****

**Trabajo Práctico N° 2 de Ingeniería de Software II: Investigación y Presentación de Patrones de diseño de Software**

**Alumnos: Atamañuk Lisandro, Benitez Matías, Leal Thiago**

**Profesor: Kutz Gabriel**

**Curso y Carrera: 3°Año Ingeniería en Sistemas de Información**

**Cátedra: Ingeniería de Software II**

**Fecha de entrega: 09/05/2024**

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

**Índice**

**¿Qué son los Patrones de Diseño?..................................................................................... 3**

**Tipos de Patrones de Diseño……………………………………………………………………..4**

1. Singleton……………………………………………………………………………………. **4**
2. Facade (Fachada)...................................................................................................... **7**
3. Strategy……………………………………………………………………………………. **11**

[**12.1 Especificación de requerimientos dirigida por riesgos**](#_2q7sg8c85tdf) 1**4**

[**12.2 Especificación de protección**](#_i05kddtl7spg) **15**

[**12.3 Especificación de fiabilidad**](#_h1zedht6kep0) **15**

[**12.4 Especificación de seguridad**](#_lhoihlu0gfbd) **15**

[**12.5 Especificación formal**](#_h13tzn8kqg05) **15**

**Ejemplos de las especificaciones aplicadas a un caso real……………………………….15**

[1. Especificación de requerimientos dirigida por riesgos:](#_ewtmvagpjgfc) **16**

[2. Especificación de protección:](#_93em0pr0edzd) **16**

[3. Especificación de fiabilidad:](#_91xw1548re3y) **16**

[4. Especificación de seguridad:](#_try1iaj3op2t) **16**

[5. Especificación formal:](#_j5t5kupe69z4) **16**

**¿Qué son los Patrones de Diseño?**

Los **patrones de diseño** son como recetas que nos ayudan a resolver problemas comunes en la creación de software. Han sido desarrollados y perfeccionados por expertos en el campo y nos proporcionan un conjunto de soluciones probadas y documentadas para desafíos que enfrentamos al diseñar programas. Estas soluciones nos otorgan un marco de trabajo para crear sistemas de software sólidos, flexibles y fáciles de mantener.

A continuación se detallan los distintos patrones de diseño que existen y cómo se diferencian entre sí:

1. Patrones Creacionales

- Proporcionan mecanismos para crear objetos.

- Se centran en mejorar la flexibilidad y reutilización del código al ofrecer

formas eficientes de instanciar y configurar objetos.

- Ejemplos incluyen Singleton, Factory Method, Abstract Factory, Builder,

Prototype, entre otros.

2. Patrones Estructurales

- Se enfocan en cómo ensamblar objetos y clases en estructuras más

grandes.

- Ayudan a organizar y relacionar clases y objetos para formar sistemas

complejos, manteniendo la flexibilidad y la eficiencia de estas estructuras.

- Ejemplos incluyen Adapter, Bridge, Composite, Decorator, Facade, Proxy,

entre otros.

3. Patrones de Comportamiento

- Tratan sobre cómo los objetos interactúan, se comunican y colaboran entre

sí.

- Se ocupan de la distribución de responsabilidades y algoritmos dentro de

un sistema, y son esenciales para definir y organizar el comportamiento

dinámico de un programa.

- Ejemplos incluyen Observer, Strategy, Command, Iterator, State, Template

Method, Visitor, entre otros.

En resumen, los patrones creacionales se centran en la creación de objetos, los patrones estructurales en la composición de estructuras más grandes, y los patrones de comportamiento en la interacción y comportamiento dinámico entre objetos.

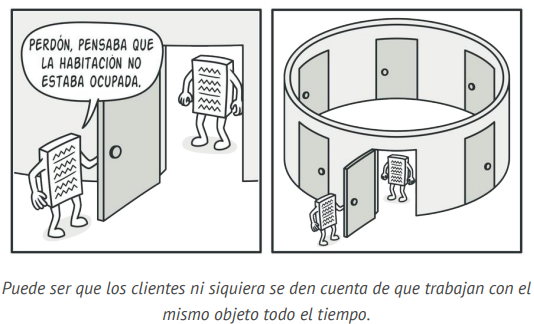
**Tipos de Patrones de Diseño**

**Singleton (Instancia única)**

El patrón de diseño Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

**Problema**

Este patrón resuelve dos problemas al mismo tiempo, pero al hacerlo, vulnera el ***Principio de Responsabilidad Única***, que establece que cada módulo o clase debe tener responsabilidad sobre una sola parte de la funcionalidad proporcionada por el software, y esta responsabilidad debe estar encapsulada en su totalidad por la clase.

Una de las características principales del patrón Singleton es garantizar que una clase tenga una única instancia. Este control sobre la cantidad de instancias puede ser útil para gestionar el acceso a recursos compartidos, como una base de datos o un archivo. En este sentido, una vez que se crea una instancia de la clase, si se intenta crear otra, en lugar de obtener un objeto nuevo, se obtendrá la instancia ya existente. Esta característica no es posible de lograr con un constructor normal, ya que siempre devuelve un nuevo objeto por diseño.

