

# Ingeniería de SOFTWARE

## bueno, bonito y barato

Consiste en establecer y usar principios de **ingeniería** orientados a obtener **software de manera económica, que sea fiable y funcione económicamente.**

## etapas = unidades

Disciplina que comprende todos los aspectos de la **producción de software**, desde las etapas iniciales de la especificación del sistema hasta el mantenimiento de éste después de que se utiliza.

**Universo de Información (UDI):**  
Contexto general en el cual el software deberá ser desarrollado y operado, el cual incluye las fuentes de información y las personas relacionadas (actores). Es influenciado por los requerimientos.

**Stakeholder:** todos aquellos que tienen algún interés en el cambio que se está considerando, aquellos que ganan y pierden con él (internos o externos). Ejemplo Facebook: desarrolladores (i), otras redes sociales, marcas con publicidad, inversores.

## SWEBOK

Software Engineering Body of Knowledge (SWEBOK) es una **guía del Cuerpo de Conocimientos de la Ingeniería de Software**, desarrollada por la IEE Computer Society y la Association for Computing Machinery.

Este proyecto busca aglutinar en un solo texto las competencias que debe tener todo ingeniero de software para desempeñarse competentemente en el mercado. De esta manera, clasifica y define todo lo que es **Ingeniería de Software (IS)**.

Está orientado hacia una variedad amplia de audiencias, como ser:

- Organizaciones públicas y privadas.
- Ingenieros de software practicantes.
- Elaboradores de políticas públicas.
- Sociedades profesionales.
- **Estudiantes de Ingeniería de Software** y Educadores y formadores.

Es así que su aplicación se da en innumerables escenarios, en muchos de manera esencial: Industria, Gobierno, empleo, contratación de personal, negociación, planificación de carreras, educación, conferencias, investigación, publicaciones, etc.

El proyecto SWEBOK tiene, en principio, cinco **objetivos**:

- Identificar el contenido de la disciplina de la Ingeniería de Software.
- Proveer acceso al cuerpo de conocimientos de la Ingeniería de Software.
- Promover una visión uniforme y consistente de la Ingeniería de Software a nivel mundial.
- Aclarar el lugar de la Ingeniería de Software con respecto a otras disciplinas tales como, ciencias de la computación, gestión de proyectos, matemáticas, etc.
- Proveer una fundamentación para el desarrollo del currículum (programas universitarios) y materiales de certificación individual.



La Guía de SWEBOK organiza el cuerpo de conocimientos en varias **Áreas de Conocimiento** (AC). En total se tienen 10 ACs. Asimismo, considera ocho disciplinas relacionadas:

AREAS DE CONOCIMIENTO	DISCIPLINAS RELACIONADAS
Requerimientos de Software	Ingeniería de la Computación
Diseño de Software	Ciencias de la Computación
Construcción de Software	Gestión
Prueba del Software	Matemáticas
Mantenimiento del Software	Gestión de Proyectos
Gestión de la Configuración Software	Gestión de la Calidad
Gestión de la Ingeniería de Software	Ergonomía del Software
Proceso de Ingeniería de Software	Ingeniería de Sistemas
Herramientas y Métodos en Ingeniería de Software	
Calidad del Software	

En este proyecto se usa una **organización jerárquica** para descomponer cada AC en un conjunto de temas con dos niveles. La Guía trata temas seleccionados de una manera compatible con grandes escuelas de pensamiento y con fallas generalmente encontradas en industrias, literatura y normas de la ingeniería de software. Después de todo, el cuerpo de conocimientos se encuentra en los temas a los que hace referencia, y no en la propia Guía.

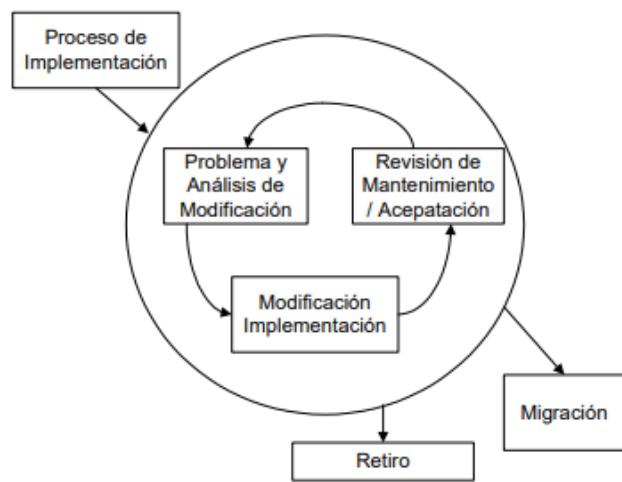
Las áreas de conocimiento que tratarán en Ingeniería de Software II son las siguientes:

1. **Requerimientos de Software:** Cuenta con cinco tareas (proceso,licitación, análisis, especificación y validación), las cuales no son secuenciales, sino que suceden común e iterativamente. Trata los procesos de análisis de requerimientos para detectar y resolver conflictos emergentes, descubrir falacias del sistema y cómo debería interactuar con su medio ambiente.
2. **Diseño de Software:** Describe cómo el sistema se descompone y se organiza en componentes, y describe las interfaces entre estos componentes, refinando la descripción de estos en un nivel de detalle conveniente por comenzar su construcción. Cuenta con las sub-áreas:
  - a. **Fundamentos de Diseño de Software:** Proceso del diseño, diseño arquitectónico, diseño detallado, técnicas clave, abstracción, acoplamiento/cohesión, descomposición y modularidad, encapsulamiento/ocultamiento de la información, separación de interfase e implementación.
  - b. **Tópicos clave de Diseño de Software:** Concurrencia, control y manejo de eventos, distribución de componentes, manejo de errores y excepciones y tolerancia a fallos, interacción y presentación, persistencia de datos.
  - c. **Estructura y Arquitectura de Software:** Estructuras arquitectónicas y puntos de vista, patrones de diseño, familias de programas y frameworks.
  - d. **Notaciones de Diseño de Software:**
    - i. Descripciones Estructurales (estáticas): Diagrama de clases y objetos, diagrama de componentes, tarjetas de clase-responsabilidad, colaborador, diagramas entidad-relación, diagramas de estructura de Jackson.
    - ii. Descripciones de Comportamiento (dinámicas): Diagramas de actividad y transición de estados, diagramas de colaboración y secuencia, diagramas de flujo y de flujo de datos, diagramas y tablas de decisión, lenguajes de especificación formales, pseudocódigo.
  - e. **Estrategias y Métodos del Diseño de Software:** Estrategias generales, diseño orientado a la funcionalidad (estructurado), diseño orientado a objetos, diseño centrado en estructuras de datos, diseño basado en componentes.

**3. Pruebas de Software:** Consisten en verificar dinámicamente la conducta del programa bajo un conjunto finito de casos de prueba y comparar los resultados con lo que se esperaba. Esta área del conocimiento se divide en dos, la primera de las cuales es organizada conforme a las fases tradicionales para testeo de grandes sistemas de software. La segunda trata las pruebas para condiciones o propiedades específicas.

**4. Mantenimiento del Software:** Es definido como una modificación al producto de software después de corregir fallas, mejorar la actuación, otros atributos o adaptar el producto a otro ambiente modificado. Sin embargo, los sistemas de software son raramente completados y constantemente evolucionan con el tiempo.

