

UCP

UNIVERSIDAD DE LA CUENCA DEL PLATA

INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Ing de software II

Patrones de diseño

**Autor:** *Kugelmann Tomas - Benitez Pablo*

**Profesor:** Gabriel Kutz

**Año:** 2024

# 

[**Actividad 1 3**](#_izt05wwhwm5b)

[Builder method 3](#_o6gaj3fs6vdf)

[Introducción 3](#_fin1erzeie6n)

[Desarrollo 3](#_1nlxmfwubah)

[Estructura 5](#_oc8me144j31l)

[Ventajas 6](#_j6y3hd27kdeh)

[Desventajas 6](#_8kgrc3g86ord)

[Ejemplo 6](#_58lk3cy5vbc)

[Singleton (Instancia Única). 7](#_98fa0zs6nnn9)

[Introducción 7](#_gpgqanqjn6s0)

[Desarrollo 7](#_oi2zhljk59u4)

[Estructura 8](#_1o6xzodd7k73)

[Ventajas 10](#_t0oiset196u7)

[Desventajas 10](#_wrnb79j5tsva)

[Ejemplo 10](#_693nwvo5f52u)

[**Actividad 2 11**](#_idprgj4t00m9)

[Especificación dirigida por riesgos 11](#_516gzvdsvvd)

[Especificación de Protección 12](#_m7axkxmb670l)

[Especificación de Fiabilidad 12](#_dp4x8zgfgyvl)

[Especificación de Seguridad 13](#_uxjzyomoszhu)

[Especificación Formal 17](#_6ouj7he03tl0)

# 

# 

# 

# Actividad 1

## Builder method

### Introducción

En el desarrollo de software, la creación de objetos complejos puede volverse problemática cuando se requiere manejar múltiples configuraciones o representaciones del mismo objeto. El patrón Builder emerge como una solución para este problema, permitiendo la construcción paso a paso de objetos complejos mientras oculta la complejidad de su creación al cliente.

### Desarrollo

Builder es un patrón de diseño creacional que nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción

Imagina un objeto complejo que requiere una inicialización laboriosa, paso a paso, de muchos campos y objetos anidados. Normalmente, este código de inicialización está sepultado dentro de un monstruoso constructor con una gran cantidad de parámetros o peor aún: disperso por todo el código cliente.

La solución más sencilla es extender la clase base Auto y crear un grupo de subclases que cubran todas las combinaciones posibles de los parámetros. Pero, en cualquier caso, acabarás con una cantidad considerable de subclases. Cualquier parámetro nuevo, como el tamaño de las ruedas, exigirá que incrementes esta jerarquía

En la mayoría de los casos, gran parte de los parámetros no se utilizará, lo que provocará que las llamadas al constructor sean bastante feas.

Crear una subclase por cada configuración posible de un objeto puede complicar demasiado el programa. Por ejemplo, pensemos en cómo crear un objeto Auto . Para construir un auto, debemos construir el chasis, las ruedas, el motor y la caja. Pero ¿qué pasa si quieres un auto distinto (At o una SUV o camioneta)?

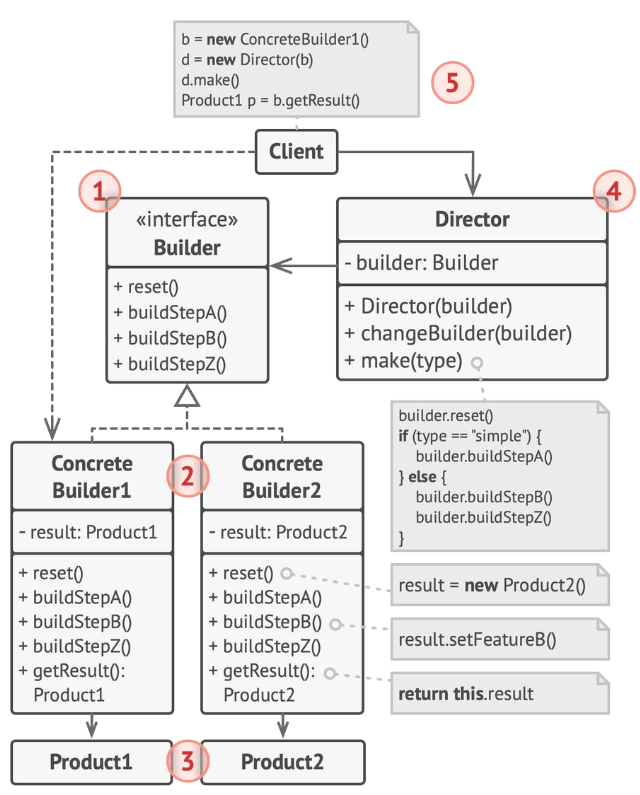
El patrón Builder sugiere que saques el código de construcción del objeto de su propia clase y lo coloques dentro de objetos independientes llamados constructores. Este no permite a otros objetos acceder al producto mientras se construye y organiza la construcción de objetos en una serie de pasos ( AddWheels , AddEngine , etc.). Para crear un objeto, se ejecuta una serie de estos pasos en un objeto constructor. Lo importante es que no necesitas invocar todos los pasos. Puedes invocar sólo aquellos que sean necesarios para producir una configuración particular de un objeto. Puede ser que algunos pasos de la construcción necesiten una implementación diferente cuando tengamos que construir distintas representaciones del producto. Por ejemplo, las ruedas de una camioneta pueden tener distinta anchura, alto y diámetro que las de un hatchback(En la mayoría de los casos). En este caso, podemos crear varias clases constructoras distintas que implementen la misma serie de pasos de construcción, pero de forma diferente entonces podemos utilizar estos constructores en el proceso de construcción cada uno con sus propias especificaciones(205/55 R16, 195/65 R15, y 175/65 R14)