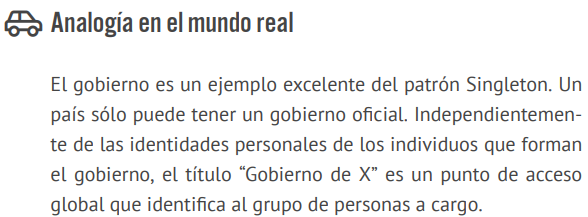
El uso de variables globales para acceder a objetos compartidos puede ser inseguro, ya que cualquier código podría sobrescribir el contenido de esas variables, lo que podría descomponer la aplicación. Al igual que una variable global, el patrón Singleton permite acceder a un objeto desde cualquier parte del programa, pero evita que otro código sobreescriba esa instancia. Además, centraliza el código que resuelve el problema de controlar el acceso a recursos compartidos dentro de una clase, lo que facilita su gestión y mantenimiento, especialmente si otras partes del código ya dependen de esa clase.

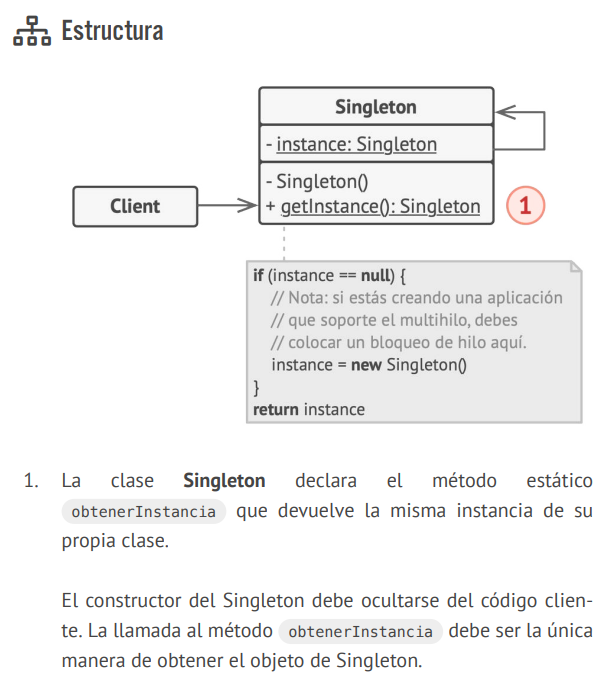
**Solución**

El patrón Singleton garantiza que una clase tenga una única instancia, lo que se logra mediante las siguientes características:

1. Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador new con la clase Singleton. Esto significa que la clase Singleton no puede ser instanciada directamente desde fuera de la clase.

2. Crear un método de creación estático que actúe como constructor. Este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché. Este método estático se puede invocar desde cualquier parte del programa donde se tenga acceso a la clase Singleton.





En síntesis el patrón Singleton tiene sus pros y contras:

**Pros**

1. Garantiza que una clase tenga una única instancia, lo que puede ser útil para controlar el acceso a recursos compartidos.

2. Proporciona un punto de acceso global a la instancia única, lo que facilita su uso desde cualquier parte del programa.

3. El objeto Singleton se inicializa sólo cuando se requiere por primera vez, lo que puede ayudar a mejorar el rendimiento.

**Contras**

1. Vulnera el Principio de Responsabilidad Única al resolver dos problemas al mismo tiempo.

2. Puede enmascarar un mal diseño al permitir que los componentes del programa sepan demasiado unos sobre otros.

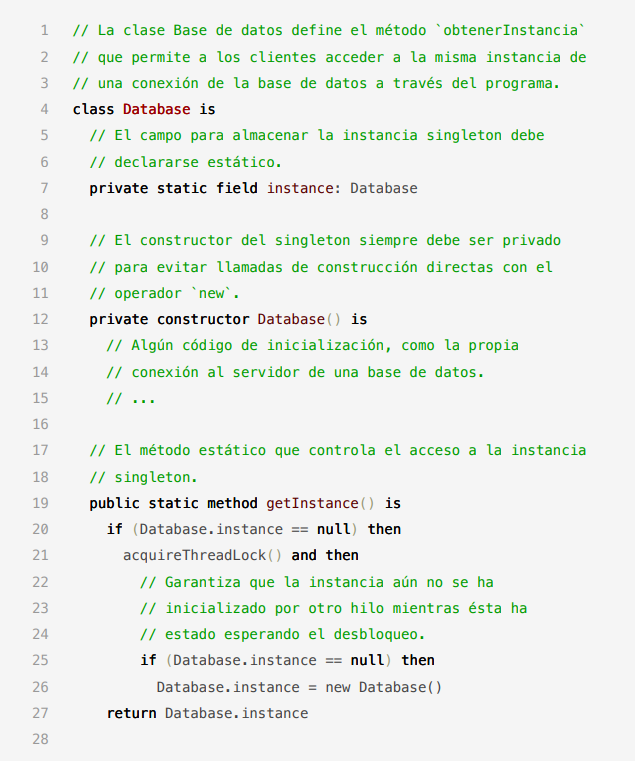
3. Requiere un tratamiento especial en entornos con múltiples hilos de ejecución para evitar la creación de múltiples instancias del Singleton.