El proceso es crítico para tener éxito y un buen entendimiento del mantenimiento y evolución del software. Los procesos de mantenimiento nos indican las actividades necesarias y las entradas y salidas de estas actividades:

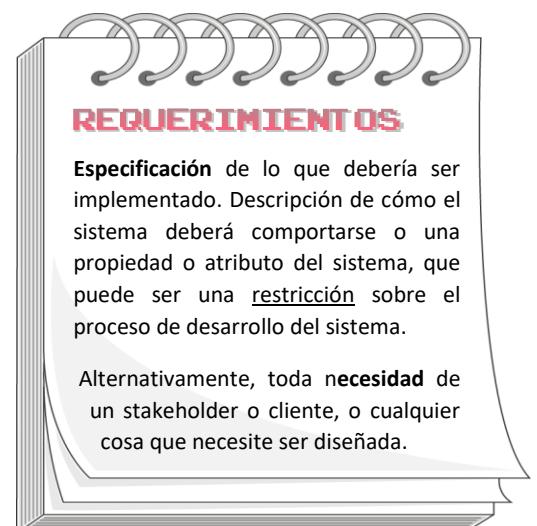


- Actividades de Mantenimiento:
  - Actividades únicas: Transición, modificación requerida, etc.
  - Actividades de soporte: Verificación y validación, etc.
  - Actividades de planificación de mantenimiento
  - Gestión de la configuración de Software
  - Calidad del Software
- Técnicas de Mantenimiento:  
Comprensión del programa, reingeniería, ingeniería reversa.

## ingeniería de requerimientos

La **Ingeniería de Requerimientos** es una subtarea de la Ingeniería de Software, es decir que está incluida dentro de ella. Esta propone métodos, técnicas y herramientas que faciliten el trabajo de definición, comunicación y administración de lo que se quiere de un software (**requerimientos**). Dichos recursos son esenciales ya que la fase de requerimientos, junto a la de diseño, es crítica en el proceso de desarrollo de software; errores en esta etapa (siendo la primera) se multiplican exponencialmente a medida que avanzamos y pueden inutilizar al producto.

Esta disciplina implica un proceso iterativo de analizar un problema, documentar las observaciones resultantes y verificar si lo que se comprendió es, en efecto, lo que el cliente busca a pesar de las dificultades de lenguaje técnico en la comunicación.

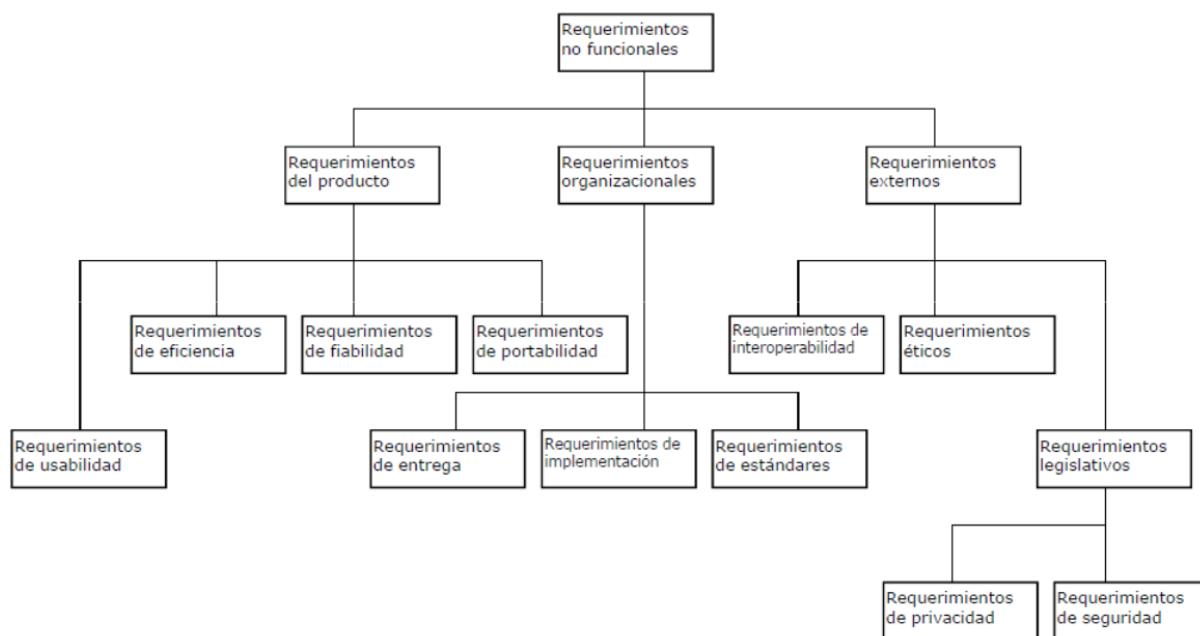


# Clasificación de los requerimientos



Las clasificaciones de los requerimientos difieren según el autor y pueden dividirse según distintos tipos de criterios. Según su **contenido**, los requerimientos pueden ser:

- **Funcionales:** Describen la **funcionalidad** o los **servicios** que se espera que el sistema proveerá. Dependen del tipo de software y del sistema que se desarrolle y de los posibles usuarios del software.  
Cuando se expresan como requerimientos del usuario, se describen de forma general, mientras que los requerimientos funcionales del sistema describen con detalle la función de éste, sus entradas y salidas, excepciones, etcétera. Esto quiere decir que los requerimientos funcionales pueden escribirse a *nivel c* y/o *nivel d* de abstracción.
- **No funcionales o atributos de calidad:** Se refieren a las **propiedades emergentes** del sistema, definiendo las restricciones del sistema (capacidad de los dispositivos de entrada/salida, representación de datos que se utiliza en interfaces, etcétera). Debido a que consideran al sistema como un todo y no se focalizan sólo en rasgos particulares del mismo, suelen ser más críticos que los requerimientos funcionales; mientras que el incumplimiento de un funcional degradará al sistema, una falla en un requerimiento no funcional del sistema lo inutiliza totalmente. Entre ellos se encuentran:
  - Calidad: confiabilidad, disponibilidad (por ejemplo: que pueda adaptarse a problemas de internet o cortes de luz, que no haya actualizaciones en horarios de trabajo), robustez.
  - Factores humanos: facilidad de uso, simplicidad de las interfaces.
  - Características de rendimiento: tiempos de respuesta, performance.
  - Restricciones de hardware y software: compatibilidad con equipamiento y/o sistemas disponibles.
  - Cambios y/o adaptaciones a nuevos requerimientos: adaptabilidad, reúso de componentes.
  - Restricciones de seguridad



La tabla anterior muestra la clasificación que hace Sommerville de los requerimientos no funcionales, evitando así hablar de los requerimientos inversos que otros autores sí proponen.

- **Requerimientos del producto:** Especifican el comportamiento del producto. Por ejemplo: requerimientos de desempeño en la rapidez de ejecución del sistema y cuánta memoria se requiere; de fiabilidad que fijan la tasa de fallas para que el sistema sea aceptable; de portabilidad y de usabilidad.
  - **Requerimientos organizacionales:** Se derivan de las políticas y procedimientos existentes en la organización del cliente y en la del desarrollador. Por ejemplo: estándares en los procesos que deben utilizarse; requerimientos de implementación como los lenguajes de programación o el método de diseño a utilizar; requerimientos de entrega que especifican cuándo se entregará el producto y su documentación.
  - **Requerimientos externos:** Este gran apartado cubre todos los requerimientos que se derivan de los factores externos al sistema y de su proceso de desarrollo. Estos incluyen: *requerimientos de interoperabilidad* (definen la manera en que el sistema interactúa con los otros sistemas de la organización); *requerimientos legales* (deben seguirse para asegurar que el sistema opere dentro de la ley); *requerimientos éticos* (impuestos al sistema para asegurar que será aceptado por el usuario y por el público en general).
- **Inversos:** Definen cómo el sistema de software **nunca se debe comportar**. Por ejemplo: un sistema que *no* admita pagos a través de MercadoPago.