Sin embargo, esto sólo funcionará si el código cliente que invoca los pasos de construcción es capaz de interactuar con los constructores mediante una interfaz común denominada “Clase directora”

La clase directora define el orden en el que se deben ejecutar los pasos de construcción, mientras que el constructor proporciona la implementación de dichos pasos. Esta clase directora sabe qué pasos de construcción ejecutar para lograr un producto que funcione.

No es estrictamente necesario tener una clase directora en el programa, ya que se pueden invocar los pasos de construcción en un orden específico directamente desde el código cliente. No obstante, la clase directora puede ser un buen lugar donde colocar distintas rutinas de construcción para poder reutilizarlas a lo largo del programa. Además, la clase directora esconde por completo los detalles de la construcción del producto al código cliente, quien sólo necesita asociar un objeto constructor con una clase directora, utilizarla para iniciar la construcción, y obtener el resultado del objeto constructor

### Estructura



1. La interfaz Constructora declara pasos de construcción de producto que todos los tipos de objetos constructores tienen en común.

2. Los Constructores Concretos ofrecen distintas implementaciones de los pasos de construcción. Los constructores concretos pueden crear productos que no siguen la interfaz común.

3. Los Productos son los objetos resultantes. Los productos construidos por distintos objetos constructores no tienen que pertenecer a la misma jerarquía de clases o interfaz.

4. La clase Directora define el orden en el que se invocarán los pasos de construcción, por lo que puedes crear y reutilizar configuraciones específicas de los productos.

5. El Cliente debe asociar uno de los objetos constructores con la clase directora. Normalmente, se hace una sola vez mediante los parámetros del constructor de la clase directora, que utiliza el objeto constructor para el resto de la construcción. No obstante, existe una solución alternativa para cuando el cliente pasa el objeto constructor al método de producción de la clase directora. En este caso, puedes utilizar un constructor diferente cada vez que produzcas algo con la clases

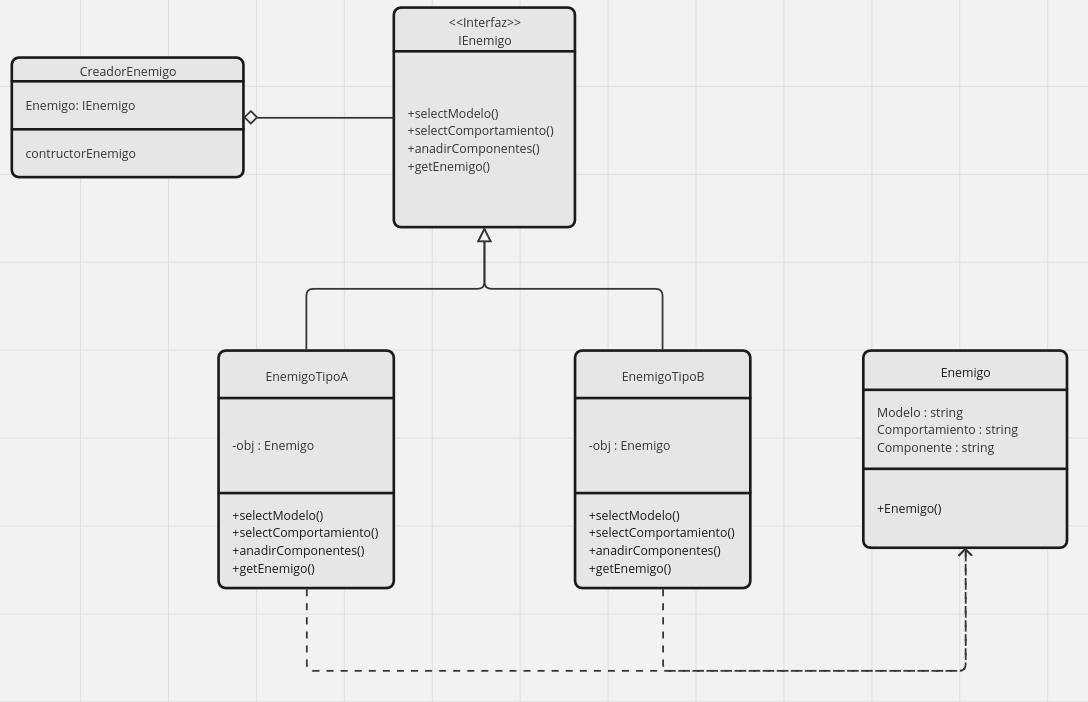
### Ventajas

La construcción y la representación (salida) se incorporan por separado. Las representaciones internas del constructor están ocultas para el director. Las nuevas representaciones como tal pueden integrarse fácilmente utilizando clases de constructores concretos. El proceso de construcción lo controla explícitamente el director. Si hay que hacer cambios, pueden hacerse sin consultar al cliente.

