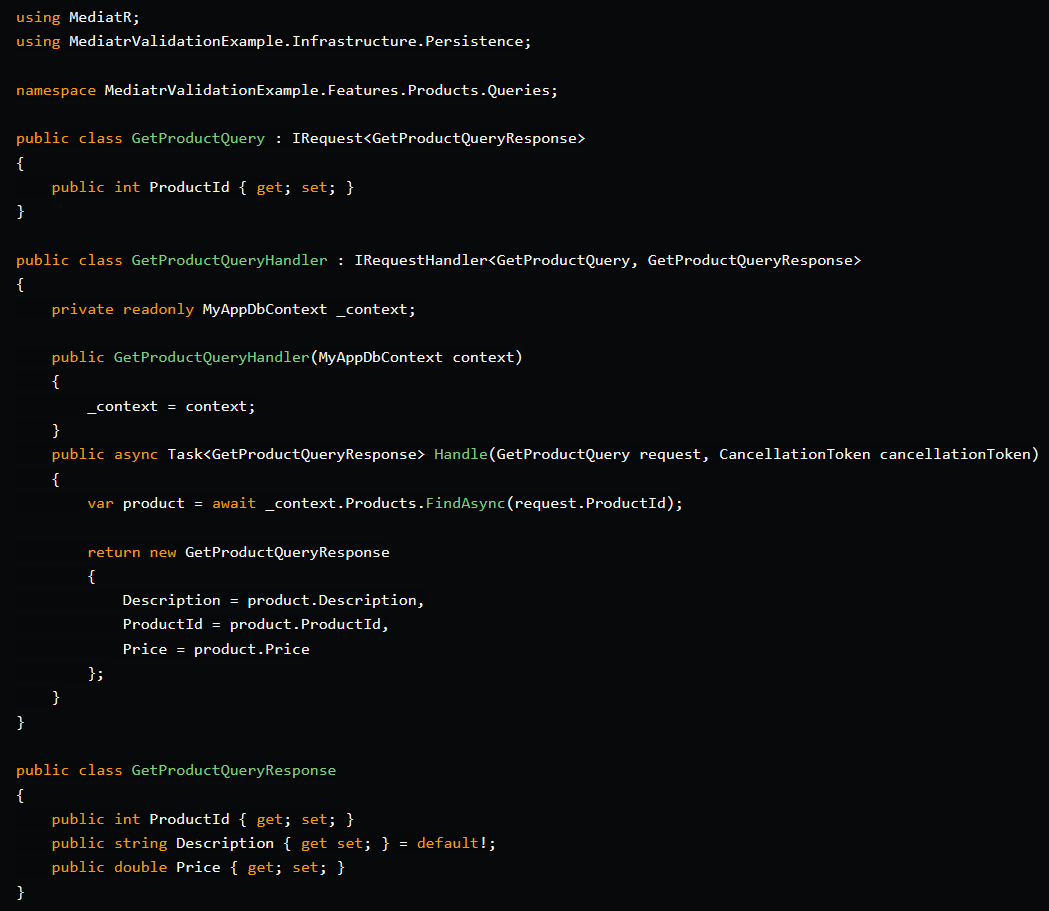


****

# **Features → Products → Queries**

Esta sección representa el núcleo de la aplicación, donde ubicaremos las consultas y comandos necesarios para la Web API. Empecemos con un ejemplo sencillo de cómo consultar productos.

En resumen, la idea es colocar todo lo necesario en un solo archivo (la solicitud, el manejador, los validadores, los mapeadores, los modelos, etc.). Como menciono en el artículo, si es necesario, se puede refactorizar



Lo más importante aquí es prestar atención a las interfaces IRequest<T> y IRequestHandler<T>.

IRequest<T> es el mensaje que especifica la tarea a realizar, solicitado por **SomeService** y dirigido a uno o más **manejadores** (como se muestra en la imagen anterior).