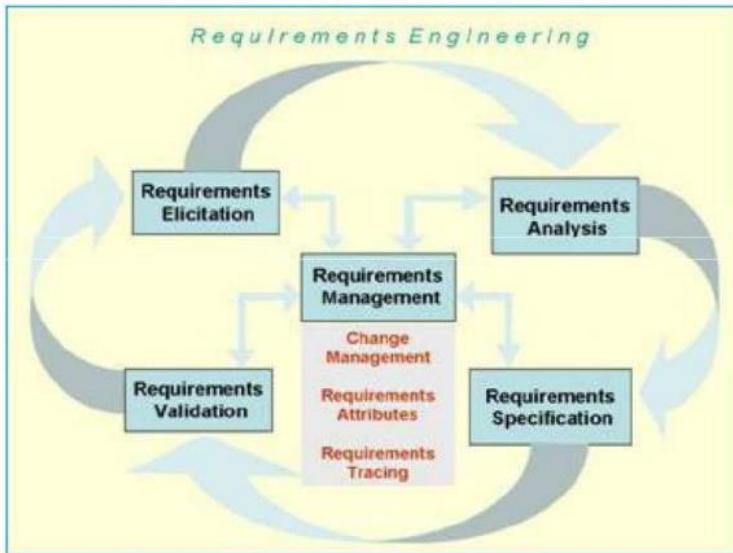
Por otro lado, y actuando de manera complementaria y no exclusiva con la clasificación de contenido, en cuanto al **nivel de abstracción** se tiene:

- **Requisitos-c:** Requisitos del cliente, los cuales van a ser documentados en un lenguaje general y entendible para cualquier grado de experiencia en el tema a desarrollar (ya sea en cuanto al desarrollo del software como a información concreta del tipo de sistema). Los clientes van a poder confirmar con seguridad si lo detallado se alinea con sus necesidades, pero es posible que los desarrolladores/diseñadores diverjan hacia otras direcciones por falta de especificidad.
- **Requisitos-d:** Requisitos del desarrollador, los cuales van a ser documentados en un lenguaje técnico y de mayor formalidad para evitar problemas en el diseño por ambigüedades. Debe ser específico, por lo cual si es el cliente quien lo lee, es posible que no comprenda del todo y acepte las condiciones a pesar de no ser exactamente lo que busca.

## Proceso de la Ingeniería de Requerimientos

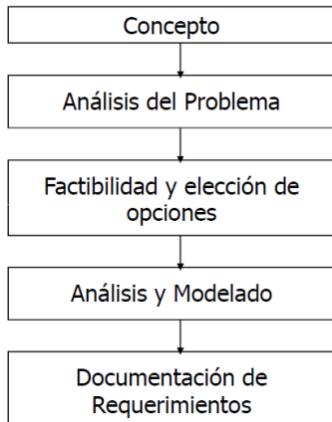


La Ingeniería de Requerimientos, si bien en sí misma es una etapa de la Ingeniería de Software, también se divide en **etapas propias**. No existe una división oficial, sino que los distintos autores varían en sus propuestas, pero se recomienda la siguiente:



En donde las etapas pueden solaparse en vez de ser una secuencia estrictamente lineal.

Elicitation = Elicitación
Management = Administración
Validation = Validación
Specification = Especificación
Analysis = Análisis
TRADUCCIÓN



El gráfico a la izquierda muestra otro tipo de proceso para la etapa de requerimientos, también válido, pero más lineal que el anterior. En este, cada etapa debe ser seguida de una validación, así poder chequear la veracidad de la información conseguida y el entendimiento sobre el problema.

A partir de esta metodología (a diferencia de si se tuviera la validación como una etapa secuencial más) se reduce la probabilidad de errores pues se evita arrastrarlos desde un principio hasta la última etapa.

Existen criterios de validación para los requerimientos, los cuales deben cumplirse obligatoriamente para asegurar la eficiencia del sistema. Estos son:

- **Completo:** El requerimiento debe describir toda la información necesaria para implementarlo en forma explícita.
- **Necesario:** El requerimiento debe poder asociarse con un uso concreto y relevante.
- **Verificable:** El requerimiento debe poder ser verificado una vez implementado.
- **Consistente:** El requerimiento debe actuar en forma armónica con el resto de los requerimientos funcionales y no-funcionales.
- **Alcanzable:** El requerimiento debe permitir su implementación. Es decir que debe ser factible para el contexto.



A la hora de eliciar requerimientos, es posible utilizar diferentes técnicas para conseguir la información necesaria. Dependiendo del contexto (quién es el cliente, quiénes son los usuarios finales, qué tipo de sistema se busca realizar, con qué fin) algunas técnicas van a ser más eficientes que otras. Es importante saber determinar cuál usar para así evitar errores en las conclusiones y obtener resultados óptimos. Es posible elegir entre las siguientes técnicas y más:

- **Lectura de documentos:** Contacto con el vocabulario de la aplicación y el Universo de Información.
- **Observación:** El analista tiene una posición pasiva en el Universo de Información observando el ambiente donde el software irá a actuar. A veces se utilizan cámaras, en general es observar, tomar notas y a veces preguntar. Como desventaja se tiene la posible intimidación a los clientes, saboteando la exactitud de los datos.
- **Entrevistas:** Son el medio más usual con el cual el analista recoge los hechos. Existen diferentes tipos de entrevista: estructuradas, informales y tutorías. Puede llegar a anticiparse las preguntas al usuario, así ya llega preparado y se optimiza el tiempo.
- **Cuestionarios:** Son utilizados cuando se tiene un buen conocimiento sobre el problema (aplicación) y se quiere abarcar un número grande de clientes o hay dispersión geográfica importante.
- **Análisis de protocolos:** Esta estrategia consiste en analizar el trabajo de determinada persona a través de sus relatos, normalmente durante su trabajo. Es como la observación, pero más activa y menos intuitiva.
- **Participación activa de los actores del Universo de Información:** Intenta incorporar al grupo de analistas los actores que demandan el software (primero le decimos como estamos acostumbrados a escribir casos de uso y los traemos a los usuarios a trabajar con los analistas). Los actores deben aprender el lenguaje de modelado.
- **Enfoque antropológico:** Se usa una técnica inversa de la descripta anteriormente, ya que los analistas deben procurar integrarse al Universo de Información de forma de alcanzar un conocimiento lo más amplio posible del problema. Es decir, está en la empresa haciendo el trabajo en vez de, tal vez, observarlo.
- **Reuniones:** Son una extensión de las entrevistas.
- **Reutilización:** Consiste en reutilizar hechos ya elicitiados. Por ejemplo: si un proyecto se suspendió y queda en stand-by, se retoma hasta donde se llegó. O si otro equipo de analistas hizo documentos, se pueden pedir.
- **Recuperación del diseño de software:** Estudio de software disponible en la organización. Trabajos de reingeniería han sido propuestos como una forma de, no solo recuperar lo que exactamente hacen los sistemas existentes, sino también como para optimizar la tarea de mantenerlos.

## Trazabilidad de los Requerimientos



La **rastreabilidad** de requerimientos (traceability) significa que los requerimientos deben estar relacionados de alguna manera y que a su vez deben estar relacionados a sus fuentes. Si elimino o modifco uno, debo ver a qué otro puede llegar a afectar por relacionarse.

Es una propiedad de una buena especificación de los requerimientos que se ve reflejada por la facilidad para encontrar requerimientos relacionados.

Algunas herramientas CASE proveen soporte para la rastreabilidad. Por ejemplo, pueden encontrar todos los requerimientos que usen los mismos términos.

Entre las **técnicas de trazabilidad** encontramos:

- Asignar un número único a todos los requerimientos.
- Hacer una referencia cruzada (una matriz también llamada, en inglés cross-reference) de los requerimientos relacionados utilizando este número único.
- Producir una matriz de referencias cruzadas para cada documento de requerimientos mostrando los requerimientos relacionados. Varias matrices pueden ser necesarias para diferentes tipos de relaciones.



# Documento de Requerimientos

El **Documento de Requerimientos** (DR), también conocido como Especificación de Requerimientos de Software (ERS o SRS, en inglés), es una especificación de lo que se requiere que haga un sistema informático y no de cómo hacerlo (etapa de diseño).

Un DR puede ser evaluado por su efectividad como un medio de comunicación, por su contenido como una medida de chequeo (checklist), y por la calidad de su contenido.