4. Puede ser complicado realizar pruebas unitarias del código cliente del Singleton debido a la dificultad para simularlo en muchos frameworks de pruebas. Esto se debe a que la clase Singleton es privada y resulta difícil sobrescribir métodos estáticos para crear objetos simulados (mock objects).

Una clase fachada a menudo puede transformarse en una Singleton, ya que un único objeto fachada es suficiente en la mayoría de los casos.

**Pseudocódigo**

En este ejemplo, la clase de conexión de la base de datos actúa como Singleton. Esta clase no tiene un constructor público, por lo que la única manera de obtener su objeto es invocando el método obtenerInstancia . Este método almacena en caché el primer objeto creado y lo devuelve en todas las llamadas siguientes





**Facade (Fachada)**

Facade es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.

**Problema**

Imagina que debes lograr que tu código trabaje con un amplio grupo de objetos que pertenecen a una sofisticada biblioteca o framework. Normalmente, debes inicializar todos esos objetos, llevar un registro de las dependencias, ejecutar los métodos en el orden correcto y así sucesivamente. Como resultado, la lógica de negocio de tus clases se vería estrechamente acoplada a los detalles de implementación de las clases de terceros, haciéndola difícil de comprender y mantener.

**Solución**

Una fachada es una clase que proporciona una interfaz simple a un subsistema complejo que contiene muchas partes

móviles. Una fachada puede proporcionar una funcionalidad

limitada en comparación con trabajar directamente con el subsistema. Sin embargo, tan solo incluye las funciones realmente

importantes para los clientes.

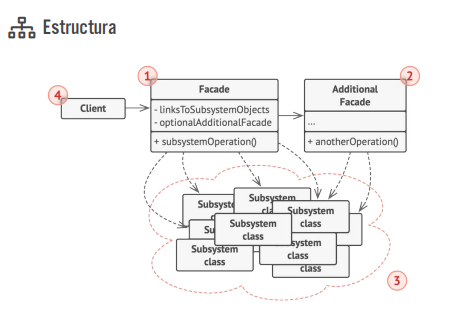
Tener una fachada resulta útil cuando tienes que integrar tu

aplicación con una biblioteca sofisticada con decenas de funciones, de la cual sólo necesitas una pequeña parte.

Por ejemplo, una aplicación que sube breves vídeos divertidos

de gatos a las redes sociales, podría potencialmente utilizar

una biblioteca de conversión de vídeo profesional. Sin embargo, lo único que necesita en realidad es una clase con el método simple codificar(nombreDelArchivo, formato) . Una vez que crees dicha clase y la conectes con la biblioteca de conversión de vídeo, tendrás tu primera fachada.



1. El patrón Facade proporciona un práctico acceso a una parte específica de la funcionalidad del subsistema. Sabe a dónde dirigir la petición del cliente y cómo operar todas las partes móviles.

2. Puede crearse una clase Fachada Adicional para evitar contaminar una única fachada con funciones no relacionadas que podrían convertirla en otra estructura compleja. Las fachadas adicionales pueden utilizarse por clientes y por otras fachadas.

3. El Subsistema Complejo consiste en decenas de objetos diversos. Para lograr que todos hagan algo significativo, debes profundizar en los detalles de implementación del subsistema, que pueden incluir inicializar objetos en el orden correcto y suministrarles datos en el formato adecuado. Las clases del subsistema no conocen la existencia de la fachada. Operan dentro del sistema y trabajan entre sí directamente.

4. El Cliente utiliza la fachada en lugar de invocar directamente los objetos del subsistema

**Aplicabilidad**

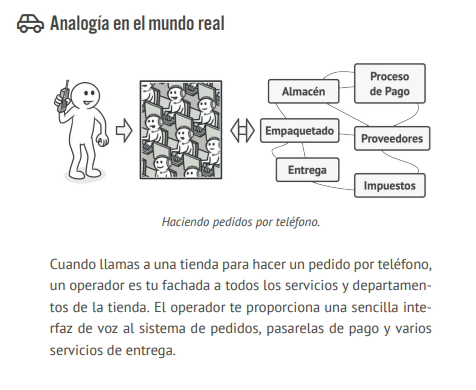
Utiliza el patrón Facade cuando necesites una interfaz limitada pero directa a un subsistema complejo. A menudo los subsistemas se vuelven más complejos con el tiempo. Incluso la aplicación de patrones de diseño suele conducir a la creación de un mayor número de clases. Un subsistema puede hacerse más flexible y más fácil de reutilizar en varios contextos, pero la cantidad de código de configuración que exige de un cliente, crece aún más. El patrón Facade intenta solucionar este problema proporcionando un atajo a las funciones más utilizadas del subsistema que mejor encajan con los requisitos del cliente.

