UNIVERSIDAD DE LA CUENCA DEL PLATA



FACULTAD DE INGENIERÍA Y TECNOLOGÍA

TRABAJO PRÁCTICO Nº 2

Estudiantes:

∙ Cáceres, Hugo Fabricio

∙ Casals, Sebastián Alejandro

Materia: Ingeniería de Software II

Curso: 3º Año - Ingeniería en Sistemas de Información

Profesor: Kutz, Gabriel

Año: 2024

**Trabajo Práctico N° 2**

**Investigación y Presentación de Patrones de diseño de Software**

**Fecha de entrega**: 04/05/2024

**Ejes Temáticos**:

* Nº 2: Diseño Orientado a Objetos (Patrones de Software).
* Nº 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

Tabla de contenido

[INTRODUCCIÓN 6](#_Toc166162101)

[PATRÓN DE CREACIÓN ELEGIDO: PROTOTYPE 7](#_Toc166162102)

[DEFINICIÓN 7](#_Toc166162103)

[ANALOGIA DEL MUNDO REAL 9](#_Toc166162104)

[ESTRUCTURA DEL PATRON 10](#_Toc166162105)

[CONSECUENCIAS 11](#_Toc166162106)

[Positivas 11](#_Toc166162107)

[Negativas 11](#_Toc166162108)

[PSEUDOCODIGO DEL PATRON 12](#_Toc166162109)

[Ejemplo 12](#_Toc166162110)

[Explicando del código: 12](#_Toc166162111)

[Explicación del Diagrama UML 13](#_Toc166162112)

[APLICABILIDAD 15](#_Toc166162113)

[¿COMO IMPLEMENTARLO? 15](#_Toc166162114)

[PATRON DE COMPORTAMIENTO ELEGIDO: OBSERVER 17](#_Toc166162115)

[DEFINICION 17](#_Toc166162116)

[PROBLEMA PLANTEADO 18](#_Toc166162117)

[SOLUCION AL PROBLEMA PLANTEADO 19](#_Toc166162118)

[ANALOGIA EN EL MUNDO REAL 20](#_Toc166162119)

[ESTRUCTURA DEL PATRON OBSERVADOR 21](#_Toc166162120)

[PSEUDOCODIGO DEL PATRON 23](#_Toc166162121)

[APLICABILIDAD 30](#_Toc166162122)

[IMPLEMENTACION 31](#_Toc166162123)

[PROS Y CONTRAS DEL PATRON 33](#_Toc166162124)

[RELACION CON OTROS PATRONES DE COMPORTAMIENTO 33](#_Toc166162125)

[CONCLUSIÓN 34](#_Toc166162126)

[ESPECIFICACION DE CONFIABILIDAD Y SEGURIDAD 35](#_Toc166162127)

[Introducción 35](#_Toc166162128)

[Especificación de requerimientos 36](#_Toc166162129)

[Especificación de Protección 37](#_Toc166162130)

[Especificación de Fiabilidad 42](#_Toc166162131)

[Especificación de Seguridad 49](#_Toc166162132)

[Especificación Formal 50](#_Toc166162133)

[Conclusión 53](#_Toc166162134)

[BIBLIOGRAFÍA 54](#_Toc166162135)

# INTRODUCCIÓN

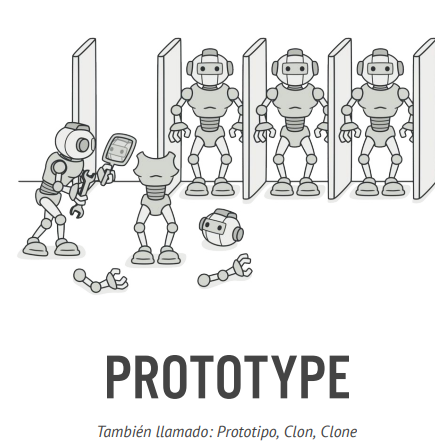
Los patrones de diseño de software son soluciones probadas y recurrentes a problemas comunes que enfrentan los desarrolladores durante el diseño y la implementación de software. Se derivan de la experiencia acumulada de expertos en el campo y ofrecen un marco de trabajo para abordar problemas de diseño de manera efectiva y eficiente.

Estos patrones proporcionan un lenguaje común para que los desarrolladores discutan soluciones a problemas comunes, lo que facilita la comunicación y la comprensión entre equipos de desarrollo. Además, permiten la reutilización de soluciones exitosas, lo que promueve la consistencia, la mantenibilidad y la escalabilidad del código.

Los patrones de diseño pueden clasificarse en varias categorías, como patrones de creación, patrones estructurales y patrones de comportamiento, cada uno con su propio propósito y aplicación específicos. Al comprender y aplicar estos patrones de manera adecuada, los desarrolladores pueden mejorar la calidad y la robustez de sus sistemas de software, al tiempo que reducen la probabilidad de errores y problemas de diseño.

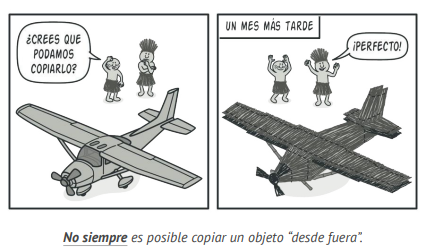
# PATRÓN DE CREACIÓN ELEGIDO: PROTOTYPE

## DEFINICIÓN



**Prototype** es un **patrón de diseño creacional** que nos permite copiar objetos existentes sin que el código dependa de sus clases.

Como primer paso, se debe crear un nuevo objeto de la misma clase. Luego, recorrer todos los campos del objeto original y copiar sus valores en el nuevo objeto. Pero no todos los objetos se pueden copiar de este modo, porque algunos de los campos del objeto pueden ser privados e invisibles desde fuera del propio objeto.



Hay otro problema con el enfoque directo. Dado que se debe conocer la clase del objeto para crear un duplicado, el código se vuelve dependiente de esa clase.

Como solución podemos definir lo siguiente:

**El patrón Prototype** delega el proceso de clonación a los propios objetos que están siendo clonados. El patrón declara una interfaz común para todos los objetos que soportan la clonación. Esta interfaz nos permite clonar un objeto sin acoplar el código a la clase de ese objeto. Normalmente, dicha interfaz contiene un único método clonar.

La implementación del método clonar es muy parecida en todas las clases. El método crea un objeto a partir de la clase actual y lleva todos los valores de campo del viejo objeto, al nuevo.

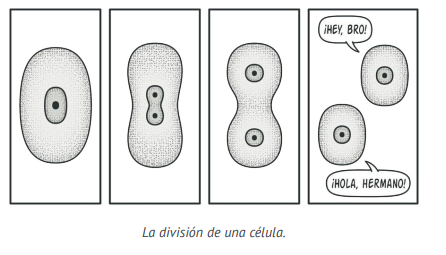
Un objeto que soporta la clonación se denomina prototipo. Cuando los objetos tienen decenas de campos y miles de configuraciones posibles, la clonación puede servir como alternativa a la creación de subclases.



Se crea un grupo de objetos configurados de maneras diferentes. Cuando se necesita un objeto ya configurado, se clona un prototipo en lugar de construir un nuevo objeto desde cero.

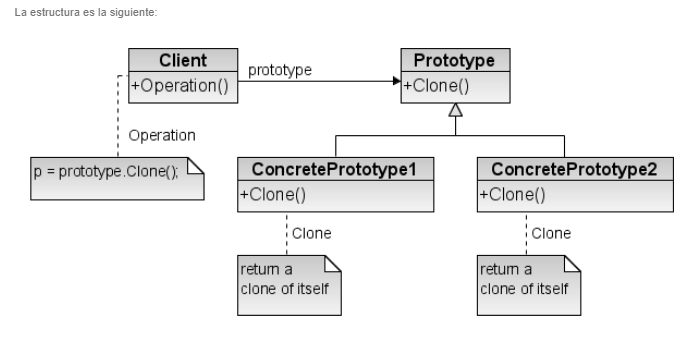
## ANALOGIA DEL MUNDO REAL

En la vida real, los prototipos se utilizan para realizar pruebas de todo tipo antes de comenzar con la producción en masa de un producto. Sin embargo, en este caso, los prototipos no forman parte de una producción real, sino que juegan un papel pasivo.



Ya que los prototipos industriales en realidad no se copian a sí mismos, una analogía más precisa del patrón es el proceso de la división mitótica de una célula. Tras la división mitótica, se forma un par de células idénticas. La célula original actúa como prototipo y asume un papel activo en la creación de la copia.

## ESTRUCTURA DEL PATRON



* **Prototype**: Interfaz que define la operación de clonado.

Será implementada por todos los objetos que puedan ser clonados. En ocasiones es implementado como una clase abstracta.

* **Client**: Crea nuevos objetos mediante la clonación de los mismos.
* **ConcretePrototype1**: Clase que puede ser instanciada mediante la clonación de un prototipo. Implementa la interfaz Prototype.
* **ConcretePrototype2**: Otra clase con las mismas características que ConcretePrototype1.

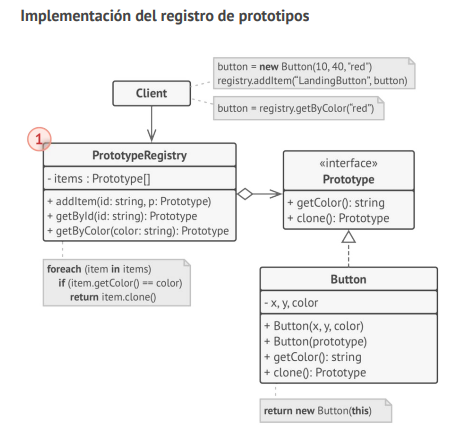
## CONSECUENCIAS

Positivas:

* Clonar un objeto es mucho más rápido que crearlo.
* Un programa puede añadir y borrar dinámicamente objetos prototipo en tiempo de ejecución.
* El cliente no debe conocer los detalles de cómo construir los objetos prototipo.

#### **Negativas**:

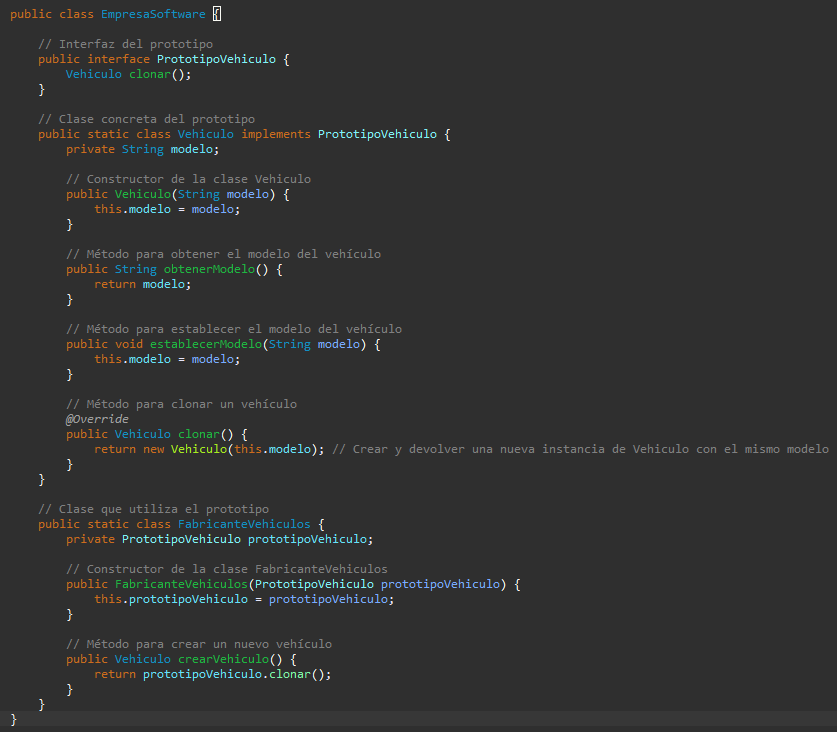
* En objetos muy complejos, implementar la interfaz Prototype puede ser muy complicada.



El Registro de Prototipos ofrece una forma sencilla de acceder a prototipos de uso frecuente. Almacena un grupo de objetos prefabricados listos para ser copiados.

## PSEUDOCODIGO DEL PATRON

Ejemplo: Supongamos que tienes una clase Vehicle y deseas crear nuevas instancias de vehículos a partir de un prototipo existente:



### Explicando del código:

* **EmpresaSoftware**: Es la clase principal que contiene las otras clases.
* **PrototipoVehiculo**: Es una interfaz que define el método “clonar()”. Esta interfaz es la abstracción del prototipo en el patrón Prototype.
* **Vehiculo**: Es una clase concreta que implementa “PrototipoVehiculo”. Representa un tipo de vehículo (por ejemplo, un automóvil).