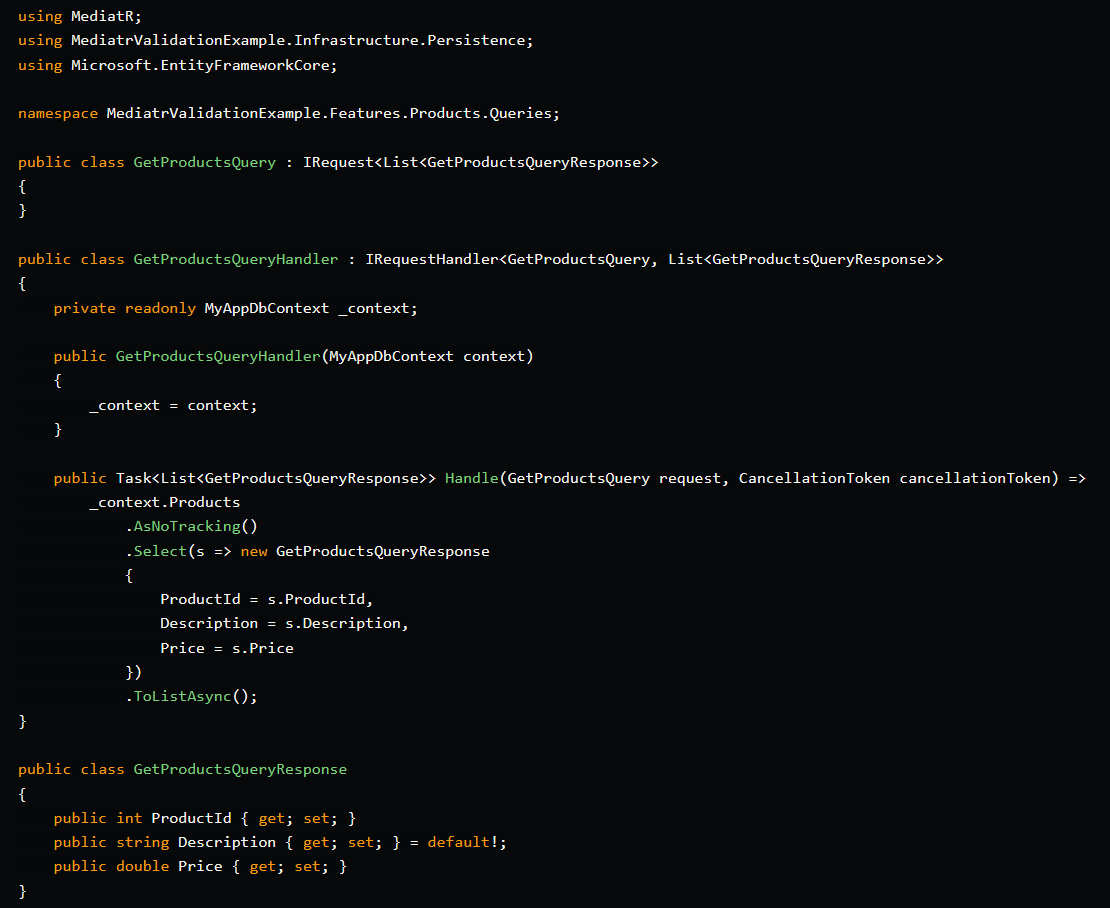
En otras palabras, el mediador tomará la IRequest<T> y la enviará a los manejadores registrados. Estos manejadores saben qué mensajes pueden recibir y cómo llevar a cabo la tarea.

En este caso, GetProductQuery es una IRequest<T> que representa la búsqueda de un producto. IRequest<T> incluye un tipo genérico para especificar el tipo de objeto que se devolverá, ya que, en este caso, estamos realizando una consulta y obteniendo el estado del dominio.

En otros tiempos, habríamos creado un ProductsService o un ProductsRepository con un método GetById. Sin embargo, en este enfoque, la clase representa la operación a realizar, en lugar de un método adicional en una clase con múltiples métodos.

Esto es lo que me encanta de este patrón; sí, tendremos muchos archivos y carpetas, pero serán archivos pequeños y fáciles de buscar gracias a los potentes editores de texto y entornos de desarrollo integrados (IDEs).

El GetProductQueryHandler es el manejador del mismo Query que definimos anteriormente. Dado que ambos están en el mismo archivo, podríamos decir que el Request y el Handler están acoplados entre sí, pero aislados del resto del código. Agregar funcionalidad o probarla simplemente implica trabajar con lo que se encuentra en este archivo y nada más.