**Pros**:

1. Simplicidad de uso: Proporciona una interfaz simplificada que oculta la complejidad del sistema subyacente, facilitando su uso para los clientes.
2. Abstracción: Permite a los clientes interactuar con el sistema sin necesidad de conocer los detalles internos de su implementación, lo que promueve una mayor modularidad y flexibilidad en el diseño del sistema.
3. Desacoplamiento: Reduce la dependencia entre los clientes y los componentes internos del sistema, lo que facilita los cambios en la implementación sin afectar a los clientes que utilizan la fachada.
4. Mejora la legibilidad y mantenibilidad del código: Al proporcionar una interfaz clara y unificada, facilita la comprensión y el mantenimiento del código, ya que los cambios en el subsistema sólo afectan a la fachada y no a los clientes.
5. Promueve buenas prácticas de diseño: Fomenta la separación de preocupaciones y la cohesión en el diseño del sistema al proporcionar una capa de abstracción entre los clientes y el subsistema.

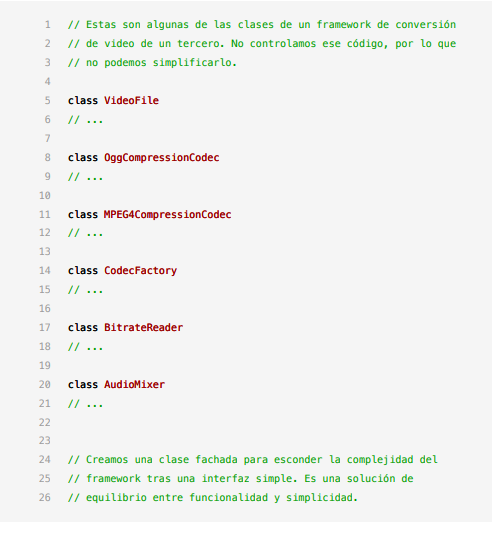
**Contras**:

1. Acoplamiento indirecto: Puede introducir un tipo diferente de acoplamiento entre la fachada y los componentes internos del sistema, lo que podría hacer que los cambios en los componentes internos requieran modificaciones en la fachada.
2. Complejidad de mantenimiento: A medida que el sistema evoluciona, la fachada puede volverse más compleja para adaptarse a nuevas funcionalidades o cambios en el subsistema, lo que podría dificultar su mantenimiento y comprensión.
3. Exceso de generalización: En algunos casos, la fachada puede volverse demasiado generalizada para satisfacer las necesidades específicas de ciertos clientes, lo que podría conducir a una interfaz poco eficiente o poco intuitiva.
4. Duplicación de funcionalidad: Si varios clientes requieren funcionalidades similares pero ligeramente diferentes, la fachada puede necesitar proporcionar métodos adicionales para satisfacer estas necesidades específicas, lo que podría llevar a la duplicación de funcionalidad dentro de la fachada.
5. Impacto en el rendimiento: Dependiendo de la implementación, el uso de una fachada podría introducir una capa adicional de indirección en el acceso a los componentes internos del sistema, lo que podría afectar al rendimiento en aplicaciones que requieren un alto rendimiento.



**Pseudocódigo**

En lugar de hacer que tu código trabaje con decenas de las clases del framework directamente, creas una clase fachada que encapsula esa funcionalidad y la esconde del resto del código. Esta estructura también te ayuda a minimizar el esfuerzo de actualizar a futuras versiones del framework o de sustituirlo por otro. Lo único que tendrías que cambiar en la aplicación es la implementación de los métodos de la fachada