Tiene un atributo “modelo” que representa el modelo del vehículo.

Tiene métodos para obtener y establecer el modelo del vehículo.

Implementa el método “clonar()”, que devuelve una nueva instancia de “Vehiculo” con el mismo modelo. Esto permite la creación de nuevas instancias de vehículos a partir de un prototipo existente.

* **FabricanteVehiculos**: Es una clase que utiliza el prototipo para crear nuevos objetos (“Vehiculo”).

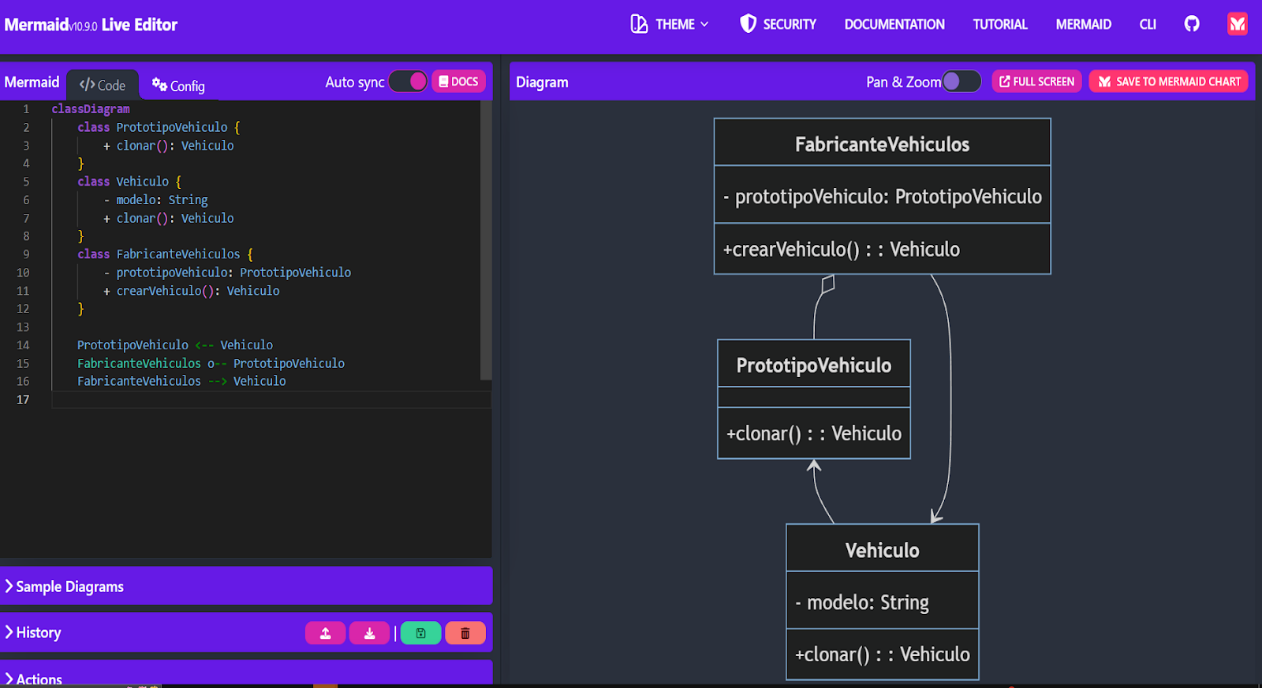
Tiene un atributo “prototipoVehiculo” que almacena una instancia de “PrototipoVehiculo”.

En su constructor, recibe una instancia de PrototipoVehiculo e inicializa el atributo prototipoVehiculo con ella.

Tiene un método “crearVehiculo()” que llama al método “clonar()” del prototipo almacenado en “prototipoVehiculo” para crear y devolver una copia del vehículo.

### Explicación del Diagrama UML

Con respecto al Patrón Prototype (Realizado Con Mermaid):



* “**PrototipoVehiculo**”: Representa la interfaz del prototipo. Define el método “clonar()” que las clases concretas deben implementar para permitir la clonación.
* “**Vehiculo**”: Es una clase concreta que implementa “PrototipoVehiculo”. Representa el prototipo que se clonará para crear nuevos objetos. Implementa el método “clonar()” para permitir la clonación de vehículos.
* “**FabricanteVehiculos**”: Utiliza el prototipo (“PrototipoVehiculo”) para crear nuevos objetos “(Vehiculo)”. Tiene una asociación con PrototipoVehiculo. El método “crearVehiculo()” permite crear nuevos vehículos utilizando el prototipo.
* La asociación entre “**FabricanteVehiculos**” y “**PrototipoVehiculo**” muestra cómo el fabricante de vehículos utiliza el prototipo para crear nuevos objetos. Esto ilustra cómo el patrón Prototype permite la creación de nuevos objetos mediante la clonación de un prototipo existente, lo que ayuda a evitar la creación de objetos costosos o complicados desde cero.

Todas las clases de forma siguen la misma interfaz, que proporciona un método de clonación. Una subclase puede invocar el método de clonación padre antes de copiar sus propios valores de campo al objeto resultante.

## APLICABILIDAD

* Utiliza el patrón Prototype cuando tu código no deba depender de las clases concretas de objetos que necesites copiar.

El patrón Prototype proporciona al código cliente una interfaz general para trabajar con todos los objetos que soportan la clonación. Esta interfaz hace que el código cliente sea independiente de las clases concretas de los objetos que clona.

* Utiliza el patrón cuando quieras reducir la cantidad de subclases que solo se diferencian en la forma en que inicializan sus respectivos objetos.

El patrón Prototype te permite utilizar como prototipos un grupo de objetos prefabricados, configurados de maneras diferentes.

En lugar de instanciar una subclase que coincida con una configuración, el cliente puede, sencillamente, buscar el prototipo adecuado y clonarlo.

## ¿COMO IMPLEMENTARLO?

1. Crea la interfaz del prototipo y declara el método clonar en ella, o, simplemente, añade el método a todas las clases de una jerarquía de clase existente, si la tienes.
2. Una clase de prototipo debe definir el constructor alternativo que acepta un objeto de dicha clase como argumento. El constructor debe copiar los valores de todos los campos definidos en la clase del objeto que se le pasa a la instancia recién creada. Si deseas cambiar una subclase, debes invocar al constructor padre para permitir que la superclase gestione la clonación de sus campos privados.

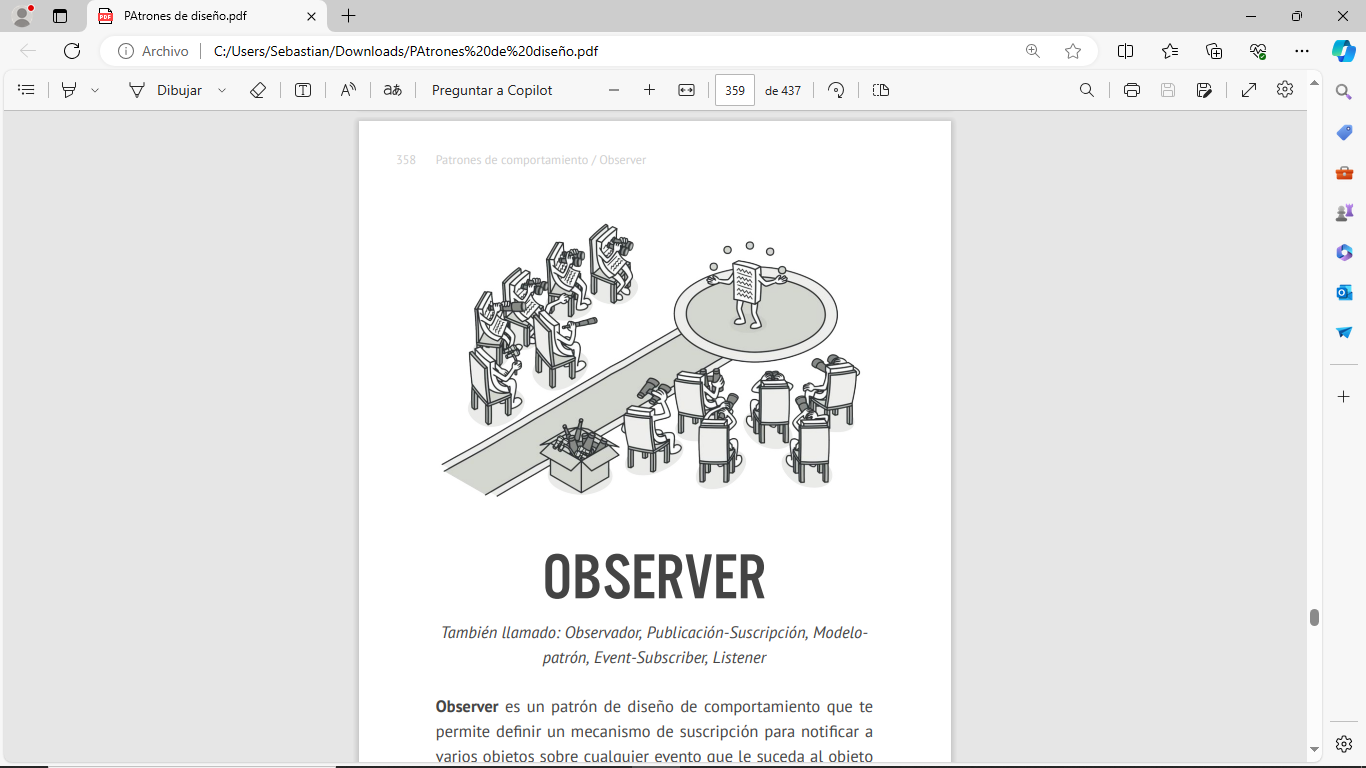
Si el lenguaje de programación que utilizas no soporta la sobrecarga de métodos, puedes definir un método especial para copiar la información del objeto.

1. Normalmente, el método de clonación consiste en una sola línea que ejecuta un operador new con la versión prototípica del constructor.
2. Opcionalmente, puedes crear un registro de prototipos centralizado para almacenar un catálogo de prototipos de uso frecuente.

Por último, sustituye las llamadas directas a los constructores de las subclases por llamadas al método de fábrica del registro de prototipos.

# PATRON DE COMPORTAMIENTO ELEGIDO: OBSERVER

## DEFINICION



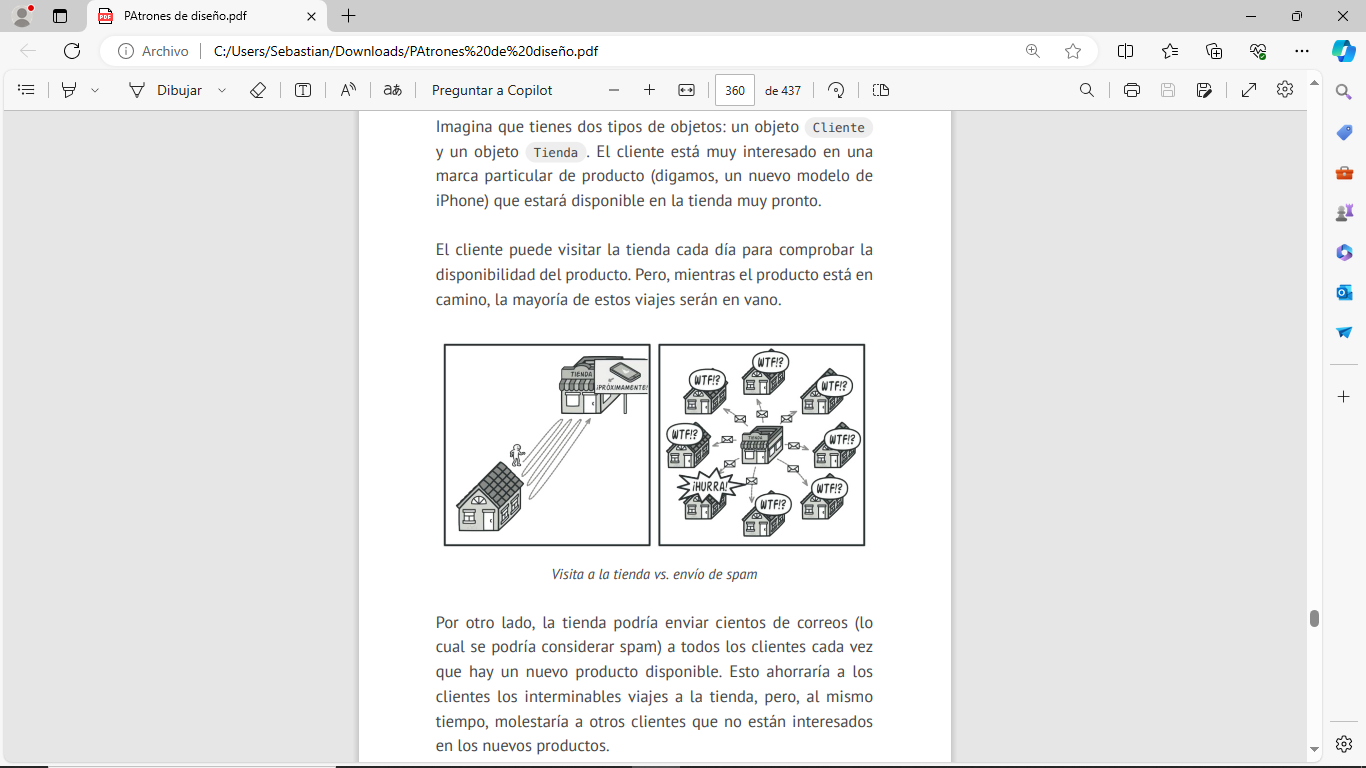
Estos tipos de patrones se centran en cómo las clases y objetos interactúan entre sí, y en cómo distribuyen las responsabilidades.