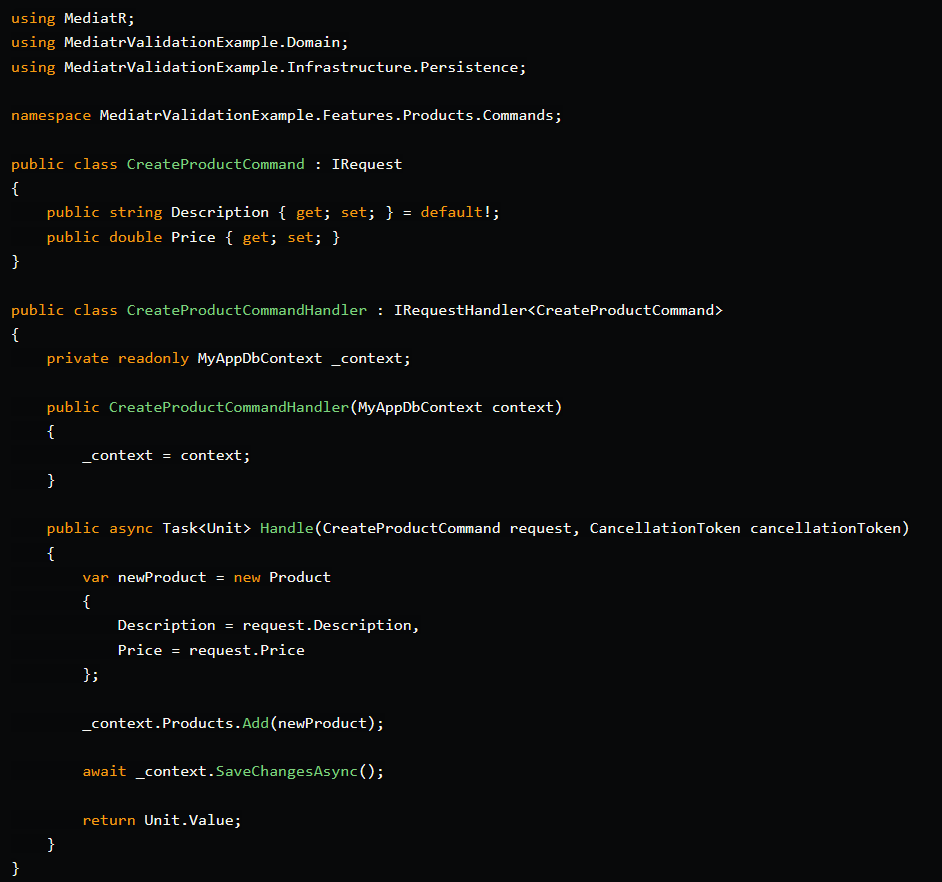


En este otro ejemplo, la interfaz IRequest<T> está vacía, pero si quisiéramos buscar productos, agregar paginación, ordenación, etc., todo se haría en esta clase GetProductsQuery, ya que representa la solicitud que recibe la API (lo veremos en el controlador).

Todos los Queries deben incluir el método AsNoTracking, ya que se tratan de consultas y no necesitan actualizar el estado de las entidades.