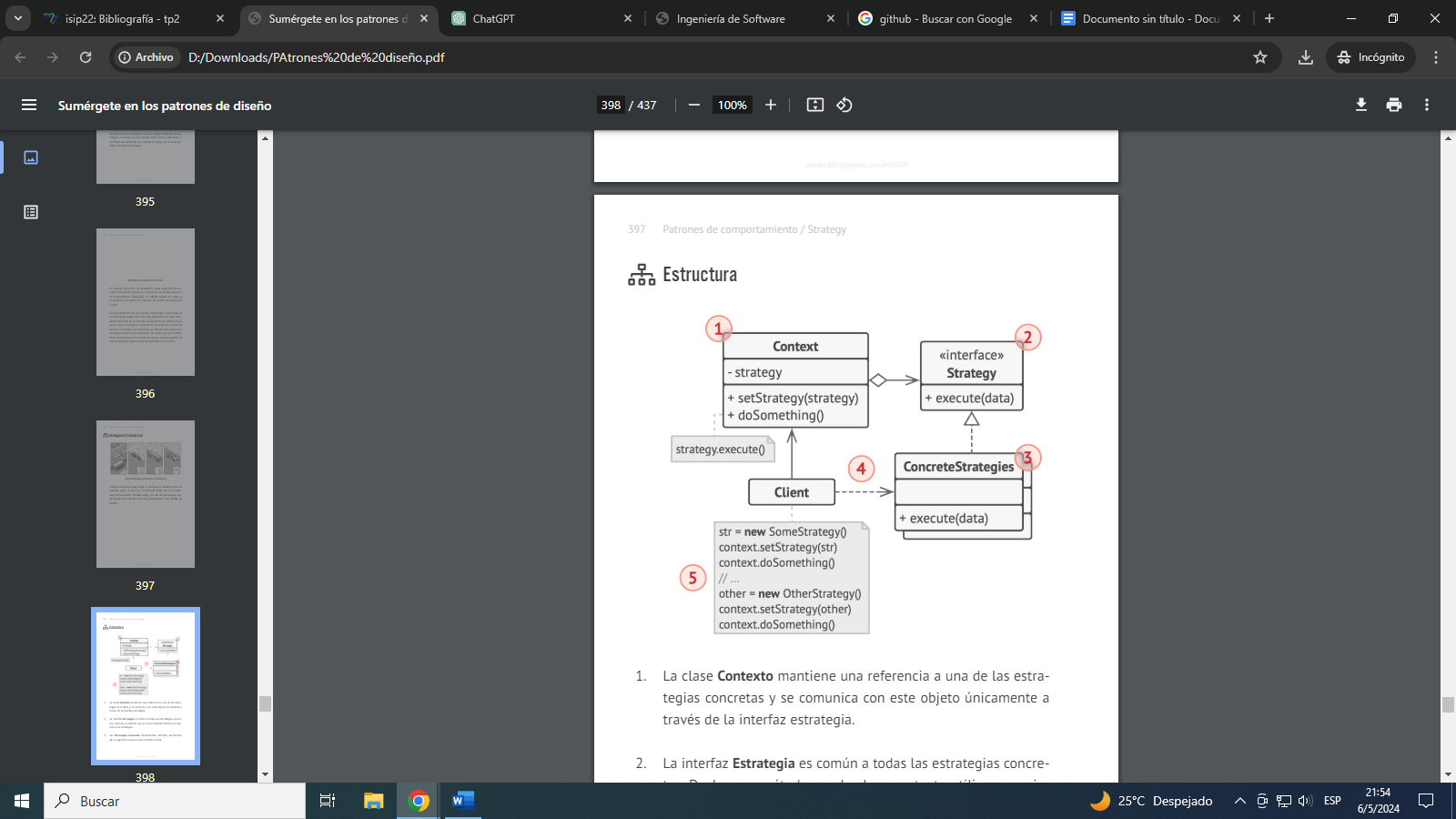
**Strategy**

El patrón de diseño de comportamiento Strategy es una metodología fundamental dentro de la ingeniería de software que permite manejar de manera flexible y eficiente la variabilidad en el comportamiento de un sistema. Básicamente, Strategy encapsula algoritmos en clases separadas, permitiendo que estos algoritmos puedan variar independientemente del código que los utiliza. Esta separación de responsabilidades facilita la modificación, expansión y mantenimiento del sistema a lo largo del tiempo.

Podemos decir que se basa en el principio de composición sobre herencia. En lugar de forzar la variabilidad a través de la herencia de clases, donde cada subclase implementa una variante del comportamiento, Strategy propone encapsular cada variante de comportamiento en su propia clase. Estas clases, conocidas como estrategias, implementan una interfaz común, lo que permite que sean intercambiables dentro del contexto de la aplicación.

Esto significa que el código cliente no depende de algoritmos concretos, sino de interfaces abstractas. El cliente puede seleccionar dinámicamente la estrategia deseada en tiempo de ejecución, lo que proporciona una gran flexibilidad y extensibilidad al sistema. Además, el patrón Strategy promueve el principio de diseño de código abierto/cerrado, ya que permite la introducción de nuevas estrategias sin modificar el código existente.

La implementación del patrón Strategy sigue una estructura definida. En primer lugar, se define una interfaz común que todas las estrategias deben implementar. Luego, se crean clases concretas que encapsulan diferentes variantes del algoritmo. Estas clases interactúan a través de la interfaz común, lo que permite que el contexto utilice cualquier estrategia sin conocer los detalles de implementación.

****

La aplicabilidad del patrón Strategy es muy amplia. Se utiliza en situaciones donde se necesita manejar múltiples algoritmos que pueden variar independientemente del código cliente. Es especialmente útil cuando se espera que estos algoritmos cambien o se expandan con el tiempo, ya que nos proporciona una estructura flexible y fácil de mantener.

Por ejemplo, vamos a plantear un problema.

**Contexto del problema:**

Imagina que estás desarrollando una aplicación de navegación. Inicialmente, esta aplicación sólo puede calcular rutas para viajar en coche. Sin embargo, a medida que la aplicación se vuelve más popular, los usuarios solicitan funciones adicionales, como rutas para caminar, usar transporte público, e incluso rutas que incluyan visitas turísticas.

**Desafíos técnicos:**

A medida que agregamos nuevas funcionalidades a la aplicación, te enfrentas a desafíos técnicos significativos. La clase principal de la aplicación se vuelve cada vez más compleja y difícil de mantener. Cada nuevo algoritmo de enrutamiento que agregas aumenta la complejidad de esta clase, lo que hace que sea más propensa a errores y más difícil de entender.