No existe un nivel de especificación universalmente correcto; los clientes con gran experiencia prefieren especificaciones de alto nivel, y aquellos con escasa experiencia pretenden mayor detalle. Como recomendaciones generales se tiene:

- Cada cláusula debe contener solamente un requerimiento.
- Evitar tener requerimientos que hace(n) referencia a otro(s).
- Agrupar los requerimientos semejantes. La ERS IEEE propone la siguiente guía general:

1. Introducción	3.1.2. Requerimiento Funcional 2 Requerimientos de Interfase externos
1.1. Propósito	3.2.1. Interfase de usuarios
1.2. Alcance	3.2.2. Interfase de Hardware
1.3. Definiciones, Acrónimos, y Abreviaciones	3.2.3. Interfase de Software
1.4. Referencias	3.2.4. Interfase de Comunicación
1.5. Resumen	3.3. Requerimientos de Performace
2. Descripción General	3.4. Restricciones de Diseño
2.1. Perspectiva de Producto	3.4.1. Cumplimiento de Estándares
2.2. Funciones	3.4.2. Limitaciones de Hardware
2.3. Características de Usuario	3.5. Atributos
2.4. Restricciones generales	3.5.1. Seguridad
2.5. Suposiciones y dependencias	3.5.2. Mantenibilidad
3. Requerimientos Específicos	3.6. Otros Requerimientos
3.1. Requerimientos Funcionales	3.6.1. Bases de Datos
3.1.1. Requerimiento Funcional 1	3.6.2. Operaciones
3.1.1.1. Introducción	3.6.3. Adaptación al sitio
3.1.1.2. Entradas (Inputs)	
3.1.1.4. Salidas (Outputs)	
	Apéndices Índice

Por lo general, los clientes solicitan un nuevo sistema ya sea porque no lo tienen o porque su viejo sistema no resulta satisfactorio. En ambos casos, la documentación de requerimientos presenta el problema que el nuevo sistema debe resolver.

- De acuerdo a la IEEE (1984), un buen DR debería contener sentencias no ambiguas, completas, verificables, consistentes, modificables, trazables (seguibles) y usables. Ejemplo: “que la contraseña sea segura” tiene mucho lugar a ambigüedades y está incompleto. Debería ser tipo “que la contraseña precise de al menos una mayúscula, un número y un carácter especial.”

Existen tres tipos de lenguajes para confeccionar un documento de requisitos:

- **Informales:** (*gráficos, arbitrarios, lenguaje natural, etcétera*) Es natural para los stakeholders, fácil de validar y no tiene automatización directa.
- **Semi-formales:** (*diagramas-ER, SADT, etc.*)
- **Formales:** (*Z, VDM, etc.*) Son aquellos lenguajes que pueden ser interpretados por una máquina sin ambigüedades, pero sólo lo conocen los técnicos. Generalmente se usa en doble-escritura (I+F).

Un usuario necesita de un lenguaje informal para exponer sus necesidades y entender lo que el analista captó del problema. El analista, a su vez, precisa de lenguajes semi-formales y formales para entender mejor el problema, verificar inconsistencias y ambigüedades.

# Casos de Uso

Los requerimientos son importantes y es donde las técnicas del **UML** son especialmente provechosas. El punto de partida son los casos de uso (técnica importante para documentar requerimientos). Estos, por lo tanto, son los motores de todo el proceso de desarrollo, ya que en todas las etapas se parte de un caso de uso.

El **diagrama de casos de uso** representa la forma en que un Cliente (Actor) opera con el sistema en desarrollo, además de la forma, tipo y orden en que interactúan los elementos (operaciones o casos de uso). Representan requisitos funcionales del sistema y se describen como conjuntos de secuencias que reflejan la interacción entre los elementos externos al sistema y el propio sistema.

Cada caso de uso es una funcionalidad del programa diferente, que responde a un **requerimiento funcional** (el caso de uso no es un requerimiento, el requerimiento está expresado de manera general). Suponete que tenes que hacer el software para un negocio que vende ropa, un caso de uso sería eliminar un producto, otro caso de uso sería listar productos en stock (muestra por pantalla e imprime), otro caso de uso es registrar un cliente, otro caso de uso será registrar venta o confeccionar factura.

## OBJETIVOS

- Definir el límite entre el sistema a desarrollar y los elementos externos a ese sistema (actores usuarios del sistema).
- Capturar el conjunto de funcionalidades y comportamientos del sistema a desarrollar.

## ELEMENTOS Y CONECTORES

Elementos de los Diagramas de Casos de Uso	Conectores de los Diagramas de Casos de Uso
Actor	Uso
Casos de Uso	Asociar
Colaboración	Generalización
Límite	Inclusión
Paquete	Extensión
	Realización
	Dependencia
	Traza

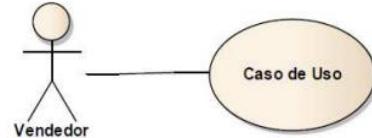


Un **actor** es un "rol" que un usuario juega con respecto al sistema. También se lo define como "entidad externa al sistema que se modela y que puede interactuar con él". Siguiendo esta línea, ya que los actores son externos al sistema que vamos a desarrollar, al identificarlos estamos comenzando a delimitar el sistema y definir su alcance.

Un actor no necesariamente representa a una persona en particular, sino más bien la labor que realiza frente al sistema. Puede ser una persona o un grupo de personas homogéneas, otro sistema, o una máquina. Es así que se diferencia "usuario" y "actor":

- Un **actor** es el rol o papel que juega un objeto externo en su relación con el sistema.
- Un **usuario** es una persona que, cuando usa el sistema, asume un rol. Así, un usuario puede acceder al sistema como distintos actores.

**Actores primarios** invocan de forma directa o indirecta un caso de uso. Un actor es **secundario** cuando, por ejemplo, tengo un caso de uso para registrar un movimiento en un cajero. Ese cajero va a tener que conectarse con otro sistema. Al ser otro sistema, es un actor. Va a tener que preguntar el saldo de la cuenta y después actualizar cuando se saca plata de la cuenta. Ese sistema es un actor secundario, no se ejecutó, pero está y hay una relación con el caso de uso.



Un **caso de uso** es una operación/tarea específica que se realiza tras una orden de algún agente externo, sea desde una petición de un actor o bien desde la invocación desde otro caso de uso. Es una secuencia de n pasos, con n alternativas. No deben anidar casos de uso como secuencias temporales. Las relaciones entre los casos de uso no marcan temporalidad, no es una secuencia.

Entre las pautas de su implementación encontramos:

- Los nombres de los casos de uso se expresan como una frase verbal: **verbo en infinitivo** + el objeto que está siendo afectado, por ej.: Diseñar reporte de clientes morosos.
- Está expresado desde el punto de vista del actor.
- Se documenta con texto informal.
- Describe tanto lo que hace el actor como lo que hace el sistema cuando interactúa con él.
- Es iniciado por un único actor.
- Está acotado a una determinada **funcionalidad** del sistema.
- Es independiente del método de diseño que se utilice y, por lo tanto, del método de programación.