Para éste trabajo, he elegido el patrón de comportamiento denominado “Observer”, también conocido de otras formas, como ser: Observador, Listener, Publicación / Suscripción.

Este patrón nos permite definir un mecanismo de suscripción para notificar a otros objetos sobre cualquier eventualidad por la que pase el objeto al cuál se está observando.

Lo que hace este patrón básicamente es definir una dependencia uno a muchos entre objetos, cuyo objetivo es tener a todos los objetos dependientes notificados y actualizados automáticamente de algún cambio en el estado del objeto del cual dependen.

## PROBLEMA PLANTEADO



Tenemos dos tipos de objetos: objeto “Cliente” y objeto “Tienda”.

El “Cliente” se interesa en un producto de una marca en particular que va a estar disponible en la tienda dentro de un tiempo próximo. Este producto puede ser la última versión del Apple Watch.

A partir de esta situación, tenemos dos escenarios: En primer lugar, el cliente podría visitar la tienda día a día para comprobar la disponibilidad del producto de interés, no obstante, mientras el producto este en camino a la tienda, la mayoría de esos viajes a la tienda por parte del cliente serán en vano.

En segundo lugar, desde la tienda se podrían enviar una gran cantidad de correos a los clientes (esto podría considerarse como spam) cada vez que un nuevo producto llega a ella, permitiéndole así al cliente ahorrarse el viaje hasta la tienda para comprobar la disponibilidad pero sería molesto para los demás clientes que no estén interesados en ese producto. Esto nos genera un conflicto.

## SOLUCION AL PROBLEMA PLANTEADO

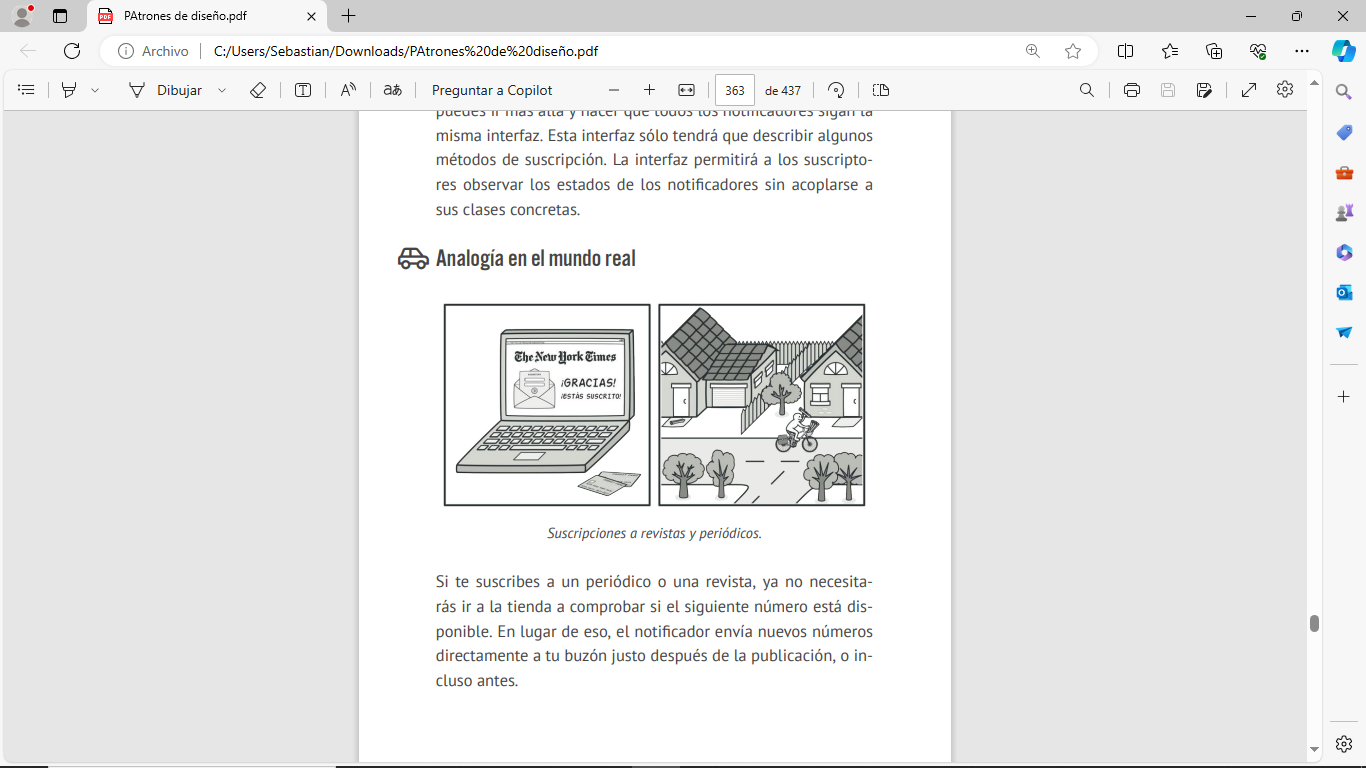
Para solucionar este conflicto analizamos algunas cuestiones: aquel objeto que tenga un estado que genere interés suele denominarse “Sujeto”, este mismo objeto puede notificar a otros objetos sobre los cambios en su estado, por lo tanto se denomina “Notificador” o “Publicador”, por último, aquellos objetos que desean conocer sobre los cambios en el estado del objeto notificador, se denominaran “Suscriptores”.

El patrón Observer nos alienta a añadir un mecanismo de suscripción a la clase notificadora para que objetos individuales puedan suscribirse o deshacer la suscripción a un flujo de eventos que provienen de la clase notificadora. Este mecanismo se puede lograr mediante un campo matriz que almacene una lista de referencias a los objetos suscriptores, y mediante métodos públicos que permitan añadir, notificar, eliminar suscritores de dicha lista. Entonces, cuando le suceda un evento importante al notificador, recorrerá sobre sus suscriptores y llamara al método de notificación específico de sus objetos.

Es necesario, y fundamental, que todos los suscriptores implementen la misma interfaz y que el notificador se comunique con ellos mediante dicha interfaz. Además, en ella debe declarar un método de notificación acompañada de parámetros que puedan ser usados por el notificador para pasar información contextual con la notificación.

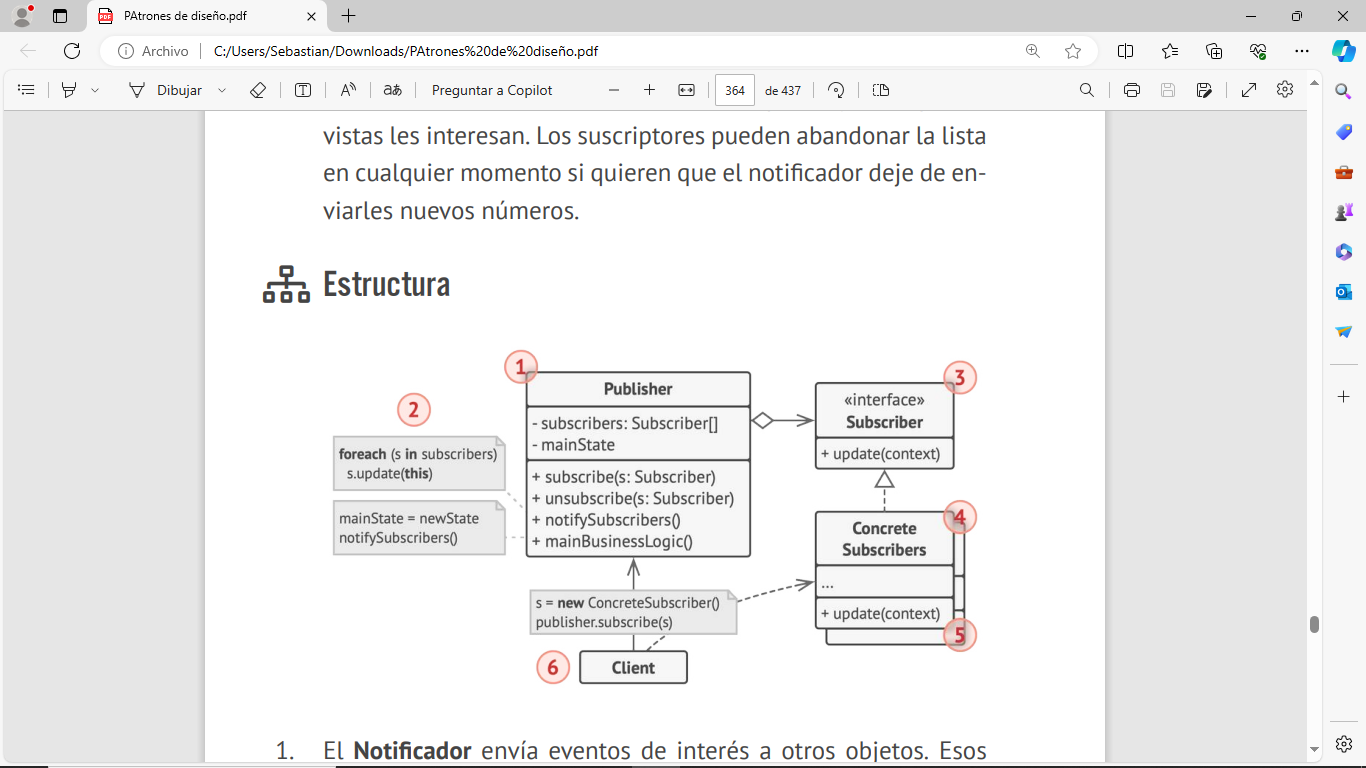
Para un caso en el que se tenga varios tipos diferentes de notificadores, y se quiera hacer compatible con todos ellos a los suscriptores, todos estos notificadores deberían seguir la misma interfaz, acompañada de algunos métodos de suscripción, permitiéndoles observar los estados de los notificadores sin estar acoplados a sus clases concretas.

## ANALOGIA EN EL MUNDO REAL



* Uno de estos ejemplos podría ser el caso de una suscripción a un periódico local o a una revista, esto nos permite evitarnos el viaje a la tienda para comprobar la disponibilidad del periódico/revista, y recibirlo por medio del notificador directamente en nuestro buzón justo después de su publicación
* Otro ejemplo podría ser un sistema de noticias o alertas meteorológicas en una aplicación móvil, el sujeto seria el servicio de noticias o proveedor de datos meteorológicos, al detectar cambios en las condiciones climáticas, pasa a ser un notificador, ya que debe informar a los usuarios registrados en la app (suscriptores) mediante una alerta en tiempo real en sus dispositivos móviles

## ESTRUCTURA DEL PATRON OBSERVADOR



1. **El Notificador** / **Publicador** envía a otros objetos eventos de interés, asociados a un cambio en su estado o a la ejecución de algunos comportamientos. Además, contienen una infraestructura de suscripción que permite a nuevos o antiguos suscriptores abandonar la lista.
2. Al suceder un evento nuevo, el notificador va a recorrer la lista de suscripción e invocara el método de notificación que está declarado en la interfaz suscriptora en cada suscriptor.
3. La interfaz de notificación estará declarada por la **interfaz Suscriptora** y mayormente consiste en un método denominado “actualizar”, el cual puede tener varios parámetros que van a permitir al notificador pasar detalles del evento junto a la actualización.
4. Los **Suscriptores Concretos** realizan acciones en respuesta a las notificaciones que son emitidas por el notificador. Las clases deben implementar la misma interfaz de forma que el notificador no este acoplado a alguna clase concreta.
5. Los **suscriptores** suelen necesitar información contextual para manejar la actualización de manera correcta, por esto los notificadores pasan cierta información de contexto como argumentos del método de notificación.