### Desventajas

El patrón Constructor consta de un fuerte vínculo entre el producto, el constructor específico y las clases del proceso de diseño, así que puede ser difícil hacer cambios en el proceso básico. La construcción de los objetos requiere conocer su uso y su entorno concretos. Utilizar patrones conocidos, como el patrón de diseño Builder, puede hacer que los programadores pasen por alto soluciones más sencillas y elegantes. En el fondo, muchos desarrolladores consideran que este es uno de los patrones de diseño menos importantes.

### Ejemplo



1. **Clase Enemigo:**
   * Esta clase representa el objeto complejo que queremos construir.
   * Contiene propiedades públicas que serán configuradas por los Builders.
2. **Interfaz Enemigo:**
   * Es una interfaz que define los métodos que serán implementados por las clases que construyen los diferentes tipos de enemigos.
   * Incluye un método GetEnemy que devuelve una instancia de la clase Enemigo después de completar el proceso de construcción.
3. **Clase EnemigoTipoA y EnemigoTipoB:**
   * Son clases concretas que implementan la interfaz IEnemigo.
   * Cada una tiene una implementación única de los métodos definidos en la interfaz.
   * Permiten crear instancias con propiedades específicas para cada tipo de enemigo.
4. **Clase CreadorEnemigo:**
   * Actúa como el Director en el patrón Builder.
   * Contiene la lógica para crear nuevos enemigos.
   * En su constructor, recibe una instancia de IEnemigo, permitiendo la creación de diferentes tipos de enemigos a través de la misma lógica.

## Singleton (Instancia Única).

### Introducción

La idea principal detrás del patrón Singleton es restringir la creación de objetos de una clase a una sola instancia, y luego proporcionar un mecanismo para acceder a esa instancia en cualquier parte del programa. Esto puede ser útil en situaciones donde solo se requiere una instancia de una clase para gestionar recursos compartidos, como configuraciones, conexiones a bases de datos, registros, entre otros.

### Desarrollo

El patrón Singleton surge para resolver dos problemas a la vez, pero vulnera el principio de responsabilidad única.

Los problemas que resuelve este patrón son:

1. Garantizar que una clase tenga una única instancia.
2. Proporcionar un punto de acceso global a dicha instancia.

La idea de poder controlar cuántas instancias tiene una clase es controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo. Si decidimos crear un objeto al cabo de un tiempo decides crear otro nuevo. En lugar de recibir un objeto nuevo, obtendrás el que ya habías creado.

Aunque las variables globales son muy útiles, también son poco seguras, ya que cualquier código podría sobrescribir el contenido de esas variables y descomponer la

aplicación. El patrón Singleton nos permite acceder a un objeto desde cualquier parte del programa. No obstante, también evita que otro código sobreescriba esa

instancia.

Tampoco queremos que el código que resuelve el primer problema se encuentre disperso por todo el programa. Es mucho más conveniente tenerlo dentro de una clase, sobre todo si el resto del código ya depende de ella.

Hoy en día el patrón Singleton se ha popularizado tanto que la gente suele llamar singleton a cualquier patrón, incluso si sólo resuelve uno de los problemas antes mencionados.

Todas las implementaciones del patrón Singleton tienen estos

dos pasos en común:

• Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador new con la clase Singleton.

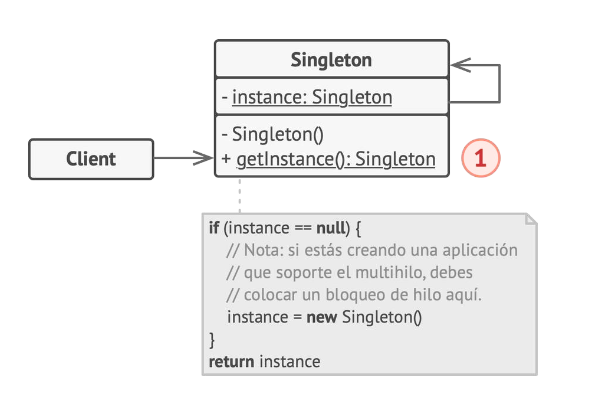
• Crear un método de creación estático que actúe como constructor. Tras bambalinas, este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Si tu código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo

El concepto de gobierno ilustra de manera efectiva el patrón Singleton. En cualquier país, existe solo un gobierno oficial reconocido. Más allá de las identidades individuales de quienes lo componen, el término "Gobierno de X" actúa como un punto de referencia universal que identifica al grupo de personas responsables de dirigir el país.

### 

### Estructura



La clase Singleton declara el método estático “obtenerInstancia” que devuelve la misma instancia de su propia clase. El constructor del Singleton debe ocultarse del código cliente. La llamada al método “obtenerInstancia” debe ser la única manera de obtener el objeto de Singleton.