Además, trabajar en equipo se vuelve ineficiente. Tus compañeros de equipo se quejan de que es difícil trabajar en el código debido a su complejidad y tamaño. Cualquier cambio en uno de los algoritmos afecta a toda la clase, lo que dificulta la colaboración y aumenta el riesgo de introducir errores.

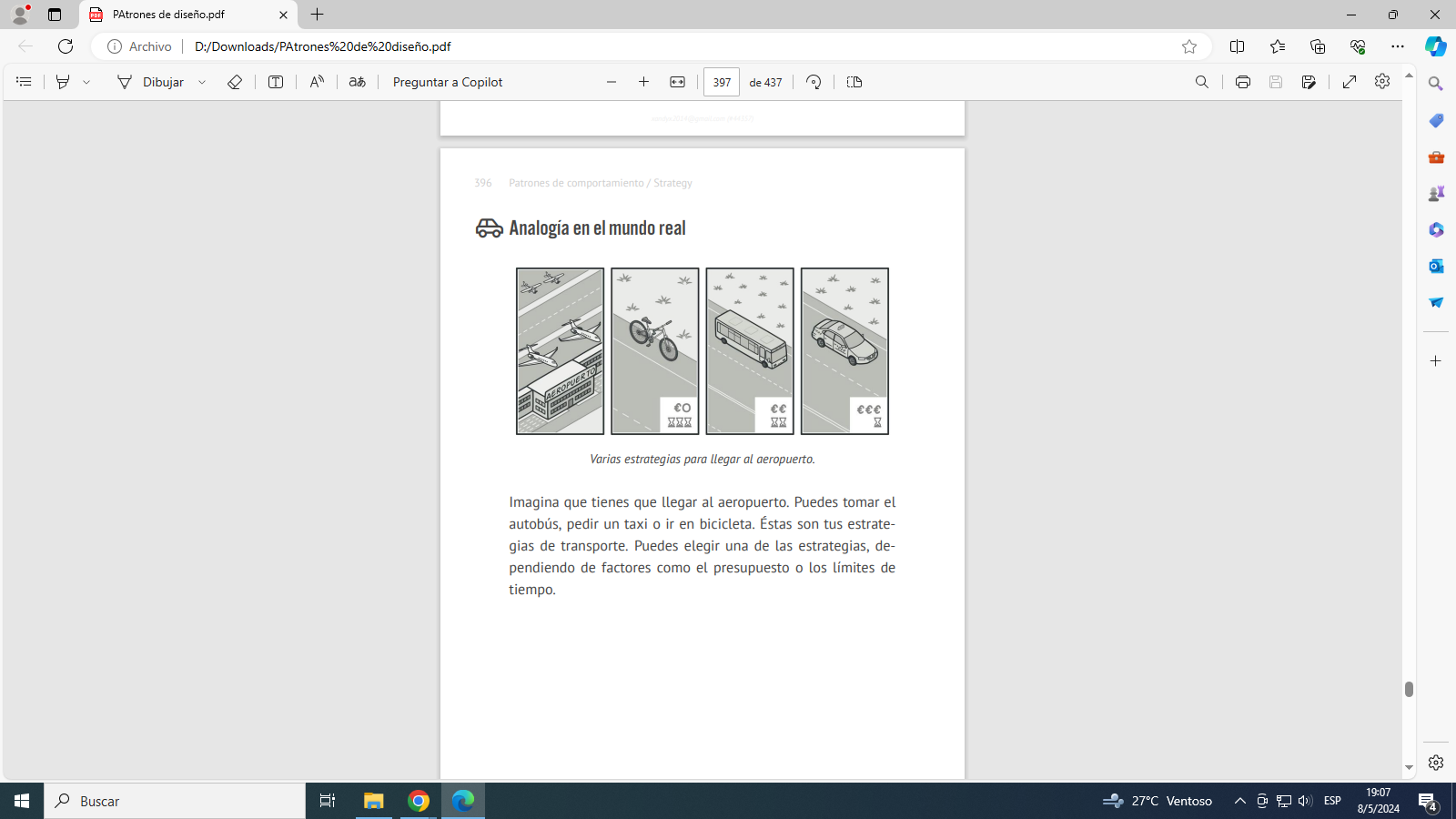
**Solución que proponemos con el patrón Strategy:**

El patrón Strategy propone una solución elegante a estos problemas. En lugar de tener un solo algoritmo de enrutamiento dentro de la clase principal de la aplicación, cada algoritmo se encapsula en su propia clase separada. Estas clases se conocen como estrategias.

La clase principal de la aplicación, conocida como contexto, tiene un campo para almacenar una referencia a una de estas estrategias. En lugar de implementar el algoritmo de enrutamiento directamente, el contexto delega esta responsabilidad a la estrategia seleccionada.