Si se desea que los suscriptores extraigan la información que necesitan, el notificador puede pasarse a sí mismo como argumento.

1. El **Cliente** crea objetos del tipo notificador y suscriptor por separado, luego registra a los suscriptores para las actualizaciones del notificador.

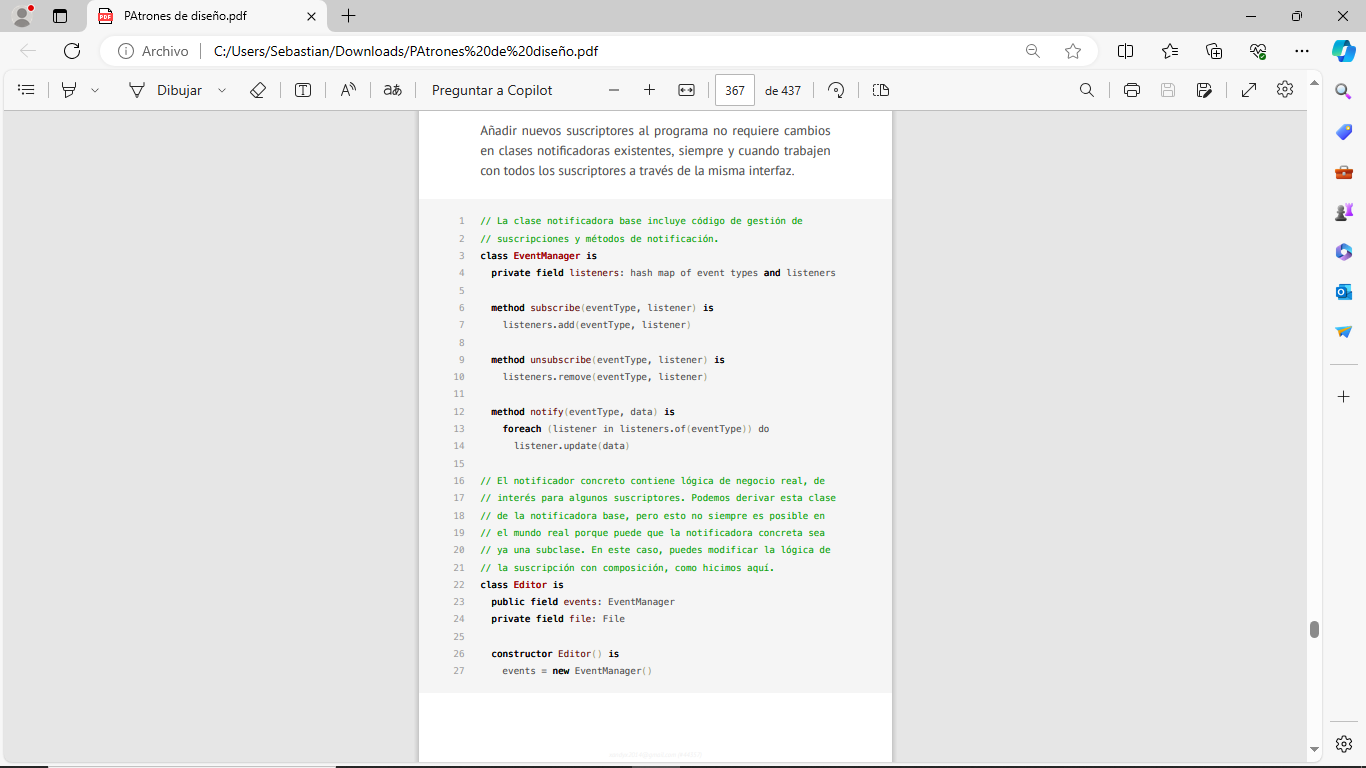
## PSEUDOCODIGO DEL PATRON

En el ejemplo que se muestra a continuación, el patrón Observer permite al objeto “editor de texto” notificar a otros objetos tipo servicio, sobre los cambios en su estado.

La lista de suscriptores se compila dinámicamente, durante el tiempo de ejecución los objetos pueden empezar o parar de escuchar notificaciones, dependiendo del comportamiento deseado para la aplicación.

La clase editora no mantiene la lista de suscripción por sí misma, delega el trabajo al objeto “ayudante especial” que estará dedicado justo a eso, podrá ser actualizado para que sirva como un despachador centralizado de eventos, dejando que cualquier objeto actué como notificador.

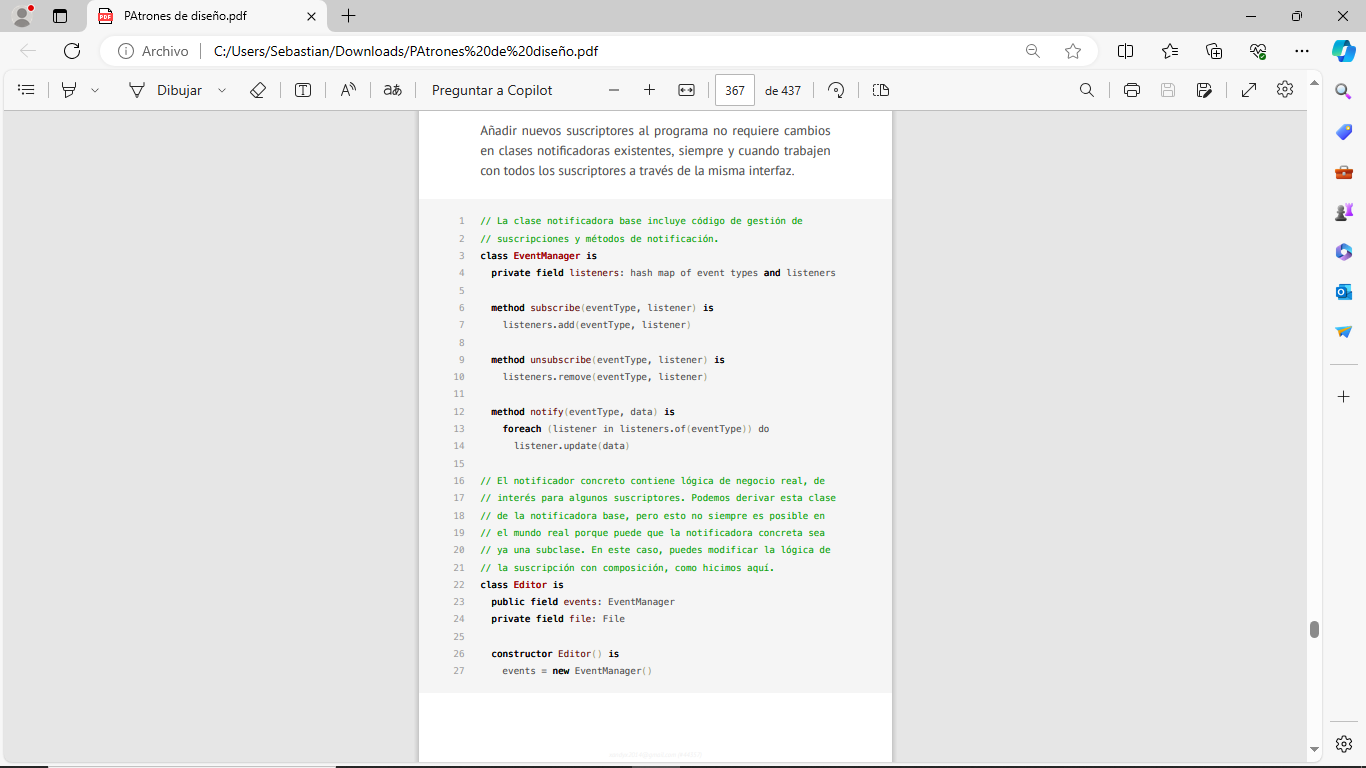
Si todos los suscriptores trabajan a través de la misma interfaz, añadir nuevos al programa no requeriría cambios en las clases notificadoras existentes.

****

En esta parte de código, tenemos la clase “EventManager” (Administrador de Eventos) que viene a ser el “Sujeto” en el patrón Observer. Desde esta clase se administra la suscripción, desuscripción y notificación de eventos a los observadores.

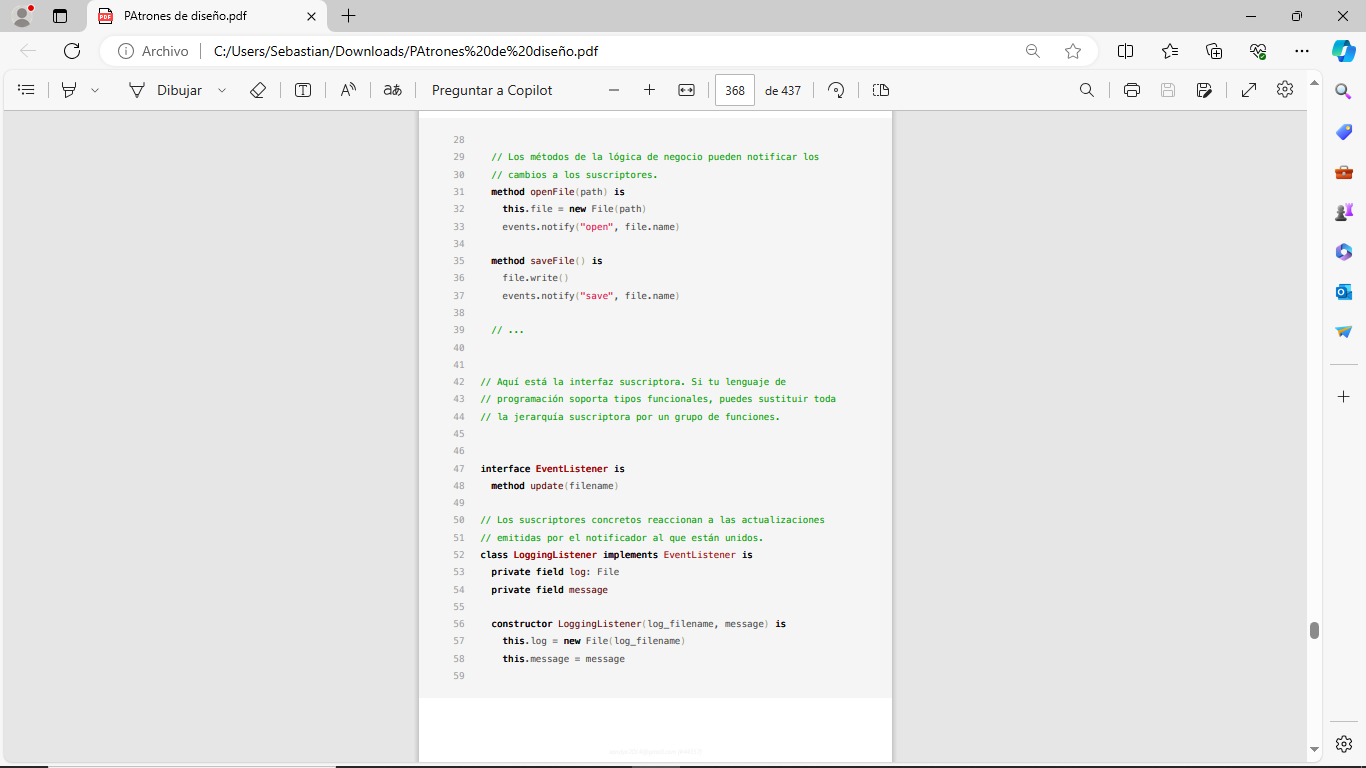
Tenemos un método de suscripción para agregar un observador para un tipo de evento especifico, un método “unsubscribe” para eliminar un observador de un tipo de evento especifico y un método de notificación para informar a todos los observadores registrados en un tipo de evento especifico los datos necesarios.

“Listeners” en la clase “EventManager” es un HashMap, mapea tipos de eventos a sus respectivos listeners. Al ser un campo privado, solo la propia clase puede acceder al HashMap, solo sus propios métodos pueden manipularlo. En el HashMap se almacena pares clave-valor, y en este caso, la clave seria el tipo de evento y el valor seria el Listener asociado al evento.

****

En esta sección de código, se define la clase “Editor”, la cual representa un editor de archivos. Podemos observar que tiene un campo público “events” que nos permite la gestión de eventos a través de una instancia de la clase “EventManager”; y un campo privado “file” donde almacenamos el archivo actualmente abierto en el editor.