El patrón Singleton es adecuado cuando una clase debe tener una única instancia compartida entre múltiples partes del programa, como un objeto de base de datos utilizado en todo el programa. El patrón garantiza que solo exista una instancia de la clase y proporciona un método especial para crear o devolver esa instancia. Además, el Singleton puede contener lógica específica de negocio que se ejecuta en su instancia. También es útil cuando se necesita un control más estricto sobre las variables globales, ya que garantiza una única instancia de una clase, a diferencia de las variables globales que pueden ser reemplazadas por cualquier parte del programa. Es posible ajustar esta limitación para permitir un número limitado de instancias Singleton, lo que solo requiere cambios en el cuerpo del método getInstance.

Lo podríamos implementar siguiendo los próximos pasos:

1. Agrega un campo estático privado a la clase para almacenar la instancia Singleton. Este campo contendrá la única instancia de la clase.
2. Declara un método de creación estático público para obtener la instancia Singleton. Este método será el punto de acceso global para obtener la instancia única de la clase.
3. Implementa una inicialización diferida dentro del método estático. Esto significa que dentro del método de creación, comprueba si la instancia ya ha sido creada. Si no, crea una nueva instancia y la asigna al campo estático. Luego, siempre devuelve esa instancia en todas las llamadas posteriores al método de creación.
4. Declara el constructor de clase como privado. Esto impide que se creen instancias de la clase desde fuera de la misma. Aunque el método estático de la clase aún puede invocar al constructor, evita que otros objetos lo hagan.
5. Repasa el código cliente y sustituye todas las llamadas directas al constructor de la instancia Singleton por llamadas a su método de creación estático.

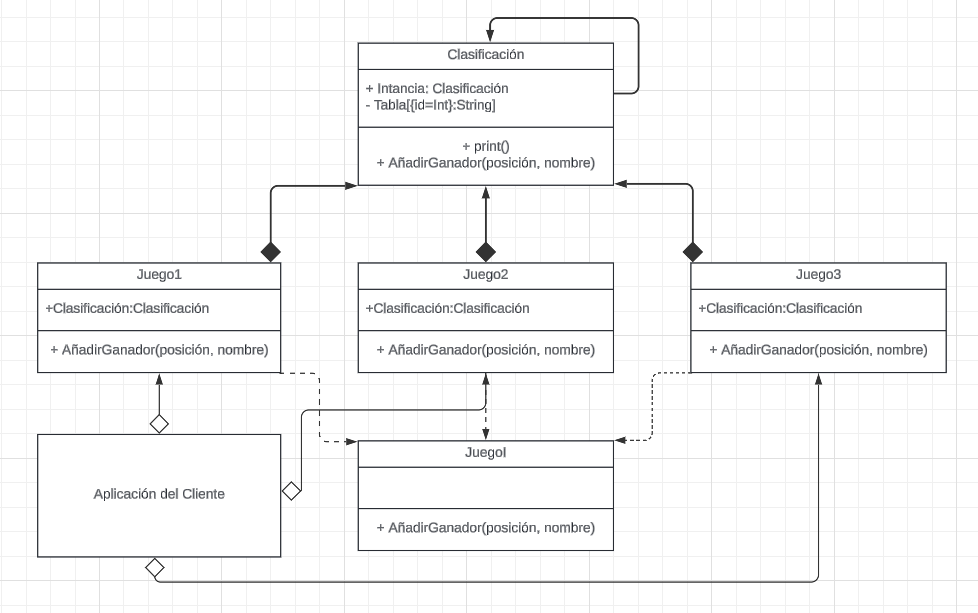
### Ventajas

1. El patrón Singleton garantiza que una clase tenga una sola instancia.
2. Proporciona un acceso global a esta instancia única.
3. La inicialización del objeto Singleton ocurre solo cuando se necesita por primera vez.

### Desventajas

El patrón Singleton puede encubrir problemas de diseño al permitir que los componentes del programa estén demasiado acoplados. Además, su implementación requiere consideraciones especiales en entornos con múltiples hilos para evitar la creación de instancias múltiples. La prueba unitaria del código cliente que utiliza Singleton puede ser complicada, ya que muchas herramientas de prueba dependen de la herencia, lo que dificulta la simulación de Singleton. Esto puede llevar a decisiones difíciles sobre si escribir pruebas unitarias o evitar el uso del patrón Singleton por completo.

### Ejemplo



En el ejemplo, se crean tres juegos. Todos son instancias independientes creadas a partir de su propia clase, pero comparten el mismo tablero de clasificación. El tablero de clasificación es un Singleton.

No importa cómo se hayan creado los juegos, ni cómo se hagan referencia al tablero de clasificación, siempre es un Singleton.

Cada juego añade de forma independiente un ganador, y todos los juegos pueden leer el tablero de clasificación modificado independientemente de qué juego lo haya actualizado.