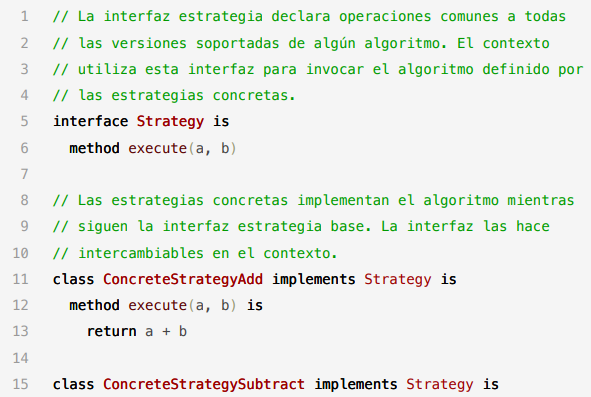
**Beneficios del patrón Strategy:**

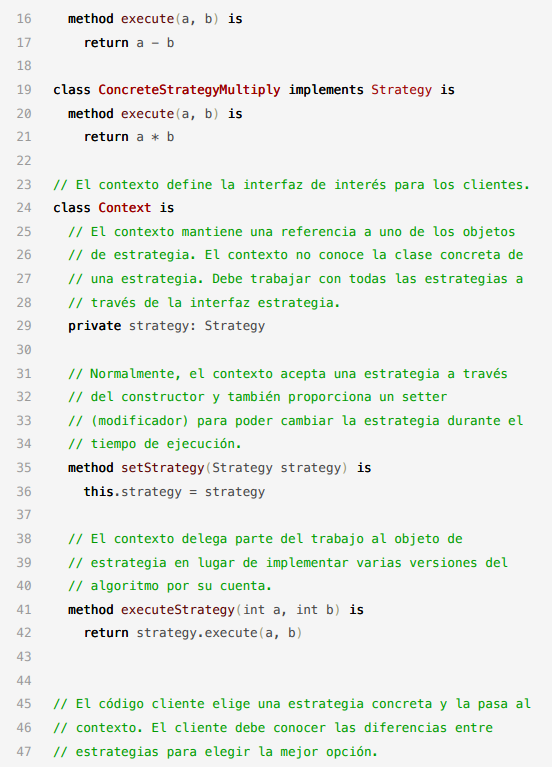
* Flexibilidad: el patrón Strategy hace que sea fácil agregar nuevos algoritmos de enrutamiento a la aplicación sin tener que modificar la clase principal.
* Mantenibilidad: al encapsular cada algoritmo en su propia clase, el código se vuelve más modular y fácil de entender, lo que facilita el mantenimiento y la colaboración en equipo.
* Extensibilidad: el patrón Strategy permite que los clientes de la aplicación elijan el algoritmo de enrutamiento que mejor se adapte a sus necesidades, lo que hace que la aplicación sea más flexible y adaptable a diferentes situaciones.

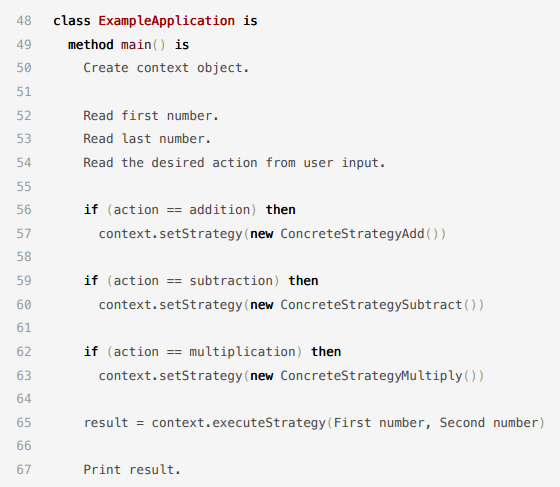


**Pseudocódigo**

En este ejemplo, el contexto utiliza varias estrategias para ejecutar diversas operaciones aritméticas







**Actividad Nº2**

Vamos a dar una breve explicación de los puntos a trabajar para luego aplicarlos a un caso real.

### **12.1 Especificación de requerimientos dirigida por riesgos**

La especificación de requerimientos dirigida por riesgos implica identificar y priorizar los riesgos asociados con el sistema que se está desarrollando. Por ejemplo, en el desarrollo de un sistema de gestión de datos médicos, un riesgo podría ser la pérdida de información confidencial del paciente. En este caso, la especificación de requerimientos dirigida por riesgos podría incluir la implementación de medidas de seguridad robustas para proteger la privacidad del paciente y garantizar la integridad de los datos.

### 

### **12.2 Especificación de protección**

La especificación de protección se refiere a la identificación y definición de los mecanismos de seguridad necesarios para proteger el sistema contra amenazas internas y externas. Por ejemplo, en un sistema de banca en línea, la especificación de protección podría incluir la autenticación de usuarios, el cifrado de datos sensibles durante la transmisión y el almacenamiento, y la implementación de medidas para prevenir ataques de denegación de servicio.