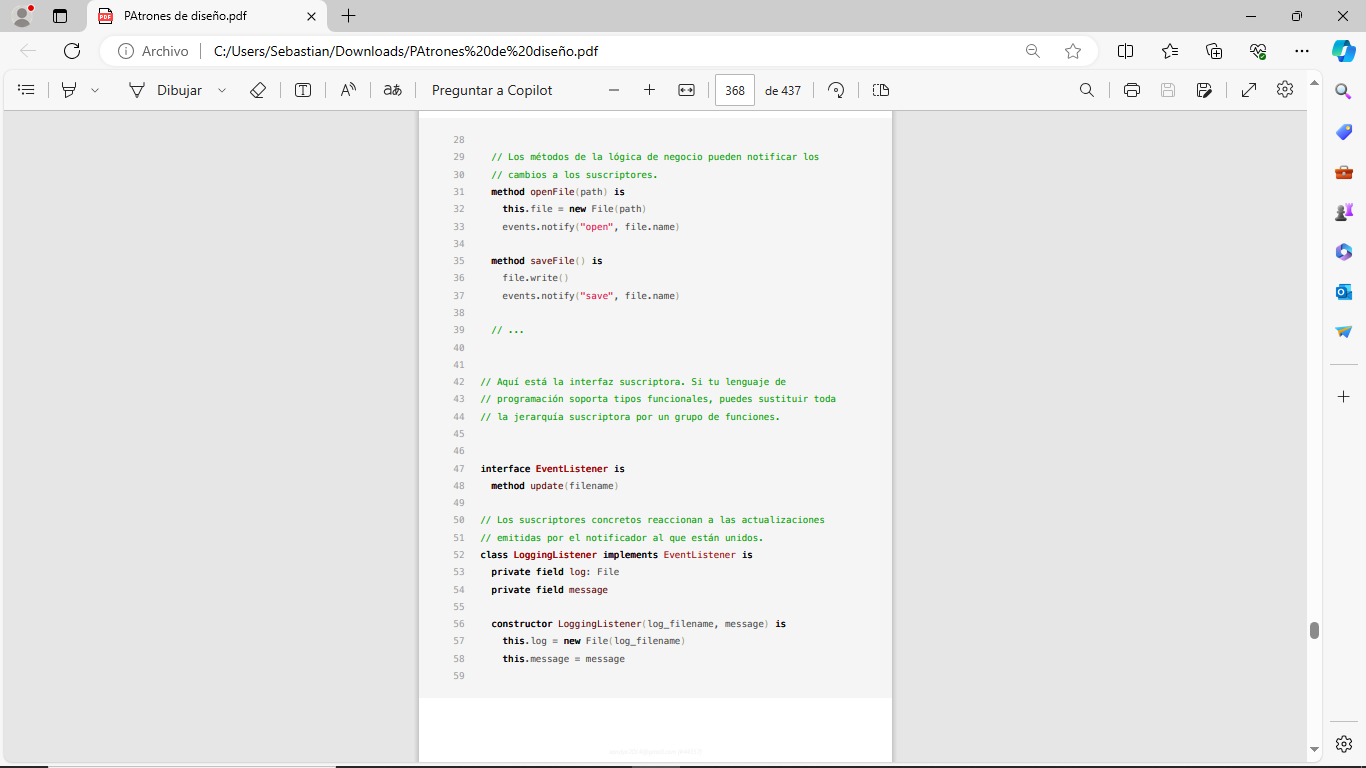
El “constructor” se llama cuando se crea una nueva instancia de “Editor”, dentro del constructor se inicializa el campo “events” creando así una nueva instancia de “EventManager”, lo cual nos garantiza que cada instancia de “Editor” tenga su propio administrador de eventos.

****

Acá vemos dos métodos importantes de la clase “Editor”. El método “openFile(path)” toma un argumento path, que básicamente es la ruta del archivo que se abrirá.

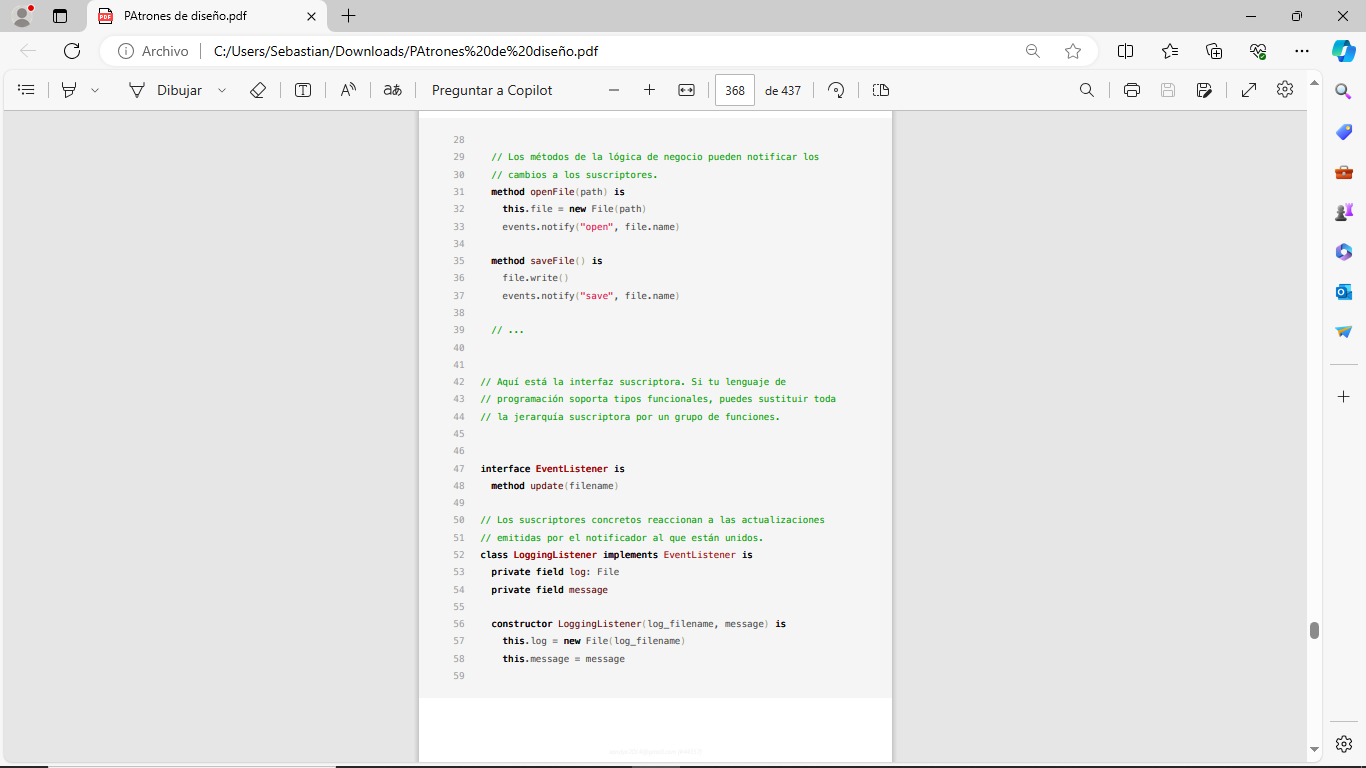
Dentro del método lo que se hace es crear una nueva instancia de “file” usando la ruta proporcionada, luego el campo “file” de la clase “Editor” se actualiza con esta nueva instancia de “file”. Por último, después de abrir el archivo, se utiliza el “EventManager” asociado ‘events’ para notificar a los observadores o listeners, que el archivo fue abierto. Se utiliza el tipo de evento “open” y se pasa el nombre del archivo con datos ‘file.name’.

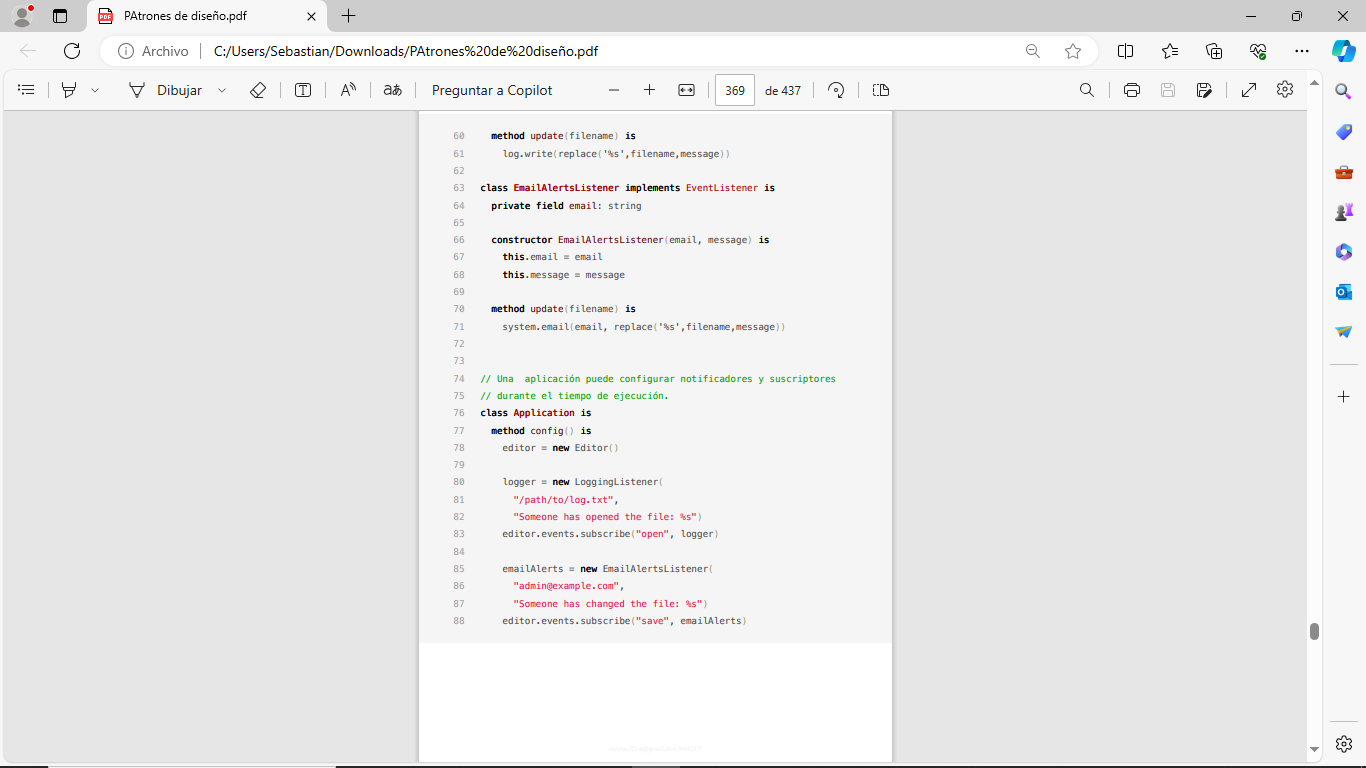
Para el caso del método “saveFile()”, guardará el archivo actualmente abierto. Llama al método ‘write()’ del objeto ‘file’ actual, que escribirá el contenido del archivo en el sistema de archivos. Por último, utiliza el “EventManager” asociado ‘events’ para notificar a los observadores que el archivo ha sido guardado. Se utiliza el tipo de evento “save” y se pasa el nombre del archivo con datos ‘file.name’.

****

Definimos una interfaz llamada “EventsListener” para definir una plantilla que sirva como modelo para cualquier clase que quiera actuar como un observador en el sistema de gestión de archivos.

El método “update(filename)” es el único definido en esta interfaz, se llama a este método cuando se notifica al observador sobre un evento, y toma un parámetro ‘filename’ que representa el nombre del archivo asociado al evento.