1. Definición de las Clases de Juego (Juego1, Juego2, Juego3):
   * Cada clase de juego implementa la interfaz JuegoI, lo que significa que debe proporcionar la función AñadirGanador(), que añade un ganador al juego.
   * En el constructor de cada clase de juego, se instancia un objeto de tipo Clasificación, que es el tablero de clasificación.
   * Cada juego añade un ganador al tablero de clasificación llamando al método AñadirGanador() del tablero de clasificación.
2. Definición de la Clase Clasificación (Tablero de Clasificación):
   * La clase Clasificación implementa el patrón Singleton.
   * La instancia Singleton se asegura en el constructor de la clase. Si ya existe una instancia, se devuelve esa instancia en lugar de crear una nueva.
   * La clase mantiene un diccionario (#tabla) que almacena la posición y el nombre de los ganadores.
   * Tiene métodos para añadir un ganador (AñadirGanador) y para imprimir el tablero de clasificación (print).
3. Explicación de cómo se utiliza el Singleton:
   * A pesar de que cada juego crea una instancia de Clasificación, todos los juegos apuntan al mismo objeto en memoria, ya que Clasificación es un Singleton. Esto garantiza que todos los juegos compartan el mismo tablero de clasificación.
   * Al final, se imprime el tablero de clasificación de cada juego. Aunque se llama al método print desde instancias de diferentes juegos, siempre se mostrará el mismo tablero de clasificación actualizado.

# Actividad 2

## Especificación dirigida por riesgos

Este enfoque se centra en identificar y priorizar los requerimientos del sistema en función de los riesgos asociados. Implica un análisis detallado de los posibles riesgos que podrían afectar la confiabilidad y seguridad del sistema

* Proceso de Especificación por Riesgos:
  1. Identificación del Riesgo: Se identifican los riesgos potenciales según el entorno operacional del sistema.
  2. Análisis y Clasificación del Riesgo: Se analizan y clasifican los riesgos según su severidad y probabilidad.
  3. Descomposición del Riesgo: Se investigan las causas fundamentales de cada riesgo.
  4. Reducción del Riesgo: Se proponen medidas para mitigar o eliminar los riesgos, estableciendo requerimientos de confiabilidad y seguridad.
* Etapas del Análisis de Riesgo en Sistemas Grandes:
  1. Análisis Preliminar del Riesgo: Identifica los riesgos principales del entorno.
  2. Análisis de Riesgo de Ciclo de Vida: Se enfoca en riesgos relacionados con decisiones de diseño.
  3. Análisis de Riesgo Operativo: Considera riesgos relacionados con la interfaz de usuario y operación del sistema.
* Importancia de las fases: Las fases estructuradas del análisis de riesgo son esenciales debido a la complejidad de los sistemas y la influencia de las decisiones de diseño en los requerimientos de seguridad y confiabilidad.
* Adaptación de Requerimientos: Los requerimientos de seguridad y confiabilidad pueden modificarse según las tecnologías disponibles y las características del sistema comercial utilizado.

## Especificación de Protección

La especificación de protección en sistemas críticos se centra en identificar requisitos que reduzcan la probabilidad de fallos que puedan afectar el entorno del sistema y causar lesiones o muerte. Estos requisitos, también conocidos como requisitos de seguridad, no se preocupan por el funcionamiento normal del sistema, sino que se enfocan en mantener la protección. La clave al derivar estos requisitos es encontrar un equilibrio entre seguridad y funcionalidad para evitar excesiva protección que no sea rentable. En los sistemas críticos, un "peligro" es algo que podría resultar en daño o muerte, mientras que el "riesgo" es la probabilidad de que el sistema entre en un estado peligroso. Por lo tanto, la especificación de protección se enfoca en los riesgos asociados con situaciones específicas y los eventos que podrían llevar a tales peligros.

Estos son los pasos clave del proceso de especificación de protección en sistemas críticos:

1. **Identificación de Peligro:** Este paso implica identificar y definir claramente los posibles peligros o situaciones que podrían resultar en daño o riesgo para las personas o el entorno debido a fallas en el sistema.
2. **Valoración del Peligro:** Aquí se evalúa la gravedad y la probabilidad de cada peligro identificado. Se considera qué tan severo podría ser el daño o la lesión asociada con cada peligro y cuán probable es que ocurra.
3. **Análisis del Peligro:** En esta fase, se analizan las causas subyacentes de cada peligro para comprender por qué podrían ocurrir. Se examinan las condiciones y los eventos que podrían desencadenar un peligro específico.
4. **Reducción del Riesgo:** Finalmente, se desarrollan estrategias y medidas para reducir o eliminar los riesgos identificados. Esto puede incluir el diseño de sistemas de protección, la implementación de controles de seguridad adicionales o la modificación de procesos para minimizar la probabilidad o el impacto de los peligros.

## Especificación de Fiabilidad

La especificación de fiabilidad es un proceso crítico en el desarrollo de sistemas, especialmente en aquellos donde la fiabilidad es crucial para la seguridad, la protección o la disponibilidad continua de los servicios. La identificación, valoración y especificación de los requerimientos de fiabilidad ayudan a garantizar que los sistemas cumplan con los estándares necesarios y proporcionen el nivel de servicio requerido sin comprometer la seguridad ni la integridad de las operaciones. La selección adecuada de métricas de fiabilidad, como POFOD, ROCOF y AVAIL, junto con la especificación de requerimientos funcionales relacionados con la fiabilidad, contribuye a diseñar sistemas robustos y confiables que satisfacen las necesidades del usuario final.