## CLASIFICACIONES DE LOS CASOS DE USO

- **DE TRAZO GRUESO:** Se realiza una descripción "gruesa" de todos los casos de uso. Primero se identifican todos los casos de uso del sistema, sólo al nivel de su nombre. No se deben contemplar los detalles de la interacción entre el actor y el sistema. Se deben incluir las alternativas más relevantes, ignorando la mayoría de los errores que pueden aparecer en el uso del sistema. No se debe entrar en detalle sobre las acciones que realiza el sistema.
- **DE TRAZO FINO:** Se especifican una vez que se ha tomado la decisión de implementarlos. Se detalla todo aquello que no se detalló en los casos de uso de trazo grueso, por lo tanto, se pueden incluir detalles sobre la forma de interfaz en la descripción del caso y especificaciones con más detalle del comportamiento interno del sistema.
- **TEMPORALES:** Cuando el inicio de una determinada funcionalidad del sistema es provocado exclusivamente por el paso del tiempo (ejemplo: cada treinta minutos, el día 7 de cada mes), entonces es el paso del tiempo el que inicia el caso de uso. Es importante que cuando se especifican los casos de uso de trazo fino, se exprese claramente cuál es el momento del tiempo en el que se inicia el caso.  
Para expresar claramente que es el paso del tiempo el que inicia el caso, podemos incluir un símbolo representando un reloj en el gráfico de Casos de Uso, o usar una línea punteada en el borde del óvalo del caso.
- **PRIMARIOS:** Aquellos que son ejecutados en forma directa por un actor. En un diagrama "general" de casos de uso, son los únicos que se modelan, con el objetivo de poder mostrar "todas" las funciones del sistema.
- **SECUNDARIOS:** Son invocados o ejecutados por otro caso de uso. Surgen luego de una "explotación" de los casos de uso primarios, cuando se comienza a mostrar mayor nivel de detalle mediante relaciones de "inclusión" o "extensión".



Una **relación** está representada por una línea entre los casos de uso y/o actores relacionados, siendo que el extremo de dicha línea dependerá del tipo de relación. Entre los distintos tipos de relaciones, utilizaremos:

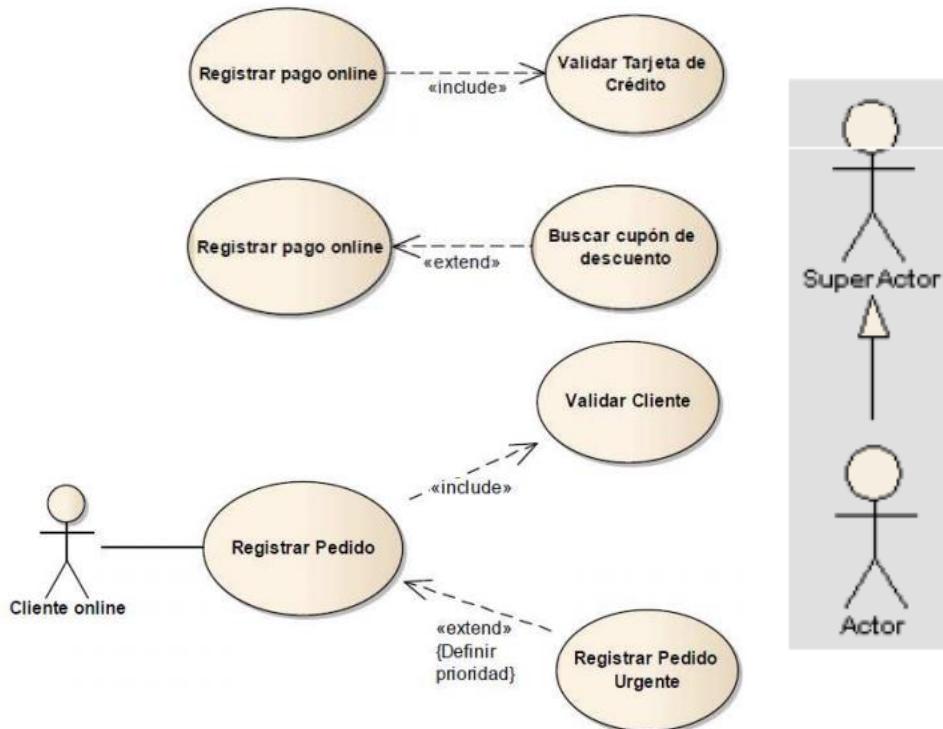
- **Comunicación/asociación:** Relación entre un actor y un caso de uso, indicando la **conexión** entre ambos. El estereotipo de la relación de comunicación es <<communicate>> aunque generalmente no se estipula ningún nombre. Se vinculan mediante una línea sin flecha para ningún lado (antes se ponía flecha, ahora no por distintas cuestiones).

- **Inclusión <<include>>**: Los casos de uso pueden contener la funcionalidad de otro caso de uso como parte de su proceso normal. En general se asume que cualquier caso de uso incluido se llamará cada vez que se ejecute una ruta básica (es necesario, por ejemplo, *Buscar Crédito* antes para poder *Pagar Crédito*).
 

Los casos de uso pueden ser incluidos por uno o más casos de uso, ayudando a reducir el nivel de duplicación de la funcionalidad realizando un factoreo del comportamiento común en casos de uso que se usan muchas veces.

Se grafican con una línea punteada y una flecha sin rellenar que apunta al **caso de uso común** (esto es, el caso de uso que contiene un comportamiento común para más de un caso de uso).
- **Extensión <<extend>>**: Refleja situaciones particulares en un caso de uso que pueden ser tratadas (extendidas) por otro. En la descripción del caso de uso que es extendido debe haber una forma de indicar en qué punto entra en juego el caso de uso que lo extiende (punto de extensión).
 

Se grafican con una línea punteada y una flecha sin rellenar que apunta del **caso de uso extendido** al básico.
- **Generalización o herencia**: Se usa para indicar herencia entre dos actores o dos casos de uso, no mezclados, cuando hay funcionalidades comunes entre estos. Con una flecha rellena dibujada desde el clasificador específico al clasificado general, la implicación de generalización es que el **origen hereda las características del destino**.



## DESARROLLO DEL MODELO DE CASOS DE USO

1. Identificar quiénes utilizarán el sistema en forma directa. Estos son los **actores**.
2. Tomar uno de esos actores para comenzar a trabajar.
3. Definir **qué quiere hacer** ese Actor con el sistema.
4. Para cada uno de esos Casos de Uso decidir de la manera más usual (en lenguaje natural), cuándo y para qué este Actor utiliza el sistema.
5. **Detallar** ese bloque descriptivo de la actividad en la descripción del Caso de Uso.
6. Una vez que se esté conforme con la descripción, se deben considerar extensiones, y agregarlas como **extensiones** de los Casos de Uso.

## DOCUMENTACIÓN DE LOS CASOS DE USO

Los casos de uso se documentan con **texto informal**, a pesar de ser una **técnica semiformal** por seguir ciertas reglas. Si la descripción del mismo es muy compleja, es conveniente complementarla con notaciones gráficas, por ej. los diagramas de actividad. Algunas de las formas son:

- A través de una lista enumerada.
- A través de una tabla.
- A través de una tabla cuando hay alternativas.
- A través de un gráfico.

Paquete:	Iteración
Nombre del Caso de Uso:	Versión
Actor Principal:	Actor Secundario:
Prioridad:	<input type="checkbox"/> Esencial <input type="checkbox"/> Útil <input type="checkbox"/> Deseable
Complejidad:	<input type="checkbox"/> Extremo    Complejo <input type="checkbox"/> Muy Complejo <input type="checkbox"/> Complejo <input type="checkbox"/> Mediano <input type="checkbox"/> Simple
Tipo de Caso de Uso:	<input type="checkbox"/> Concreto <input type="checkbox"/> Abstracto
Objetivo:	
Precondiciones:	No aplica
Post- Condiciones:	Éxito: <input checked="" type="checkbox"/>
	Fracaso: <input type="checkbox"/> <input type="checkbox"/>
Cursos Normales	Cursos Alternativos
1.	
2.	
...	
n.	
Requerimientos No funcionales Especiales:	No aplica
Asociaciones de Extensión:	No aplica
Asociaciones de Inclusión:	No aplica
Caso de Uso de Generalización:	No aplica
Observaciones:	No aplica

No tiene actor principal si es caso de uso temporal

Caso de Uso: Nombre del Caso de Uso	
Actor: Nombre del Actor	
Curso Normal:	Alternativas:
1) Paso 1	
2) Paso 2	2.1 Alternativa 1 del Paso 2 2.2 Alternativa 2 del Paso 2
.....	
n) Paso n	

Usamos esta

### RECOMENDACIONES:

1. ¿Quién inicia y finaliza el sistema?
2. ¿Quién es el usuario y quién administra la seguridad?
3. ¿Quién hace la administración del sistema?
4. ¿Interviene el tiempo como un actor, debido a que el sistema hace algo en respuesta a un evento de tiempo?
5. ¿Quién evalúa la actividad o el desempeño del sistema?
6. ¿Quién evalúa los ingresos (login)?
7. Si falla el sistema, ¿hay un proceso de monitoreo que lo reinicia?
8. ¿Cómo se manejan las actualizaciones del sistema?