****

****

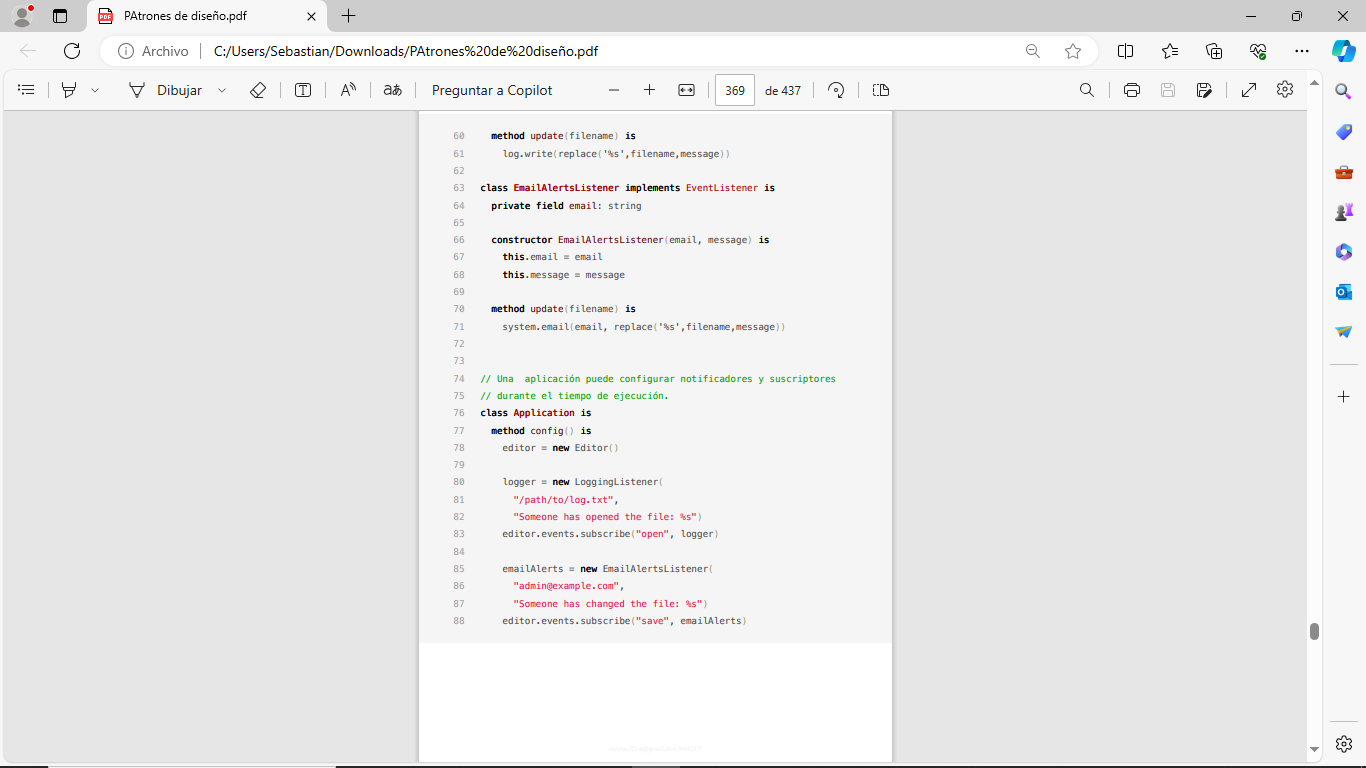
Por lo que podemos observar, se define una clase “LoggingListener” que implementa la interfaz “EventListener”.

Esta clase es un observador que es capaz de escribir registros de eventos en un archivo. Cuando se le notifica sobre un evento, utiliza la información proporcionada para generar un registro y lo escribe en el archivo de registro especificado.

* **“private field log: File”** es un campo privado que almacena una instancia de ‘File’ que se utiliza para escribir registros de eventos.
* **“private field message”** es un campo privado que almacena un mensaje predeterminado que será incluido en los registro de eventos.

Hay un constructor de la clase “loggingListener”, y se utiliza para inicializar los campos ‘log’ y ‘message’. Toma dos argumentos ‘log\_filename’ que es el nombre del archivo de registro y ‘message’ que es el mensaje predeterminado que se icluira en los registros de eventos.

El método “update(filename)” implementa el método ‘update()’ de la interfaz “EventListener” y se llama cuando se notifica al observador sobre un evento, toma un parámetro ‘filename’ que representa el nombre del archivo asociado al evento. En el método, se escribe un registro en el archivo de registro ‘log’ utilizando el método “write()”, bajo un formato de mensaje predeterminado con el nombre de archivo proporcionado en la notificación.

****

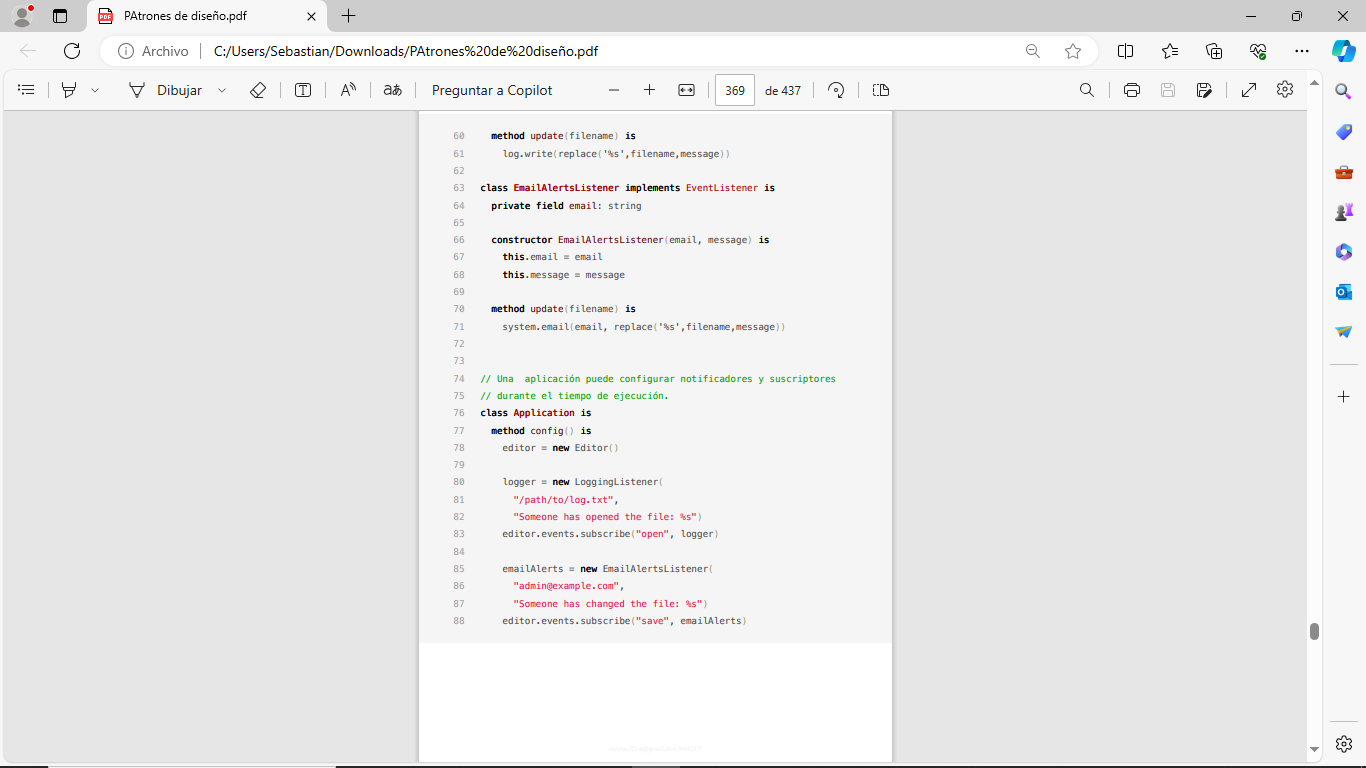
Hay una definición de otra clase “EmailAlertsListener” que también implementa la interfaz “EventListener”.

Esta clase es otro observador que es capaz de enviar alertas por correo electrónico cuando se le notifica sobre un evento. Utiliza la información proporcionada para generar el contenido del correo electrónico y lo envía a la dirección de correo electrónico especificada.

* **“private field email: string”** es un campo privado que almacena la dirección de correo electrónico a la que se enviaran las alertas.

Hay un constructor de la clase, que toma dos argumentos ‘email’ que es la dirección de correo electrónico a donde se enviaran las alertas, y ‘message’ que es el mensaje que se enviara en las alertas por correo electrónico.

El método “update(filename)” implementa el método ‘update()’ de la interfaz “EventListener”, y se lo llama cuando se notifica al observador sobre un evento, toma un parámetro ‘filename’ que representa el nombre del archivo asociado al evento. En el método se envía la alerta utilizando alguna función de sistema ‘system.email()’, y utiliza el formato de mensaje proporcionado con el nombre de archivo en la notificación.

****

Vemos definida una clase “Application” con un método “config()” que básicamente configura el editor y los observadores de registro de eventos y alertas por correo electrónico. Los observadores se suscriben a eventos específicos del editor y realizan acciones cuando se notifican estos eventos.

Se crea una instancia de “LoggingListener” para poder manejar los registros de eventos, le pasamos la ruta del archivo de registro y un mensaje que será utilizado para registrar eventos de apertura de archivo. Luego, suscribimos el observador ‘logger’ al evento “open” del editor. Cada vez que se abra un archivo en el editor, el observador será notificado.

Se crea una instancia de “EmailAlertsListener” para poder manejar las alertas por correo electrónico, le pasamos la dirección de correo electrónico a la que se enviaran las alertas y un mensaje que se utilizara para enviar alertas sobre cambios en el archivo. Por utlimo se suscribe al observador “emailAlerts” al evento “save” del editor, entonces cada vez que se guarde un archivo en el editor, el observador será notificado.

## APLICABILIDAD

Se recomienda utilizar éste patrón en casos como:

* 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 de manera dinámica.

Este problema se puede experimentar a menudo al trabajar con clases de la interfaz gráfica de usuario, un ejemplo seria: creamos clases personalizadas de botón y queremos permitir al cliente colgar código cliente de nuestros botones para que se active cuando un usuario pulsa un botón. Este patrón, permite que cualquier objeto que implemente la interfaz suscriptora pueda suscribirse a notificaciones de eventos en objetos notificadores.

Podemos con esto añadir el mecanismo de suscripción a nuestros botones, y permitirle al cliente acoplar su código personalizado a través de clases suscriptoras personalizadas.

* Cuando objetos de mi aplicación deban observar a otros durante un tiempo limitado o en casos específicos.

En este caso tenemos una lista de suscripción dinámica, implica que los suscriptores pueden unirse o abandonar la lista cuando lo deseen.

## IMPLEMENTACION

Para la implementación del patrón “Observer”, seguiremos algunos pasos:

1. Es necesario que repasemos la lógica del negocio y la dividamos en dos partes: por un lado la funcionalidad central (será independiente del resto del código) va a actuar como notificador, todo lo restante se convertirá en un grupo de clases suscriptoras.
2. Debemos declarar la interfaz suscriptora, y al menos un método actualizar.
3. Debemos declarar la interfaz notificadora, y describir métodos para añadir o eliminar de la lista a un objeto suscriptor. (notificadores y suscriptores deben trabajar únicamente a través de la interfaz suscriptora)
4. Es necesario decidir dónde colocar la lista de suscripción, y la implementación de los métodos de suscripción. Lo normal sería colocarla en una clase abstracta que derive directamente de la interfaz notificadora. Cada notificador concreto extienden esa clase, heredando el comportamiento de suscripción.

Si aplicamos el patrón a una jerarquía de clases ya existentes, lo recomendable seria colocar la lógica de la suscripción en un objeto separado, y que todos los notificadores reales la utilicen.

1. Debemos crear notificadoras concretas, y cada vez que suceda algo “importante” dentro de una notificadora, se deberá notificar a todos sus suscriptores.
2. Debemos implementar métodos de notificación de actualizaciones en las clases suscriptoras concretas, estas necesitaran cierta información de contexto sobre el evento que puede pasarse como argumento del método de notificación. También podría el suscriptor extraer la información directamente de ella al recibir una notificación, para esto el notificador debe pasarse a sí mismo a través del método de actualización.
3. El cliente debe crear todos los suscriptores necesarios y registrarlos con los notificadores adecuados.

## PROS Y CONTRAS DEL PATRON

Entre los pros, tenemos:

* Principio de abierto/cerrado: Podemos introducir nuevas clases suscriptoras sin tener que cambiar el código de la clase notificadora, y de manera inversa si existe una interfaz notificadora
* Podemos establecer relaciones entre objetos durante el tiempo de ejecución
* La contra podría ser: que los suscriptores son notificados en un orden aleatorio

## RELACION CON OTROS PATRONES DE COMPORTAMIENTO

Patrones como “Command”, “Mediator”, “Chain of Responsibility”, y el de estudio “Observer”, tienen distintas formas de conectar emisores y receptores de solicitudes:

* “Chain of Responsibility” pasa una solicitud secuencialmente a lo largo de una cadena dinámica de receptores potenciales hasta que uno de ellos la gestiona.
* “Command” establece conexiones unidireccionales entre emisores y receptores.
* “Mediator” elimina las conexiones directas entre emisores y receptores, forzándolos a comunicarse indirectamente a través de un objeto mediador.
* “Observer” permite a los receptores suscribirse o darse de baja dinámicamente a la recepción de solicitudes.