# **Features → Products → Commands**

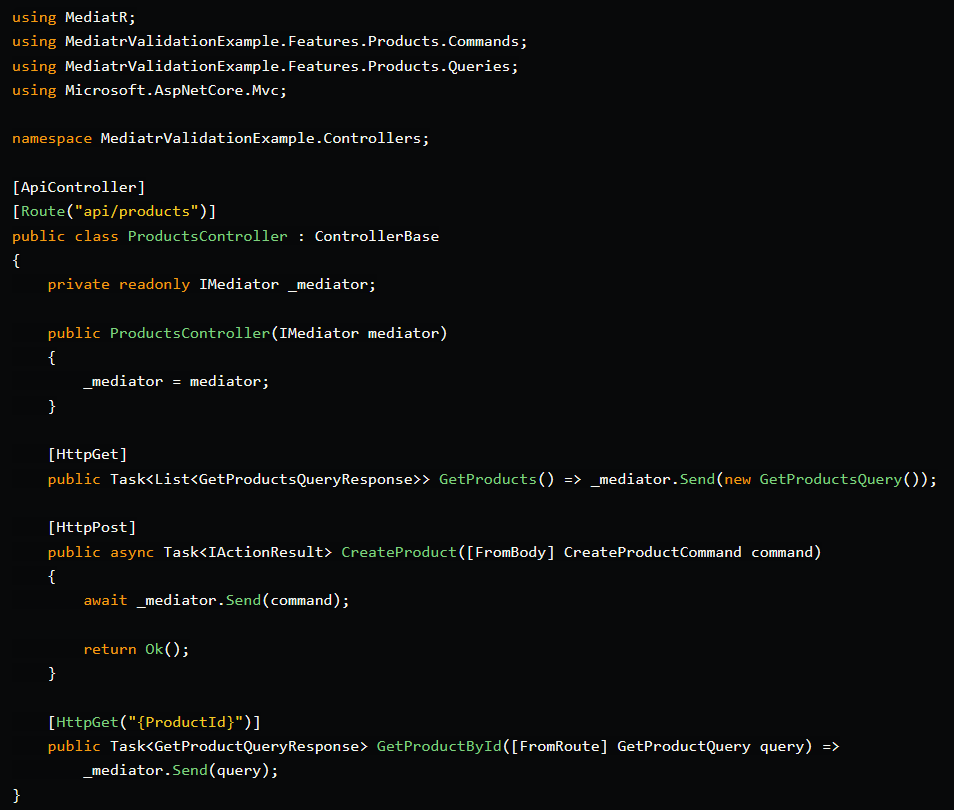
Los comandos son donde finalmente se actualizarán las entidades. En publicaciones posteriores, mostraré cómo agregar validaciones, decoradores y otras funcionalidades que son fáciles de implementar gracias a otras bibliotecas como **FluentValidation** y **MediatR**, que ya estamos utilizando.



Aquí, la única información que necesitamos del request son el nombre del producto que deseamos registrar y su precio. Seguimos utilizando la interfaz de MediatR IRequest, aunque en este caso no es necesario un tipo genérico, ya que los comandos generalmente no devuelven información.

# **Controladores**

Dentro de los controladores, finalmente utilizaremos el mediador. Así es como se ve en la práctica:

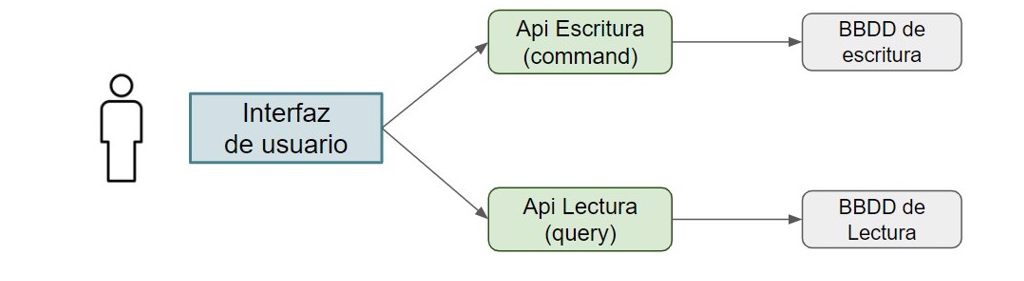


A través de la inyección de dependencias, solicitamos el mediador utilizando la interfaz IMediator. Una vez que inicializamos la solicitud IRequest correspondiente, simplemente la enviamos al mediador, que se encargará de determinar qué manejadores deben ejecutar la solicitud.

En CreateProduct, la IRequest (también conocida como comando) se recibe desde el cuerpo de la solicitud (como una clase POCO), lo que permite recibir y serializarla sin ningún problema.

En GetProductById, la IRequest (también conocida como consulta) se obtiene del segmento de la URL. Aquí es importante que el nombre en el segmento coincida con la propiedad correspondiente para que hagan juego.

En GetProducts, inicializamos la solicitud manualmente, ya que no estamos recibiendo ningún dato adicional en la solicitud, pero podría utilizarse [FromQuery] para recibir parámetros adicionales.



2) **Confiabilidad y seguridad**

**Importancia de la Confiabilidad en los Sistemas de Software**

En la medida en que los sistemas de cómputo se insertan cada vez más en las operaciones empresariales y en la vida cotidiana de las personas, la preocupación por las fallas del sistema y del software se vuelve cada vez más prominente. Desde una falla en el servidor de una empresa de comercio electrónico que podría resultar en una pérdida significativa de ingresos y clientes, hasta un error en el software de un sistema de control embebido en un automóvil que podría provocar costosas devoluciones y, en casos extremos, contribuir a accidentes, las implicaciones de tales fallos son vastas y pueden afectar a múltiples partes interesadas.