**COMPLEJIDAD DE UN CASO DE USO:** La complejidad de un caso de uso está determinada por el tiempo que me va a llevar realizarla. La suma de complejidades y tiempos asociados va a determinar cuánto necesitas para realizar el software.

**TIPOS DE CASO DE USO:** Si un caso de uso hereda de otro, el caso de uso padre es abstracto y el caso de uso hijo es concreto. Si no hay herencia, son todos concretos. Los que voy a programar son los concretos.

**PRECONDICIONES:** "Loggearse y tener permisos" -> CdU "Añadir publicación a novedades"

**POSCONDICIONES:** Éxito y fracaso. Si el objetivo es que se dé de alta un nuevo cliente, la condición de éxito es "se registró con éxito el nuevo cliente.", mientras que las condiciones de fracaso son "no se registró por x razón" "x razón" pueden ser varias dependiendo qué sucedió exactamente.

**NIVEL DE ABSTRACCIÓN:** Cuando armo un modelo de casos de uso, elijo el nivel de abstracción que voy a representar con las funcionalidades que estoy modelando. Tengo que usar ese mismo nivel de abstracción en todo el modelo. Si digo que un caso de uso es equivalente a una pantalla, tengo que respetar ese nivel de abstracción que estoy usando.

**CURSO NORMAL Y CURSOS ALTERNATIVOS:** El paso n es la finalización normal, satisfactoria para llegar al objetivo del caso de uso. Complicaciones de las tareas van en cursos alternativos (ej: datos incompletos, operación cancelada), los cuales ralentizan o terminan el proceso. Por ejemplo:

- Paso 5: Usuario presiona aceptar y el sistema valida la operación.
- Alternativa 5.1: Si el usuario presiona cancelar, el caso de uso termina.
- Alternativa 5.2: Si hay datos incompletos, se informa un mensaje de error y volvemos al paso 3.

Como ejemplificación en un formato distinto al anteriormente mostrado, se tiene:

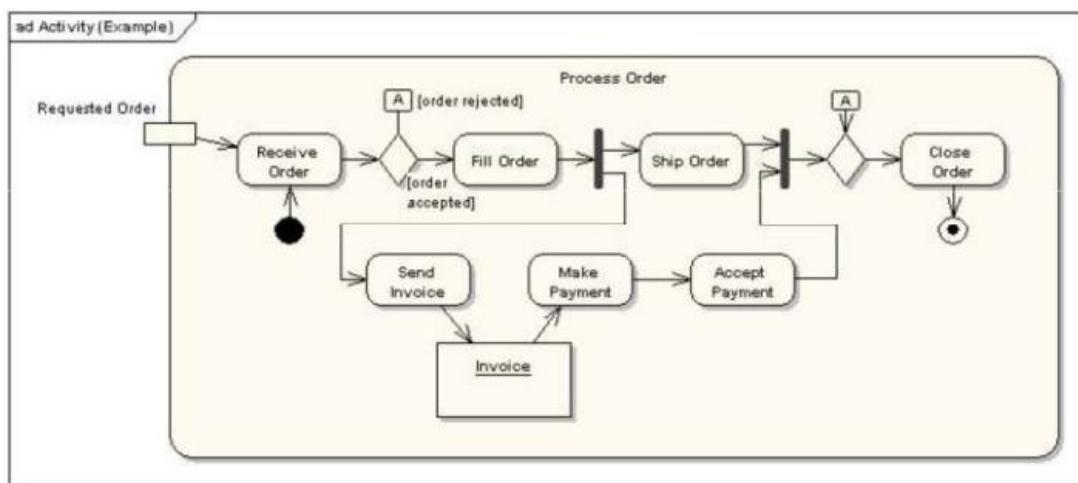
Paso	Acción
1	El caso de uso se inicia cuando el usuario presiona el botón <b>Eliminar</b> en un registro del resultado de la de Búsqueda (pantalla Gestionar Tema) o en el botón <b>Eliminar</b> de la pantalla de Consulta de Tema.
2	El sistema alerta al usuario si existen trámites creados bajo este tema. 2.a.1.1: El usuario confirma la eliminación del tema.
	2.a.1.2: El sistema informa que se eliminó correctamente el tema. Si el tema estaba relacionado a algún trámite, el borrado es lógico (marca de borrado). De otra forma, el borrado es físico.
	2.b : El usuario presiona Cancelar.

# Diagramas de Actividad

Los **Diagramas de Actividades**, gráficamente, serán un conjunto de arcos y nodos. Sirven fundamentalmente para modelar el flujo de control entre actividades, pero entre sus funciones encontramos:

- Mostrar el flujo de actividades que tienen lugar a lo largo del tiempo, así como las tareas concurrentes que pueden realizarse a la vez.
- Muestran el flujo de trabajo desde el punto de inicio hasta el punto final, detallando muchas de las rutas de decisiones que existen en el progreso de eventos contenidos en la actividad.
- Detallan situaciones donde el proceso paralelo puede ocurrir en la ejecución de algunas actividades.

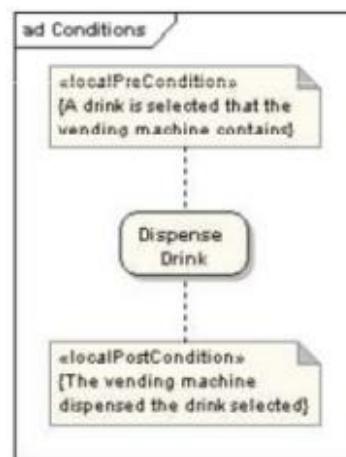
Los Diagramas de Actividades son útiles para el Modelado de Negocios donde se usan para **detallar el proceso involucrado en las actividades de negocio**.



**ELEMENTOS DEL DIAGRAMA DE ACTIVIDAD:** Un diagrama de actividades tiene **estados de actividad**, **estados de acción**, **transiciones** y **objetos**. Como convención se grafica de izquierda a derecha.

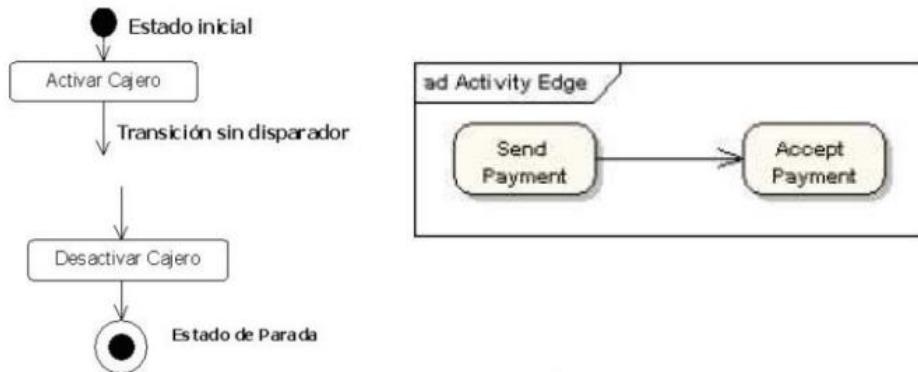
**DIFERENCIAS ENTRE ACTIVIDAD Y ACCIÓN:** La representación de ambas es un rectángulo con las puntas redondeadas, en cuyo interior se representa una actividad o una acción respectivamente. La forma de expresar tanto una actividad como una acción no queda impuesta por UML, se podría utilizar lenguaje natural, una especificación formal de expresiones, un metalenguaje, etc.