Los patrones Mediator y Observer pueden ser aplicados ambos a la vez, o por separado. La meta del patrón Mediator consiste en eliminar las dependencias mutuas que existen entre un grupo de componentes del sistema, dependiendo solamente de un único objeto mediador, en cambio, la meta del patrón Observer es establecer conexiones dinámicas de un único sentido entre objetos, donde uno de estos actúa como subordinados de otros.

El Mediator puede asemejarse mucho al Observer cuando el objeto mediador juega el papel del notificador, y los componentes actúan como suscriptores que se suscriben o se dan de baja de los eventos del mediador.

# CONCLUSIÓN

Los patrones de diseño de software son herramientas invaluablemente poderosas en el arsenal de cualquier desarrollador. Proporcionan soluciones probadas y eficientes para problemas comunes de diseño, promoviendo la coherencia, la reutilización y la escalabilidad del código. Al adoptar patrones de diseño, los equipos de desarrollo pueden mejorar la calidad del software, reducir costos y tiempos de desarrollo, y facilitar la colaboración entre miembros del equipo. Sin embargo, es importante recordar que los patrones de diseño no son una solución universal para todos los problemas de diseño de software, y su aplicación debe ser cuidadosamente considerada en el contexto específico de cada proyecto. Con una comprensión sólida de los patrones de diseño y su aplicación adecuada, los desarrolladores pueden construir sistemas de software más robustos, flexibles y fáciles de mantener a lo largo del tiempo.

# ESPECIFICACION DE CONFIABILIDAD Y SEGURIDAD

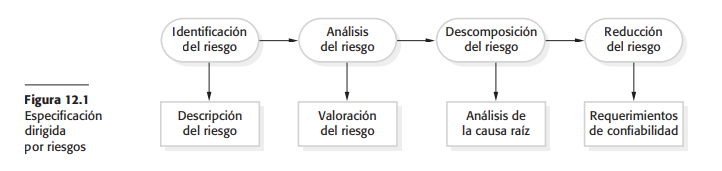
## Introducción

La confiabilidad del sistema no sólo depende de buena ingeniería, sino también requiere de la atención a los detalles cuando se derivan los requerimientos del sistema y la inclusión de requerimientos especiales de software que se ajustan para garantizar la confiabilidad y seguridad de un sistema. Estos requerimientos de confiabilidad y seguridad son de dos tipos:

* Requerimientos funcionales, definen mecanismos de comprobación y recuperación que deben incluirse en el sistema y en las características que ofrecen protección contra fallas de sistema y ataques externos.
* Requerimientos no funcionales, definen la confiabilidad y disponibilidad requeridas del sistema.

## Especificación de requerimientos

Los requerimientos de confiabilidad y seguridad pueden considerarse como requerimientos de protección. Especifican cómo un sistema debe protegerse a sí mismo de fallas internas, detener fallas de sistema que causen daño a su entorno, contener los accidentes o ataques del entorno que dañen el sistema, así como facilitar la recuperación en el caso de falla.



Las etapas de este proceso son:

1. **Identificación del riesgo**: Se identifican los riesgos potenciales al sistema. Éstos dependen del entorno en que se usa el sistema.
2. **Análisis y clasificación del riesgo**: Cada riesgo se considera por separado. En esta etapa, los riesgos pueden eliminarse porque es improbable que surjan o porque no se pueden detectar con el software
3. **Descomposición del riesgo**: Cada riesgo se analiza para descubrir las causas raíz potenciales de dicho riesgo. Pueden ser errores de software, hardware o vulnerabilidades inherentes que son consecuencia de decisiones de diseño del sistema.
4. **Reducción del riesgo**: Se hacen proposiciones de formas para reducir o eliminar los riesgos identificados.

Para sistemas grandes, el análisis de riesgo puede estructurarse en fases, donde cada fase considera diferentes tipos de riesgos:

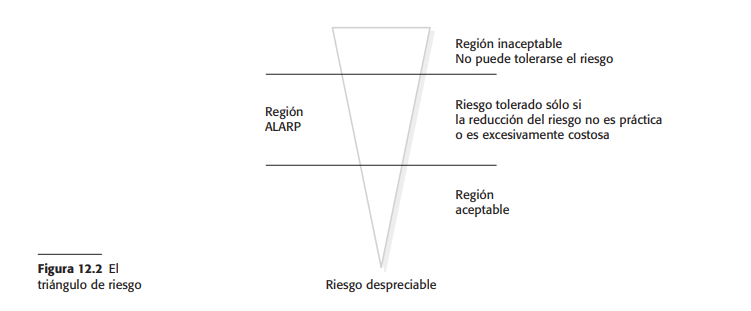
1. **Análisis preliminar del riesgo**: en él que se identifican los principales riesgos del entorno del sistema.
2. **Análisis de riesgo de ciclo de vida**: se dirige principalmente a los riesgos surgidos por decisiones de diseño del sistema. En esta etapa, se deben extender los requerimientos para protegerse contra estos riesgos.
3. **Análisis de riesgo operativo**: el cual se preocupa por la interfaz de usuario del sistema y los riesgos de error del operador.

## Especificación de Protección

La preocupación principal de la especificación de protección es identificar los requerimientos que reducirán la probabilidad de que ocurran tales fallas de sistema.

Al derivar requerimientos de protección, se necesita encontrar un equilibrio aceptable entre seguridad y funcionalidad para evitar la sobreprotección. Por consiguiente, la especificación de protección se enfoca usualmente en los riesgos que surgen en una situación dada, y los eventos que conducen a dichos peligros.

* **Identificación del Peligro**: Es posible tratar el problema de identificación del riesgo al considerar los diferentes tipos, como los físicos, eléctricos, biológicos, de radiación, de falla de servicio, etcétera. Entonces, cada una de estas clases puede analizarse para descubrir peligros específicos que podrían ocurrir. También deben identificarse aquellas posibles combinaciones de riesgos que sean potencialmente peligrosas.
* **Valoración de Peligro**: Se enfoca en comprender la probabilidad de que sobrevenga un peligro y las consecuencias de que ocurriera un accidente o incidente asociado con dicho peligro. Es necesario hacer este análisis para comprender si un peligro constituye una seria amenaza al sistema o el entorno. El análisis proporciona también una base para decidir sobre cómo manejar el riesgo asociado.



Para cada peligro, el resultado del proceso de análisis y clasificación es un enunciado de aceptabilidad. Esto se expresa en términos de riesgo, donde el riesgo considera la probabilidad de un accidente y sus consecuencias.

Existen tres categorías que se utilizan en la valoración del riesgo:

1. Los riesgos intolerables en los sistemas críticos de protección son aquellos que amenazan la vida humana.
2. Los riesgos “tan bajos como sea razonablemente práctico” (ALARP, por las siglas de As Low As Reasonably Practical) son aquellos que tienen consecuencias menos serias o que, aun cuando son graves, tienen una muy baja probabilidad de ocurrencia.
3. Los riesgos aceptables son aquellos en que los accidentes asociados derivan por lo general en un daño menor.

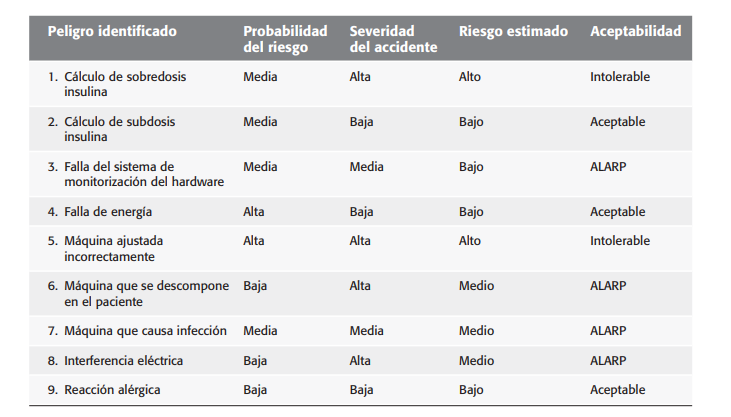
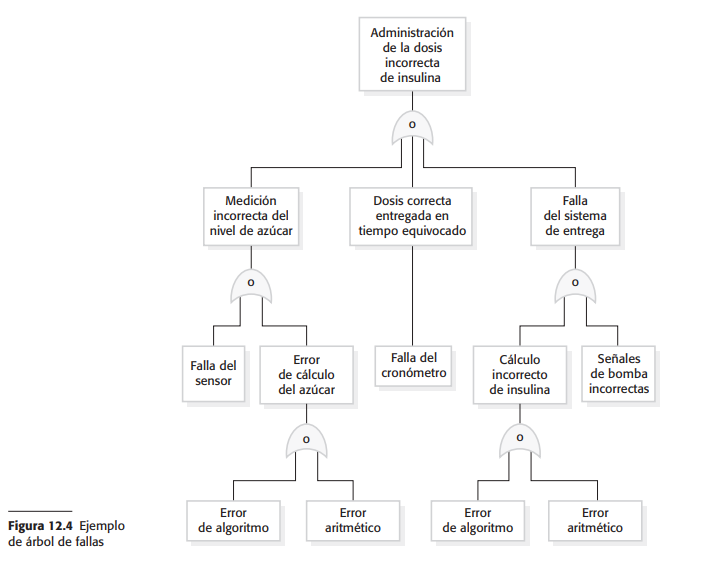


Figura 12.3 Clasificación del riesgo para la bomba de insulina

El software que monitoriza el hardware debe monitorizar el estado del sistema y advertir acerca de problemas potenciales. Con frecuencia, las advertencias permitirán la detección del peligro antes de que ocurra un accidente.

* **Análisis del Peligro**: Es el proceso que descubre las causas raíz de los peligros en un sistema de protección crítico. Su meta es detectar qué eventos o combinaciones de eventos causarían una falla de sistema que derive en un peligro.

Se han planteado varias técnicas como posibles enfoques para la descomposición o el análisis del peligro, una de ellas es el análisis de árbol de fallas (Leveson y Stolzy, 1987; Storey, 1996).



* **Reducción del Peligro**: Una vez identificados los riesgos potenciales y sus causas raíz, entonces se podrán derivar requerimientos de seguridad que administren los riesgos y garanticen que no ocurran los incidentes o accidentes.

Existen tres posibles estrategias por utilizar:

1. **Evitar el peligro**: El sistema se diseña de modo que no pueda ocurrir el peligro.
2. **Detectar y eliminar el peligro**: El sistema se diseña de forma que los peligros se detecten y neutralicen antes de que suceda un accidente.
3. **Limitar el daño**: El sistema se diseña de manera que se minimicen las consecuencias de un accidente.

En un sistema crítico de seguridad, los peligros intolerables pueden manejarse al disminuir su probabilidad y agregar un sistema de protección que brinde un respaldo de seguridad.

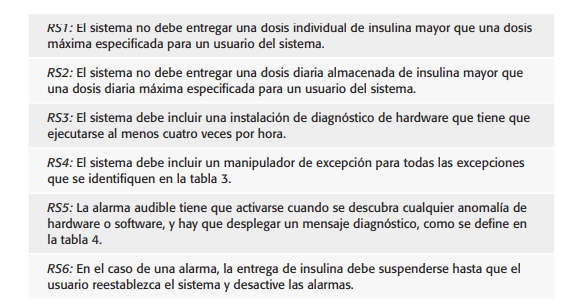


Figura 12.5 Ejemplos de requerimientos de seguridad

## Especificación de Fiabilidad

La fiabilidad global de un sistema depende de la fiabilidad del hardware, la fiabilidad del software y la fiabilidad de los operadores del sistema. Se debe tomar en cuenta el software del sistema y, además, de incluir requerimientos que compensen la falla de software. También puede haber requerimientos relacionados de fiabilidad para ayudar a detectar y recuperar fallas de hardware y errores del operador.

En contraste, seguridad y protección tratan de evitar situaciones indeseables, en vez de especificar un “nivel” deseado de seguridad o protección.

Por consiguiente, los requerimientos de fiabilidad son de dos tipos:

1. **Requerimientos no funcionales:** definen el número de fallas aceptables durante el uso normal del sistema, o el tiempo en que el sistema no está disponible para su uso. Se trata de requerimientos de fiabilidad cuantitativos.
2. **Requerimientos funcionales**: definen las funciones del sistema y el software que evitan, detectan o toleran fallas del software y, de ese modo, aseguran que esto no conduzca a fallas de sistema.