1. Métricas de Fiabilidad:
   * Probabilidad de Falla a Pedido (POFOD):
     + Representa la probabilidad de que una solicitud de servicio del sistema resulte en una falla.
     + Por ejemplo, una POFOD de 0.001 indica que hay una probabilidad de 1/1000 de que ocurra una falla por cada solicitud de servicio.
   * Tasa de Ocurrencia de Fallas (ROCOF):
     + Indica la cantidad esperada de fallas por unidad de tiempo o número de ejecuciones del sistema.
     + Por ejemplo, una ROCOF de 10 fallas por día significa que se espera que ocurran 10 fallas diarias en promedio.
   * Disponibilidad (AVAIL):
     + Representa la probabilidad de que el sistema esté operativo cuando se solicita un servicio.
     + Por ejemplo, una disponibilidad de 0.999 significa que el sistema estará disponible el 99.9% del tiempo.
2. Tipos de Requerimientos de Fiabilidad:
   * Requerimientos No Funcionales:
     + Definen el número aceptable de fallas o el tiempo de inoperatividad del sistema durante el uso normal.
   * Requerimientos Funcionales:
     + Especifican las funciones del sistema y del software que ayudan a prevenir, detectar o tolerar fallas.

## Especificación de Seguridad

La especificación de seguridad implica definir los requisitos y controles necesarios para proteger el sistema contra amenazas internas y externas.

La especificación de los requisitos de seguridad para sistemas comparte similitudes con los requisitos de protección. No es práctico especificarlos de manera cuantitativa, y con frecuencia, los requisitos de seguridad se expresan en términos negativos, como "no debe", para definir el comportamiento inaceptable del sistema en lugar de la funcionalidad requerida. La seguridad es un problema que exige mayor esfuerzo que la protección por las siguientes razones:

1. Cuando se habla de protección, se asume un entorno no hostil donde no hay intenciones maliciosas por parte de terceros. En este caso, se implementan medidas para prevenir accidentes o errores no intencionales. Por ejemplo, se pueden establecer políticas de acceso y privilegios para asegurar que los usuarios accedan sólo a los recursos necesarios para su trabajo, evitando así alteraciones no autorizadas.
2. Por otro lado, cuando se trata de seguridad, se parte del supuesto de que pueden existir ataques deliberados al sistema por parte de personas malintencionadas. Se considera que los atacantes tienen conocimiento de las posibles vulnerabilidades del sistema y pueden intentar explotarlas para causar daño o acceder a información confidencial. Por lo tanto, se implementan medidas más rigurosas y sofisticadas para proteger el sistema contra amenazas externas, como firewalls, sistemas de detección de intrusiones y encriptación de datos.
3. Cuando nos enfrentamos a una posible falla de protección, como una vulnerabilidad conocida o una amenaza inminente, los administradores del sistema pueden optar por desactivar o degradar los servicios afectados temporalmente como medida de precaución. Esto se hace para minimizar el riesgo de una explotación exitosa de la vulnerabilidad o para prevenir un ataque en curso.Sin embargo, esta acción puede ser aprovechada por un atacante como parte de un ataque de denegación de servicio (DoS), cuyo objetivo es precisamente apagar o degradar el sistema objetivo. Si los servicios se desactivan o degradan debido a una acción del administrador del sistema en respuesta a una posible amenaza, el ataque de denegación de servicio se considerará exitoso, ya que ha logrado su objetivo de interrumpir el funcionamiento normal del sistema
4. Los problemas relacionados a la protección, no están generados por personas “inteligentes”, en cambio, los problemas relacionados a la seguridad están desarrollados por personas con la capacidad de analizar el sistema, obtener información y variar sus métodos de ataque con el fin de cumplir con cual sea que sean sus objetivos.

Los requerimientos de protección se centran en garantizar que el sistema funcione correctamente y en prevenir o mitigar eventos y fallas que podrían comprometer su operación. Esto se logra mediante la implementación de controles y mecanismos que verifiquen problemas potenciales y tomen medidas correctivas si surgen tales problemas.

Por otro lado, los requerimientos de seguridad son más amplios y abarcan una variedad de amenazas y riesgos que enfrenta el sistema. Estos requerimientos se centran en proteger el sistema contra posibles ataques maliciosos, incluidos los ataques de hackers, virus, malware y otras amenazas cibernéticas. Los requerimientos de seguridad pueden incluir medidas como el cifrado de datos, la autenticación de usuarios, la gestión de accesos y la detección de intrusiones.