Para abordar estas preocupaciones, se ha propuesto el término "confiabilidad" por Laprie, que abarca una serie de atributos críticos para la operación segura y efectiva de los sistemas de software, como la disponibilidad, fiabilidad, protección y seguridad. Esta confiabilidad es esencial debido a varias razones fundamentales:

Primero, las fallas del sistema pueden tener un impacto masivo, afectando a un gran número de usuarios y poniendo en peligro la continuidad de las operaciones comerciales normales. La disponibilidad del sistema es crucial para mantener la normalidad del negocio y la satisfacción del cliente.

Segundo, los usuarios tienden a rechazar los sistemas que perciben como poco fiables o inseguros, lo que puede resultar en una disminución de la adopción del producto y dañar la reputación de la empresa. La confiabilidad es esencial para generar confianza en los usuarios y garantizar la aceptación y la lealtad del cliente.

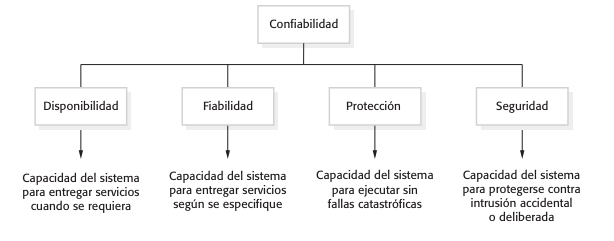
Tercero, las fallas del sistema pueden tener costos financieros significativos, especialmente en aplicaciones críticas como el control de reactores o sistemas de navegación de aeronaves, donde el costo de una falla puede ser mucho mayor que el costo del sistema en sí. Estos costos pueden incluir pérdida de ingresos, gastos de reparación y mantenimiento, así como posibles sanciones regulatorias.

Cuarto, la pérdida de información valiosa debido a fallas del sistema puede ser costosa y perjudicial para una organización. Los datos son un activo crítico para muchas empresas, y el costo de recuperar datos perdidos o dañados puede ser considerable.

Además, es importante reconocer que la confiabilidad del sistema no se limita solo al software, sino que también depende del hardware subyacente y de los procesos operativos. Las fallas en cualquiera de estas áreas pueden tener consecuencias significativas y pueden estar interrelacionadas entre sí. Por lo tanto, los diseñadores y desarrolladores de sistemas confiables deben adoptar un enfoque holístico que considere todos los aspectos del sistema, desde el diseño del hardware y el software hasta la operación del sistema en su conjunto.

# **Dimensiones y Desafíos de la Confiabilidad en Sistemas de Software**

La confiabilidad de un sistema de cómputo es una característica esencial que refleja el grado de confianza que los usuarios tienen en que el sistema funcionará como se espera y sin errores durante su uso normal. Esta confiabilidad es crucial en una variedad de contextos, especialmente en los llamados "sistemas críticos", donde la falla del sistema podría resultar en lesiones, daños ambientales o pérdidas económicas significativas.



La confiabilidad se desglosa en varias dimensiones clave:

1. **Disponibilidad:** Es la probabilidad de que un sistema esté disponible y funcione correctamente en un momento dado, brindando servicios útiles a los usuarios.
2. **Fiabilidad:** Es la probabilidad de que un sistema brinde servicios correctamente durante un período de tiempo específico, según lo esperado por el usuario.
3. **Protección:** Es la evaluación de cuán probable es que un sistema cause daño a las personas o al medio ambiente.
4. **Seguridad:** Es la probabilidad de que un sistema pueda resistir intrusiones accidentales o deliberadas.

Estas propiedades de confiabilidad son complejas y pueden descomponerse en atributos más simples, como integridad, confidencialidad y exactitud.

Además de estas dimensiones principales, existen otros aspectos importantes a considerar:

* **Reparabilidad:** La capacidad del sistema para diagnosticar y corregir rápidamente fallos.
* **Mantenibilidad:** La facilidad con la que el sistema puede adaptarse a nuevos requisitos y cambios sin introducir errores adicionales.
* **Supervivencia:** La capacidad del sistema para continuar proporcionando servicios incluso cuando está bajo ataque.
* **Tolerancia para el error:** La capacidad del sistema para detectar y corregir errores de entrada del usuario.