Una **acción es atómica**, no puede subdividirse en actividades menores ni pausarse o interrumpirse su ejecución. Si yo veo gráficamente, no los puedo diferenciar porque se representan igual. El rectángulo que engloba todo es sí o sí una actividad porque tiene varias subactividades, pero los que están dentro no los puedo diferenciar solo por la forma que tienen.



Las restricciones se pueden adjuntar a una acción, como se ve en la imagen:

**TRANSICIONES O FLUJOS DE CONTROL:** Las **transiciones** reflejan el paso de un estado a otro, bien sea de una actividad o de acción. Esta transición se produce como resultado de la finalización del estado del que parte el arco dirigido que marca la transición.

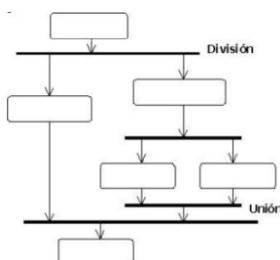


**BIFURCACIONES:** Un flujo de control no tiene porqué ser siempre secuencial ni tener una linealidad, puede presentar caminos alternativos. Para poder representar dichos caminos alternativos o **bifurcación** se utilizará como símbolo el rombo, a los cuales llega una y sólo una transición de entrada y tiene de 2 a n salidas.

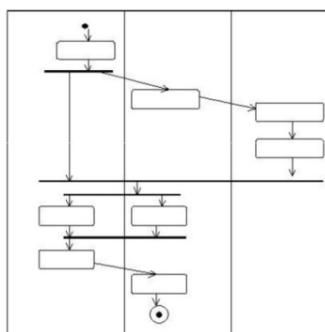


En cada transición de salida se colocará una **expresión booleana** que será evaluada una vez al llegar a la bifurcación, las guardas de la bifurcación han de ser excluyentes y contemplar todos los casos ya que de otro modo la ejecución del flujo de control quedaría interrumpida.

Para poder cubrir todas las posibilidades se puede utilizar la palabra **ELSE**, para indicar una transición obligada a un determinado estado cuando el resto de guardas han fallado.



**DIVISIÓN Y UNIÓN – FORK Y JOIN:** Hay casos en los que se requieren tareas concurrentes. UML representa gráficamente el proceso de división, que representa la concurrencia, y el momento de la unión de nuevo al flujo de control secuencial, por una línea horizontal ancha.



**CALLES – PARTICIONES – SWIMLINES:** Cuando se modelan flujos de trabajo de organizaciones, es especialmente útil dividir los estados de actividades en grupos, cada grupo tiene un nombre concreto y se denominan **calle**s.

Cada calle representa a la parte de la organización responsable de las actividades que aparecen en esa calle. Pueden ser verticales u horizontales.

## SEMÁNTICA DEL DA: EL JUEGO DEL TOKEN

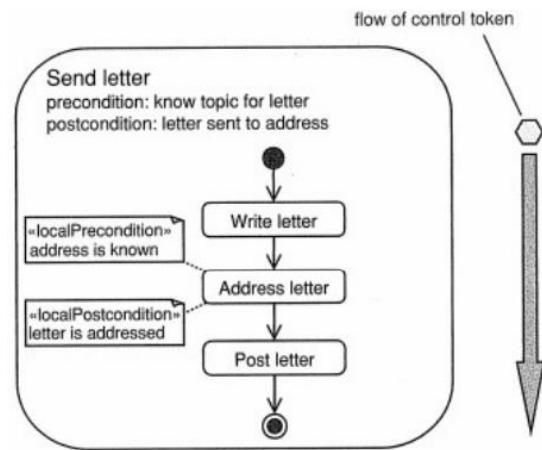
Los diagramas de actividad modelan el comportamiento usando **el juego del token**. Este juego describe el flujo de un token a través de una red de nodos y arcos de acuerdo a reglas específicas. El token en un DA puede representar el flujo de control, un objeto o algún dato.

El estado del sistema en un punto cualquiera está determinado por la posición de sus tokens.

Los tokens se mueven de un nodo de origen a un nodo de destino a través de un arco (edge). El movimiento de un token está sujeto a condiciones, y solo puede ocurrir cuando “todas” las condiciones se cumplen. Las condiciones varían dependiendo del tipo de nodo.

Para **nodos de acción**, las condiciones son:

- Las postcondiciones del nodo de origen.
- Las condiciones de guarda del flujo de control (edge/flecha).
- Las precondiciones del nodo de destino.



En criollo, un token es la forma que tiene el diagrama de actividad de mostrar en cada momento por donde está pasando el flujo o en qué estado está el flujo de lo que estoy modelando. Ejemplo del micrófono, se va pasando el micrófono y puede hablar sólo el que tiene el micrófono. A veces hay uno sólo y otras veces tenemos dos micrófonos o más. En algún punto se termina el micrófono o se une con otro micrófono.

En las bifurcaciones no tengo dos tokens, si voy por un camino el token va por ahí y si voy por el otro camino, el token sigue ese otro camino.

En cuanto a los caminos en paralelo (división y unión – fork and join), tengo dos tokens. El token se divide y se necesitan dos para seguir ambos caminos paralelos, los tokens se vuelven a unir en el nodo final si ambos llegaron correctamente por sus respectivos caminos.

**NODO DE FUSIÓN:** Copia los tokens de entrada en su extremo de salida. Puede recibir más de dos flujos, pero sólo tiene uno de salida.

## USOS DE LOS DIAGRAMAS DE ACTIVIDAD

Los diagramas de actividad pueden acompañar a cualquier elemento del modelo con el objetivo de mostrar su comportamiento.

Es muy común encontrarlos para modelar procesos de negocio cuando usamos un caso de uso, ya que estamos mostrando una funcionalidad de un software. Antes de llegar a esa funcionalidad tenemos que entender como funciona el negocio. Una actividad típica del negocio puede ser realizar una venta – eso se puede hacer con un diagrama de actividad.

El diagrama de actividad se puede usar para mostrar el flujo de actividades desde los niveles más altos (negocio) hasta los más bajos (método de una clase).

# Historias de Usuario

Las **metodologías ágiles** como Scrum utilizan las historias de usuario como el instrumento principal para identificar los requerimientos de usuario, así como con las tecnologías más tradicionales utilizamos los casos de uso. Como analogía, las historias de usuario serían casos de uso a trazo grueso.

Las **historias de usuario** son descripciones cortas y simples de una funcionalidad, escritas siempre desde la perspectiva de la persona que necesita una nueva capacidad de un sistema, por lo general el usuario, área de negocio o cliente. Su utilización es común cuando se aplican marcos de trabajo ágiles, tales como Scrum o el Extreme Programming (XP).