Los requerimientos de fiabilidad cuantitativa conducen a requerimientos de sistema funcionales relacionados. Para lograr cierto nivel de fiabilidad requerido, los requerimientos funcionales y de diseño del sistema deben especificar las fallas a detectar y las acciones que deben tomarse para garantizar que éstas no conduzcan a fallas de sistema.

En la figura 12.1 se muestra que el proceso de especificación de fiabilidad puede basarse en el proceso general de especificación dirigido por riesgo:

1. **Identificación del riesgo**: En esta etapa se examinan los tipos de fallas de sistema que originarían pérdidas económicas de cierto tipo.
2. **Análisis del riesgo**: Implica la estimación de los costos y las consecuencias de diferentes tipos de fallas de software y selecciona para un análisis ulterior las fallas de graves consecuencias.
3. **Descomposición de riesgo**: En esta etapa, se realiza un análisis de la causa raíz de las graves y probables fallas de sistema.
4. **Reducción del riesgo:** En esta etapa, deben generarse especificaciones cuantitativas de fiabilidad que establezcan las probabilidades aceptables de los diferentes tipos de fallas.

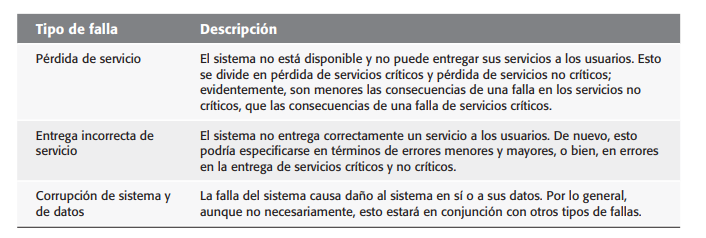


Figura 12.6 Tipos de fallas del sistema

* **Métricas de Fiabilidad**: La fiabilidad puede especificarse como una probabilidad de que una falla del sistema ocurrirá cuando un sistema está en uso dentro de un entorno operacional especificado.

Existen dos importantes métricas que se usan para especificar la fiabilidad, además de una métrica adicional que se utiliza para especificar el atributo de sistema relacionado con la disponibilidad. La elección de métrica depende del tipo de sistema especificado y de los requerimientos del dominio de la aplicación. Las métricas son:

1. **Probabilidad de falla a pedido** **(POFOD, por las siglas de Probability Of Failure On Demand):** De este modo, POFOD = 0.001 significa que hay una probabilidad de 1/1,000 de que ocurra una falla al hacer una petición.
2. **Tasa de ocurrencia de fallas (ROCOF, por las siglas de Rate Of Occurrence Of Failures):** Esta métrica establece el número probable de fallas de sistema que se observan en relación con cierto tiempo o el número de ejecuciones del sistema. El MTTF es el promedio de unidades de tiempo entre las fallas observadas de sistema.
3. **Disponibilidad (AVAIL):** AVAIL es la probabilidad de que un sistema esté en operación cuando se haga una demanda por servicio. En consecuencia, una disponibilidad de 0.9999 significa que, en promedio, el sistema estará disponible el 99.99% del tiempo de operación.

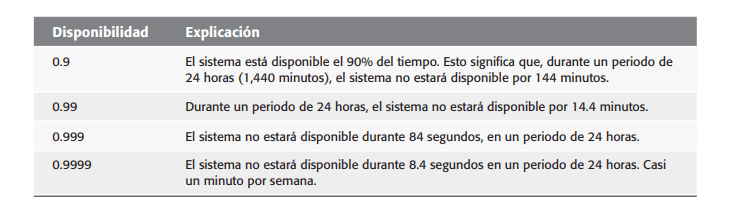


Figura 12.7 Especificación de disponibilidad

**La POFOD** debe usarse como una métrica de fiabilidad en situaciones donde una falla sobre demanda podría conducir a una falla grave del sistema. Esto se aplica sin importar la frecuencia de las demandas.

**La ROCOF** es la métrica más adecuada para usar en situaciones donde las demandas al sistema se hacen con regularidad en vez de esporádicamente.

Las unidades de tiempo que pueden usarse son tiempo de calendario, tiempo de procesador o una unidad discreta como el número de transacciones.

* **Requerimientos de Fiabilidad no Funcionales**: son especificaciones cuantitativas de la fiabilidad y disponibilidad requeridas de un sistema.

Para establecer el nivel requerido de fiabilidad del sistema, hay que considerar las pérdidas asociadas que resultarían de una falla del sistema.

El problema con la especificación de la fiabilidad usando métricas como POFOD, ROCOF y AVAIL es que es posible sobre especificar la fiabilidad y, por ende, incurrir en altos costos de desarrollo y validación.

Si la fiabilidad se especifica como una métrica, evidentemente es importante valorar que se logró el nivel de fiabilidad requerido. Esta valoración se hace como parte de las pruebas del sistema. Para valorar estadísticamente la fiabilidad de un sistema, hay que observar algunas fallas.

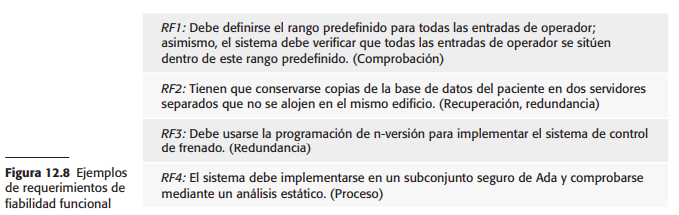
Las organizaciones deben ser realistas sobre si vale la pena especificar y validar un nivel de fiabilidad muy alto.

* **Especificación de fiabilidad funcional**: Incluye identificar los requerimientos que definen las restricciones y las características que contribuyen a la fiabilidad del sistema.

Existen tres tipos de requerimientos de fiabilidad funcional para un sistema:

1. **Requerimientos de comprobación**: Tales requerimientos identifican las comprobaciones de las entradas al sistema, para garantizar que las entradas incorrectas o fuera de rango se detecten antes de que las procese el sistema.
2. **Requerimientos de recuperación**: Dichos requerimientos se implementan para ayudar al sistema a recuperarse luego de que ocurre una falla.
3. **Requerimientos de redundancia**: Especifican las características redundantes del sistema que aseguran que la falla en un solo componente no conduzca a una pérdida completa del servicio.

Además, los requerimientos de fiabilidad pueden incluir requerimientos de proceso para fiabilidad. Se trata de requerimientos que aseveran que la buena práctica, conocida porque reduce el número de fallas de un sistema, se usa en el proceso de desarrollo.



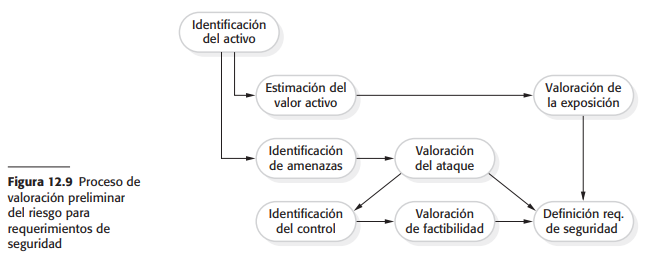
## Especificación de Seguridad

La seguridad es un problema que exige mayor esfuerzo que la protección, por diversas razones.

Tales distinciones significan que los requerimientos de seguridad suelen ser más extensos que los requerimientos de protección. Los requerimientos de protección se enfocan en la comprobación de problemas y en la toma de acciones si ocurren dichos problemas.

El proceso de análisis y valoración del riesgo, que se estudia en la sección 12.1, puede utilizarse para identificar requerimientos de seguridad del sistema. Existen tres etapas en este proceso:

1. **Análisis preliminar del riesgo**: En esta etapa todavía no se toman decisiones sobre los requerimientos detallados del sistema, el diseño del sistema o la tecnología de implementación. La meta de este proceso de valoración es derivar requerimientos de seguridad para el sistema en su conjunto.
2. **Análisis del riesgo del ciclo de vida**: Esta valoración de riesgo tiene lugar durante el ciclo de vida de desarrollo del sistema, después de tomarse elecciones de diseño. Los requerimientos adicionales de seguridad toman en cuenta las tecnologías usadas en la construcción del sistema, así como las decisiones de diseño e implementación del sistema.
3. **Análisis del riesgo operativo**: Esta valoración de riesgo considera los riesgos al sistema operativo impuestos por ataques maliciosos de los usuarios, con o sin conocimiento interno del sistema.



Una importante entrada al proceso de valoración y gestión del riesgo es la política de seguridad de la organización. Una política de seguridad organizacional se aplica a todos los sistemas y en ella se establece lo que debe y no debe permitirse. La política de seguridad establece condiciones que un sistema de seguridad debe mantener siempre, ya que ayudan a identificar amenazas que se puedan presentar. Las amenazas son cualquier intrusión que vulnere la seguridad de la empresa.

## Especificación Formal

Son aproximaciones con base matemática al desarrollo del software, donde se define un modelo formal del software.

La especificación formal es una descripción sin ambigüedades de qué debe hacer el sistema. Al usar métodos manuales o soportados por herramientas, es posible comprobar que el comportamiento de un programa es congruente con la especificación.

Las especificaciones formales no sólo son importantes para una verificación del diseño e implementación del software, sino también son la forma más precisa de especificar sistemas y, por ende, de reducir el ámbito para las malas interpretaciones.

Las especificaciones formales se desarrollan con frecuencia como parte de un proceso de software basado en un plan, en que el sistema se especifica completamente antes de su desarrollo. Los requerimientos del sistema y el diseño se definen a detalle, y se analizan y comprueban cuidadosamente antes de comenzar la implementación.

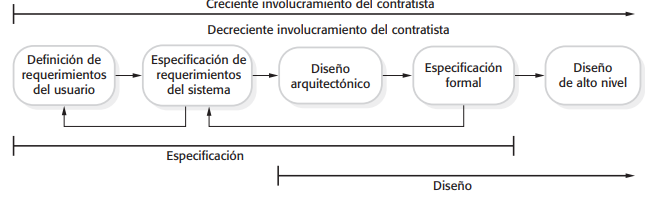


Figura 12.12 Especificación formal en un proceso de software basado en un plan.

Las ventajas de desarrollar una especificación formal y de usar ésta en un proceso de desarrollo formal son:

* Mientras se desarrolla una especificación formal a detalle, se obtiene una comprensión profunda y pormenorizada de los requerimientos del sistema.
* Conforme la especificación se expresa en un lenguaje con semántica definida formalmente, puede analizarse de manera automática para descubrir inconsistencias y aquello que no se completó.
* Si se usa un método, se puede transformar la especificación formal en un programa, a través de una secuencia de transformaciones que preserven la exactitud. En consecuencia, se garantiza que el programa resultante cumpla su especificación.
* Los costos de las pruebas del programa suelen reducirse porque el programa se verificó contra su especificación.

Los argumentos que se esgrimen contra el desarrollo de una especificación formal del sistema son:

* El problema de que los propietarios y expertos de dominio no entiendan una especificación formal, de modo que no pueden comprobar que representa con precisión sus requerimientos.
* Es bastante sencillo cuantificar los costos de crear una especificación formal, pero es más difícil estimar el posible ahorro en los costos que resultará de su uso. Como resultado, los administradores no están dispuestos a asumir el riesgo de adoptar este enfoque

# Conclusión

La especificación de confiabilidad y seguridad en el desarrollo de software es fundamental para garantizar la integridad, disponibilidad y protección de los sistemas informáticos y los datos sensibles. Al definir y establecer requisitos claros y precisos de confiabilidad y seguridad desde las etapas iniciales del proceso de desarrollo, las organizaciones pueden mitigar riesgos, prevenir vulnerabilidades y garantizar la fiabilidad operativa de sus sistemas. Además, una especificación adecuada proporciona una base sólida para la implementación de prácticas de diseño y codificación seguras, así como para la realización de pruebas exhaustivas de seguridad y confiabilidad. En última instancia, la especificación de confiabilidad y seguridad no solo fortalece la resistencia de los sistemas informáticos frente a amenazas y fallos, sino que también promueve la confianza del usuario y protege la reputación de la organización.

## BIBLIOGRAFÍA

Shvets, Alexander (2019). *Sumérgete en los patrones de diseño.* Refactoring.Guru

Sommerville, Ian (9ª Edicion) (2011). *Ingeniería de Software.* Pearson Education.

<https://programacion.net/articulo/patrones-de-diseno-v:-patrones-de-creacion---prototipo_1005>