El desarrollo de software confiable requiere un enfoque exhaustivo que aborde la prevención de errores, la detección de fallas, la protección contra ataques externos y la configuración adecuada del entorno operativo. Además, dado que ningún software es perfecto, es crucial incorporar mecanismos de recuperación para restaurar el servicio normal del sistema lo más rápido posible después de una falla.

Sin embargo, mejorar la confiabilidad del sistema conlleva costos adicionales significativos, tanto en términos de diseño, implementación y validación, así como en pruebas exhaustivas para demostrar su confiabilidad. A medida que un sistema se vuelve más confiable, los costos asociados con su mejora también aumentan, lo que puede plantear desafíos financieros y logísticos para las organizaciones.

# **Disponibilidad y fiabilidad**

La disponibilidad y la fiabilidad son dos propiedades cruciales en la evaluación de la confiabilidad de un sistema de computación. La disponibilidad se refiere a la probabilidad de que el sistema esté operativo y brinde servicios cuando los usuarios lo soliciten, mientras que la fiabilidad se refiere a la probabilidad de que el sistema brinde correctamente los servicios según lo especificado durante un período determinado.

La relación entre disponibilidad y fiabilidad es estrecha y se expresa a menudo en términos numéricos. Por ejemplo, si la disponibilidad de un sistema es del 99.9%, esto significa que está disponible el 99.9% del tiempo. Del mismo modo, si la tasa de ocurrencia de fallas de un sistema es de 0.002, significa que, en promedio, 2 de cada 1,000 entradas causan fallas.

La importancia relativa de disponibilidad y fiabilidad puede variar según el tipo de sistema. Por ejemplo, en un conmutador telefónico, donde los usuarios esperan un servicio continuo, la disponibilidad puede ser más importante que la fiabilidad. En otros casos, donde las fallas del sistema tienen consecuencias graves, la fiabilidad puede ser prioritaria sobre la disponibilidad.

Es importante tener en cuenta que las definiciones estándar de disponibilidad y fiabilidad pueden no capturar completamente la percepción de los usuarios sobre la confiabilidad de un sistema. Por ejemplo, la gravedad de una falla y las consecuencias de la indisponibilidad pueden influir en la percepción de los usuarios sobre la confiabilidad del sistema.

Las fallas en el desarrollo del sistema pueden conducir a errores del sistema, que a su vez pueden resultar en caídas del sistema. Sin embargo, no todas las fallas en el desarrollo necesariamente conducen a caídas del sistema, ya que los usuarios pueden adaptar su comportamiento para evitar ciertas características propensas a fallar.

Para mejorar la fiabilidad de un sistema, se pueden aplicar diversas técnicas, como la prevención de fallas en el desarrollo, la detección y eliminación de fallas en el desarrollo y la tolerancia a fallas en el desarrollo. Estas técnicas ayudan a minimizar la probabilidad de fallas y a mitigar su impacto en caso de que ocurran.

# Protección

Los sistemas críticos para la protección son aquellos en los que es esencial que la operación sea segura en todo momento, sin causar daño a las personas o al entorno, incluso en caso de falla. Ejemplos incluyen sistemas de control y monitorización en aeronaves, sistemas de control de procesos en plantas químicas y farmacéuticas, y sistemas de control de automotores.

El control de hardware es más fácil de implementar y analizar que el control de software en estos sistemas, pero la complejidad creciente ha hecho necesario el control de software para manejar sensores y actuadores de manera más sofisticada.

El software crítico para la protección se divide en dos clases:

1. **Software primario crítico para la protección**: Controla directamente el sistema y su mal funcionamiento podría resultar en lesiones o daños.
2. **Software secundario crítico para la protección**: Podría influir indirectamente en lesiones o daños, por ejemplo, a través de errores de diseño.

La fiabilidad y la seguridad del sistema están relacionadas, pero un sistema fiable no siempre es seguro, y viceversa. Esto se debe a diversas razones, como la presencia de fallas no detectadas en el desarrollo, especificaciones incompletas, mal funcionamiento del hardware y posibles entradas de usuarios que conduzcan a un mal funcionamiento.

Para garantizar la seguridad, es fundamental evitar peligros, detectar y eliminar peligros antes de que ocurran accidentes, y limitar el daño en caso de accidente. Los sistemas complejos a menudo fallan debido a la interacción de múltiples fallas en diferentes partes del sistema, lo que hace imposible anticipar todas las combinaciones posibles de fallas.