### **12.3 Especificación de fiabilidad**

La especificación de fiabilidad implica definir los requisitos relacionados con la disponibilidad, confiabilidad y mantenibilidad del sistema. Por ejemplo, en el caso de un sistema de control de tráfico aéreo, la especificación de fiabilidad podría incluir requisitos para garantizar que el sistema esté disponible las 24 horas del día, los 7 días de la semana, que sea capaz de manejar cargas de trabajo variables y que pueda recuperarse rápidamente de posibles fallos.

### **12.4 Especificación de seguridad**

La especificación de seguridad implica definir los requisitos relacionados con la protección de los activos del sistema (como datos y recursos) contra amenazas maliciosas o accidentales. Por ejemplo, en el desarrollo de un sistema de gestión de inventario para una empresa, la especificación de seguridad podría incluir requisitos para controlar el acceso a la información confidencial, detectar y prevenir intrusiones, y garantizar la integridad de los datos.

### **12.5 Especificación formal**

La especificación formal implica expresar los requisitos del sistema utilizando un lenguaje formal y matemático. Esto permite una mayor precisión y rigurosidad en la definición de los requisitos, lo que puede ayudar a evitar ambigüedades y errores de interpretación. Por ejemplo, en el desarrollo de un sistema de control de semáforos, la especificación formal podría incluir modelos matemáticos que describan el comportamiento esperado del sistema bajo diferentes condiciones de tráfico.

**A continuación vamos a aplicar estos puntos en un caso real.**

En la era digital actual, la protección de la información sensible y la garantía de la confiabilidad de los sistemas de software son aspectos críticos para cualquier organización, especialmente aquellas que manejan datos altamente confidenciales. En este contexto, consideremos el desarrollo de un sistema de control de acceso biométrico para una empresa que gestiona información altamente confidencial, como datos financieros y de investigación.Sin embargo, la implementación de tales sistemas conlleva una serie de desafíos relacionados con la confiabilidad y la seguridad, que deben abordarse meticulosamente desde las etapas iniciales del proceso de ingeniería de software.

#### **1. Especificación de requerimientos dirigida por riesgos:**

Para ilustrar este punto, consideremos un escenario donde una empresa multinacional de tecnología está desarrollando un sistema de control de acceso biométrico para sus instalaciones de investigación y desarrollo. Uno de los riesgos identificados es la posibilidad de suplantación de identidad mediante falsificación biométrica. En respuesta a este riesgo, se establece el requisito de implementar múltiples métodos biométricos, como reconocimiento facial y huella dactilar, para aumentar la robustez del sistema contra suplantaciones de identidad.

#### **2. Especificación de protección:**

El sistema debe garantizar la protección de los datos biométricos y la confidencialidad de la información almacenada. En este sentido, se establece el requisito de que todas las transmisiones de datos entre el dispositivo biométrico y el servidor central deben estar cifradas utilizando algoritmos criptográficos seguros. Además, se implementan protocolos de comunicación seguros, como HTTPS, para garantizar la confidencialidad e integridad de los datos durante la transmisión.

#### **3. Especificación de fiabilidad:**

La disponibilidad del sistema es fundamental para garantizar un acceso continuo y sin interrupciones a las instalaciones y recursos de la empresa. Por lo tanto, se establece el requisito de que el sistema debe estar disponible las 24 horas del día, los 7 días de la semana. Para cumplir con este requisito, se implementan medidas de redundancia en los servidores y sistemas de respaldo para minimizar el tiempo de inactividad en caso de fallos o mantenimiento programado.

#### **4. Especificación de seguridad:**

El control de acceso es esencial para garantizar que solo los usuarios autorizados puedan acceder al sistema biométrico y a las instalaciones protegidas. Se establece el requisito de que se deben implementar medidas de seguridad adicionales, como contraseñas o tarjetas de acceso, para aumentar la protección contra accesos no autorizados. Además, se integra un sistema de gestión de identidades que administra los roles y privilegios de usuario, así como la auditoría de accesos para rastrear y registrar todas las interacciones con el sistema.

#### **5. Especificación formal:**

Los algoritmos de reconocimiento biométrico deben estar formalmente especificados y validados para garantizar su precisión y confiabilidad. Para cumplir con este requisito, se utiliza un enfoque de modelado formal, donde se emplean lenguajes y herramientas formales para describir y verificar el comportamiento del sistema en diferentes escenarios de uso. Esto asegura que el sistema cumpla con los estándares de calidad y precisión requeridos para su despliegue en un entorno operativo real.

Como conclusión, podemos decir que, el desarrollo de un sistema de control de acceso biométrico implica una cuidadosa consideración de los aspectos relacionados con la confiabilidad y seguridad del sistema. Desde la identificación de riesgos hasta la especificación de medidas de protección y la validación formal de los algoritmos biométricos, cada paso en el proceso de ingeniería de software contribuye a garantizar la integridad y la confiabilidad del sistema en su conjunto. Este enfoque proactivo y meticuloso es fundamental para asegurar que el sistema cumpla con los requisitos de seguridad y rendimiento establecidos por la organización.