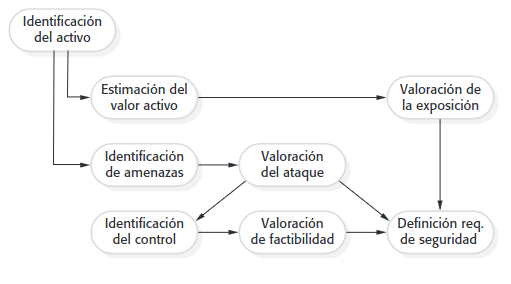
Estos son los 10 requerimientos de seguridad que pudo identificar Firesmith(2003):

1. Requerimientos de Identificación: Establecen si el sistema debe identificar a los usuarios antes de interactuar con ellos.
2. Requerimientos de Autenticación: Describen los métodos utilizados para identificar a los usuarios de manera confiable.
3. Requerimientos de Autorización: Detallan los privilegios y permisos de acceso que tienen los usuarios una vez que han sido identificados.
4. Requerimientos de Inmunidad: Especifican cómo el sistema debe protegerse contra virus, gusanos y otras amenazas similares.
5. Requerimientos de Integridad: Describen cómo se puede evitar la corrupción de datos, asegurando su integridad.
6. Requerimientos de Detección de Intrusiones: Puntualizan qué mecanismos deben utilizarse para detectar y responder a posibles ataques al sistema.
7. Requerimientos de No Repudio: Especifican que una parte en una transacción no puede negar su participación en la misma.
8. Requerimientos de Privacidad: Detallan cómo se debe mantener la privacidad de los datos del usuario y la información sensible.
9. Requerimientos de Auditoría de Seguridad: Plantean cómo se puede auditar y verificar el uso del sistema para garantizar su seguridad.
10. Requerimientos de Seguridad de Mantenimiento del Sistema: Especifican cómo el sistema puede protegerse contra cambios no autorizados, incluso en caso de desactivación accidental de los mecanismos de seguridad.

Posteriormente a esto tenemos que proceder a hacer el proceso de análisis y valoración de riesgos. Consta de 3 etapas:

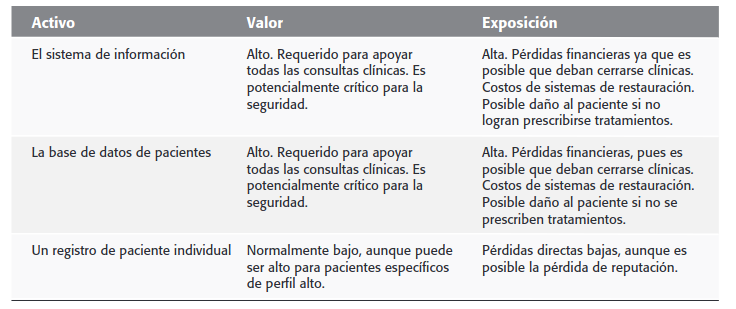
1. Análisis Preliminar del Riesgo:
   1. En esta etapa inicial, aún no se toman decisiones detalladas sobre los requerimientos del sistema, su diseño o la tecnología de implementación.
   2. El objetivo principal de este proceso es identificar y derivar los requerimientos de seguridad para el sistema en su conjunto.
   3. Se realiza una evaluación general de los riesgos potenciales que pueden afectar la seguridad del sistema, sin entrar en detalles específicos de diseño o implementación.
2. Análisis del Riesgo del Ciclo de Vida:
   1. Esta evaluación de riesgo ocurre durante el ciclo de vida del desarrollo del sistema, después de que se han tomado decisiones de diseño.
   2. Los requerimientos adicionales de seguridad tienen en cuenta las tecnologías utilizadas en la construcción del sistema, así como las decisiones de diseño e implementación.
   3. Se analizan los riesgos específicos asociados con las decisiones de diseño y tecnología seleccionadas, y se toman medidas adicionales para mitigar estos riesgos.
3. Análisis del Riesgo Operativo:
   1. Este análisis de riesgo considera los riesgos operativos para el sistema impuestos por posibles ataques maliciosos de los usuarios, ya sea con o sin conocimiento interno del sistema.
   2. Se evalúan las amenazas y vulnerabilidades que podrían afectar la operación del sistema en producción y se implementan medidas de seguridad adecuadas para mitigar estos riesgos.

Estos tres niveles de análisis del riesgo se complementan entre sí y proporcionan una visión holística de los posibles riesgos de seguridad que enfrenta el sistema en diferentes etapas de su ciclo de vida.



Se muestra un proceso de requerimiento de seguridad dirigido por riesgo. Las etapas del proceso son:

1. Identificación del Activo**:** Identificar qué aspectos del sistema necesitan protección, como el propio sistema, funciones específicas o datos asociados.
2. Estimación del Valor del Activo: Evaluar la importancia de los activos en términos de su contribución al negocio y el impacto en caso de pérdida o compromiso.
3. Valoración de la Exposición: Evaluar las posibles pérdidas asociadas con cada activo en caso de amenazas o incidentes de seguridad.
4. Identificación de Amenazas: Identificar las posibles amenazas que podrían afectar los activos del sistema, tanto internas como externas.
5. Valoración del Ataque: Descomponer cada amenaza en posibles ataques contra el sistema y considerar cómo podrían llevarse a cabo.
6. Identificación del Control: Proponer medidas de seguridad, como la encriptación, para proteger los activos del sistema.
7. Valoración de Factibilidad: Evaluar la viabilidad técnica y los costos asociados con la implementación de los controles propuestos.
8. Definición de Requerimientos de Seguridad: Derivar requisitos de seguridad del sistema basados en las evaluaciones previas, asegurando que sean proporcionales al valor y la importancia de los activos protegidos.