Aunque los sistemas controlados por software pueden introducir más formas de falla, también permiten una mejor monitorización y protección, contribuyendo a mejorar la seguridad global del sistema. Sin embargo, es importante mantener un sentido de proporción sobre la seguridad del sistema, ya que es imposible hacer un sistema completamente seguro.

# **Seguridad**

La seguridad en un sistema refleja su capacidad para protegerse de ataques externos, ya sean accidentales o deliberados. En la era de la interconexión, la mayoría de las computadoras están en red y son accesibles desde el exterior, lo que aumenta la posibilidad de ataques como la instalación de virus, el uso no autorizado de servicios del sistema o la modificación no aprobada de datos.

Aunque desconectar un sistema de Internet puede limitar sus problemas de seguridad, en la práctica, los beneficios de la conectividad en red son enormes para sistemas más grandes, como los militares o de comercio electrónico. Sin embargo, esto también aumenta la necesidad de un alto nivel de seguridad.

Los sistemas en red enfrentan tres principales tipos de amenazas a la seguridad: confidencialidad, integridad y disponibilidad. Estas amenazas están interrelacionadas, y un ataque exitoso contra una de ellas puede comprometer las otras dos.

En muchos casos, las vulnerabilidades en los sistemas sociotécnicos se deben a fallas humanas más que a problemas técnicos. Los errores humanos, como contraseñas débiles o malas decisiones de diseño, pueden comprometer la seguridad del sistema.

Los controles para mejorar la seguridad del sistema pueden clasificarse en evitar vulnerabilidades, detectar y neutralizar ataques, y limitar la exposición y recuperación en caso de incidentes.

Sin un nivel razonable de seguridad, no se puede confiar en la disponibilidad, fiabilidad y protección del sistema. Los errores en el desarrollo del sistema pueden crear lagunas de seguridad que los atacantes pueden aprovechar para obtener acceso no autorizado.

Es esencial tener en cuenta la especificación de seguridad junto con los otros aspectos clave del sistema, como la fiabilidad y la protección.

# Conclusión

En conclusión, el patrón Builder y el patrón Mediator son herramientas valiosas en el conjunto de herramientas de un desarrollador de software. El patrón Builder permite la creación paso a paso de objetos complejos, ofreciendo flexibilidad y promoviendo la reutilización del código. Por otro lado, el patrón Mediator simplifica la comunicación entre objetos al introducir un intermediario que centraliza las interacciones, reduciendo el acoplamiento y mejorando la mantenibilidad del sistema.

Ambos patrones proporcionan beneficios significativos en términos de modularidad, flexibilidad y mantenibilidad del código. Al comprender sus principios y aplicaciones, los desarrolladores pueden mejorar la calidad y la eficiencia de sus sistemas, promoviendo un diseño robusto y escalable.

Además, es importante destacar que este informe se benefició de la información proporcionada en el libro de Sommerville, específicamente en el capítulo 11, que aborda temas clave como la confiabilidad y la seguridad en el desarrollo de software. La integración de estos conceptos en nuestro análisis ha enriquecido nuestra comprensión de la importancia de la confiabilidad y la seguridad en el diseño y la implementación de sistemas de software.

Al considerar tanto los aspectos técnicos de los patrones de diseño como los principios fundamentales de confiabilidad y seguridad, los desarrolladores están mejor preparados para enfrentar los desafíos del desarrollo de software, garantizando la creación de sistemas sólidos, seguros y confiables que cumplan con las necesidades de los usuarios y las demandas del mercado.

# **Bibliografía**

[Mediator en C# / Patrones de diseño (refactoring.guru)](https://refactoring.guru/es/design-patterns/mediator/csharp/example)

[Patrón Builder en c#.net (albertcapdevila.net)](https://albertcapdevila.net/patron-builder-csharp-net/)

[Patrón CQRS explicado en 10 minutos (netmentor.es)](https://www.netmentor.es/entrada/patron-cqrs-explicado-10-minutos)

[Desarrollando una API en ASP.NET con CQRS y MediatR: Parte 1 - DEV Community](https://dev.to/isaacojeda/parte-1-cqrs-y-mediatr-implementando-cqrs-en-aspnet-56oe)

Ingenieria de Software 9na edición Sommerville.