La política de seguridad de una organización es fundamental en el proceso de valoración y gestión del riesgo. Define lo que está permitido y lo que no en términos de seguridad, estableciendo condiciones que un sistema de seguridad debe mantener siempre para identificar posibles amenazas. Las políticas de seguridad pueden expresarse de manera informal, pero también es posible definirlas en un lenguaje formal y verificar su cumplimiento automáticamente.

## Especificación Formal

La especificación formal implica describir los requerimientos y propiedades del sistema utilizando métodos formales como lógica matemática o lenguajes formales. Esto ayuda a eliminar ambigüedades y facilita la verificación y validación del sistema.

Durante más de tres décadas, muchos investigadores han explorado el uso de métodos formales en el desarrollo de software. Estos métodos son enfoques con base matemática que implican la definición de un modelo formal del software. Esto permite un análisis formal del modelo y su uso como base para especificar formalmente el sistema. En esencia, es posible comenzar con un modelo formal para el software y demostrar formalmente que un programa desarrollado es coherente con dicho modelo. Esto ayuda a eliminar errores de software que pueden surgir debido a errores de programación.

El desarrollo formal de software comienza con la creación de un modelo formal del sistema, que actúa como su especificación. Este modelo se obtiene al traducir los requerimientos de usuario del sistema, expresados en lenguaje natural, diagramas y tablas, a un lenguaje matemático con semántica definida. La especificación formal describe de manera clara y sin ambigüedades lo que el sistema debe hacer, y permite verificar que el comportamiento de un programa sea congruente con ella. Estas especificaciones son esenciales para garantizar la precisión en el diseño e implementación del software, y para reducir errores derivados de malas interpretaciones. Además, facilitan un análisis detallado de los requisitos y promueven la detección temprana de posibles problemas. Por lo general, estas especificaciones se desarrollan como parte de un proceso de desarrollo de software planificado, donde se definen y verifican exhaustivamente los requisitos y el diseño antes de la implementación del sistema.

Hace unos años, se desarrollaron herramientas automatizadas de apoyo para analizar especificaciones formales, conocidas como comprobadores de modelos. Estas herramientas toman una especificación formal del sistema junto con algunas propiedades deseadas, como "no hay estados inalcanzables", y las analizan exhaustivamente. El programa de comprobación de modelos verifica si el modelo satisface las propiedades del sistema o si hay algún ejemplo que muestre lo contrario.

Algunos de los beneficios que podemos encontrar en desarrollar una especificación formal son:

1. Comprensión profunda de los requerimientos del sistema:

* El desarrollo de una especificación formal proporciona una comprensión detallada y exhaustiva de los requerimientos del sistema.
* Permite detectar errores de requerimientos de manera temprana, lo que evita problemas costosos en etapas posteriores del desarrollo.

Análisis automático de la especificación:

* Al expresar la especificación en un lenguaje con semántica formal, es posible realizar análisis automático.
* Esto facilita la detección de inconsistencias y elementos incompletos en la especificación.

Transformación precisa en programas:

* Mediante métodos como el método B, la especificación formal puede transformarse con precisión en un programa.
* Se utilizan secuencias de transformaciones que garantizan que el programa resultante cumpla con la especificación original.

Reducción de costos en las pruebas:

* Los costos asociados con las pruebas del programa se reducen significativamente.
* Al verificar el programa directamente contra su especificación formal, se eliminan las pruebas redundantes y se asegura la conformidad con los requerimientos establecidos.

Aunque tienen ventajas, los métodos formales no son ampliamente utilizados en el desarrollo práctico del software, incluso para sistemas críticos. Por lo tanto, hay poca experiencia en la comunidad sobre cómo desarrollar y usar especificaciones formales del sistema. Los argumentos en contra de utilizar especificaciones formales del sistema incluyen:

1. Los propietarios y expertos de dominio a menudo no comprenden las especificaciones formales, lo que dificulta la verificación precisa de los requerimientos del sistema. Además, los ingenieros de software pueden comprender la especificación formal pero no el dominio de aplicación, lo que también plantea dudas sobre su precisión.
2. Es fácil calcular los costos de crear una especificación formal, pero resulta complicado estimar los posibles ahorros que podría generar su uso. Por lo tanto, los administradores son renuentes a adoptar este enfoque debido al riesgo percibido.
3. La mayoría de los ingenieros de software carecen de entrenamiento en el uso de lenguajes de especificación formal, lo que los hace poco dispuestos a sugerir su implementación en los procesos de desarrollo.
4. Resulta complicado adaptar los enfoques actuales a la especificación formal para sistemas de gran escala. Generalmente, se utiliza la especificación formal para describir el núcleo crítico del software en lugar de sistemas completos.
5. La especificación formal no se alinea bien con los métodos de desarrollo ágiles.