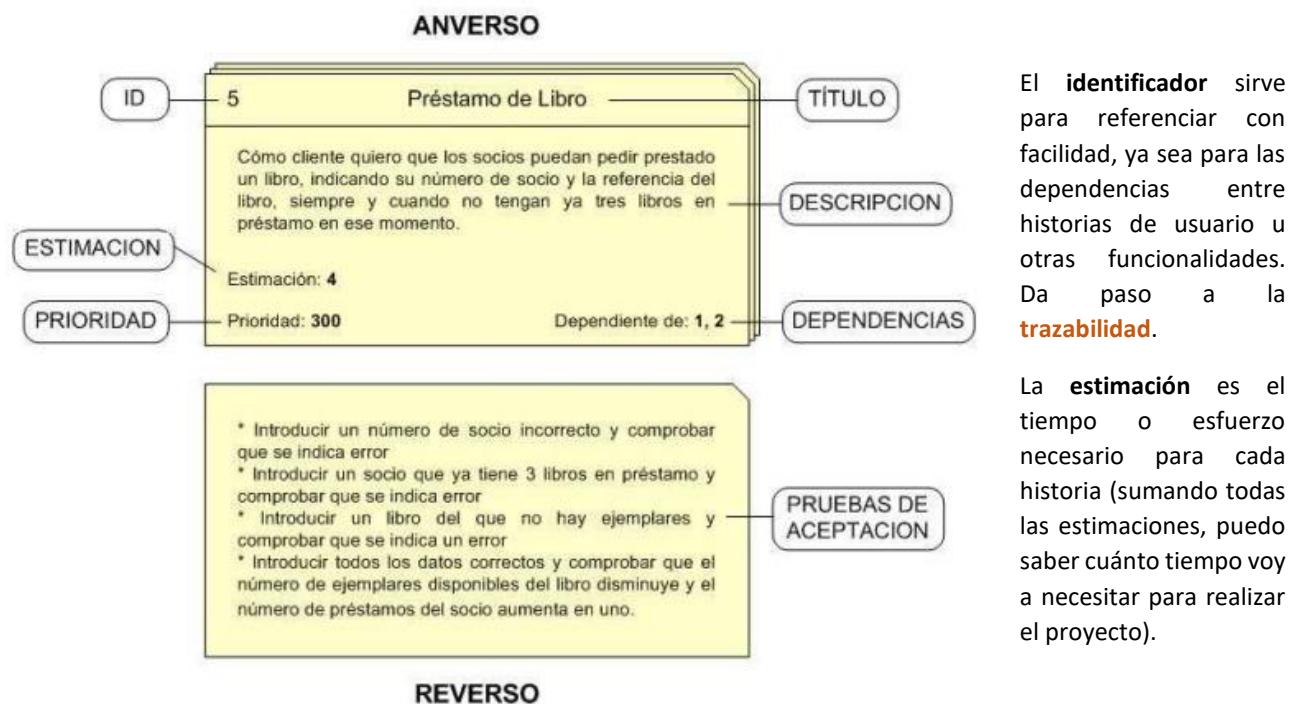
No considera el “*como*” y no representa el fin porque muchas veces las historias de usuario las realiza el usuario mismo y a partir de ahí se negocian los requerimientos.

A diferencia del enfoque tradicional, en el cual la etapa de requerimientos contiene documentación detallada del software y representa el final de las conversaciones, en las metodologías ágiles se hace uso de las “historias de usuario”, las cuales se enfocan en definir lo que el usuario necesita hacer, sin describir el cómo, por lo que representa el inicio y no el final de las conversaciones.

## REDACCIÓN

Al redactar una historia de usuario deben tenerse en cuenta describir SÍ O SÍ el rol, la funcionalidad y el resultado esperado de la aplicación en una **frase corta**.

Adicionalmente, debe venir acompañado de los criterios de aceptación, no más de 4 por historia, redactados también en una frase que incluya el contexto, el evento y el comportamiento esperado ante ese evento.



## CARACTERÍSTICAS

- Que sean escritas por el usuario o por un analista de negocio que le represente. Eso no significa que esto no lo pueda realizar el desarrollador, pero lo ideal es que lo haga el mismo usuario con sencillez qué está esperando que el software haga.
- Frase corta que encaje en una tarjeta de 3 por 5 pulgadas.
- Debe describir el rol desempeñado por el usuario (cliente) en el sistema, descrito de forma explícita.
- Debe describir el beneficio para el área de negocio que representa esta funcionalidad.

## DIFERENCIAS CON OTRAS TÉCNICAS

No debemos confundirlas con casos de uso o escenarios. La gran diferencia es que **son más cortas y no deben describir la interfaz con el usuario**, los pasos de navegación o flujos de procesos de la aplicación. En caso de uso describo la interfaz con el usuario (menú desplegable, opciones), no describo los mensajes que el software le va a mostrar al usuario, las validaciones.

## EN QUÉ CASOS UTILIZARLAS

En proyectos cortos de tiempo y teniendo que sacar un producto rápido, se va por metodologías ágiles. Sobre todo cuando los requerimientos son cambiantes es mejor esto, no puedo estar meses trabajando con ciertos requerimientos y terminan cambiando una vez se entrega. Para proyectos largos y críticos en cuanto a la cantidad de recursos necesarios, se va por metodologías tradicionales.

A pesar de esto, no hay un camino estricto a seguir. En proyectos finales de carrera se suele ir por métodos clásicos de cascada, ya que no van a ser requerimientos cambiantes en general.

## PARA QUÉ SE UTILIZAN

Su propósito principal es la estimación del esfuerzo necesario para implementar una nueva funcionalidad (feature) en un software, siguiendo la definición de “hecho” (DONE) que defina el equipo. Se confirma que una historia ha sido implementada adecuadamente y está “hecha” (DONE) por medio de la **Prueba de Aceptación**, la cual a su vez deriva de los criterios de aceptación (también denominados condiciones de satisfacción) asociados a la historia.

Adicionalmente, se utilizan como iniciadores de conversaciones entre desarrolladores de software y usuarios del área de negocio, las cuales servirán para identificar los requerimientos del negocio, requerimientos técnicos y para encontrar los supuestos (premisas) no visibles en una primera aproximación.

Para estas estimaciones, se le asignan unidades de medida a las historias que representen magnitud y no días reales de duración, tales como: puntos de historia, días ideales u otra unidad de medida.

## CONSECUENCIAS DE MALA APLICACIÓN

La elicitación de requerimientos y las pruebas en el desarrollo ágil de software dependen en gran medida de las historias de usuario, por lo que si cometemos errores al definir las mismas podemos tener problemas a lo largo del desarrollo.

Los errores cometidos al escribir las historias pueden ocasionar un mal entendimiento de los requerimientos, mala formulación de casos de pruebas, y lo que es peor, una mala implementación, ocasionando el rechazo de los entregables de una iteración.

## ASPECTOS DE UNA BUENA HISTORIA DE USUARIO

Luego de hacer la CCC (Card, Confirmation, Conversation), definitivamente el criterio más importante que se usa es cumplir **INVEST**:

- **Independiente:** No requiere de otra.
- **Negociable:** Se puede reemplazar por otra de diferente prioridad.
- **Valor:** Que sea necesaria y de valor para el proyecto.

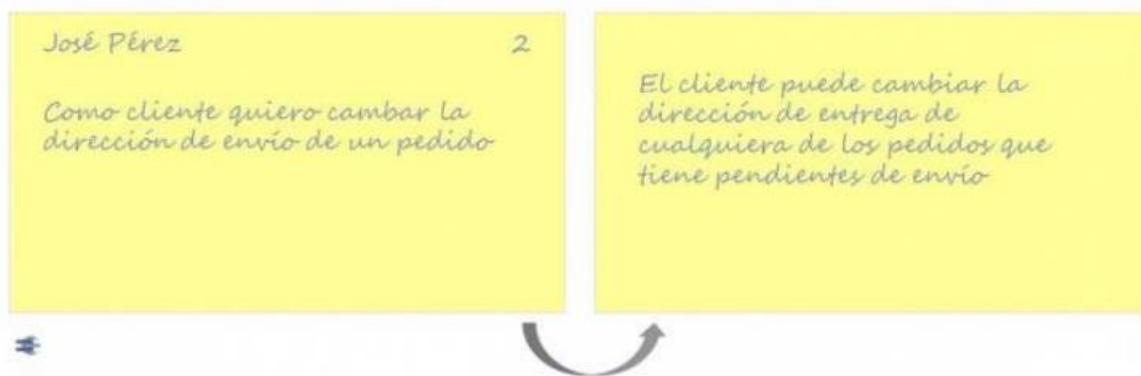
Que su implementación toque todas las capas o que el resultado de su implementación tenga sentido para el cliente (Product Owner). En caso de hacer desarrollos asociados exclusivamente a la Base de Datos o a temas complejos que no logran mostrar su utilidad para el cliente, se sugiere crear historias en las cuales se le explique a este el valor de las mismas en el proyecto. Evitar al máximo historias técnicas pues son de difícil justificación.

Preguntar lo más que se pueda sobre la necesidad de la historia de usuario. Preguntar para ciertas historias que en principio parecen innecesarias (validaciones de negocio excesivas, autocompletar recargados, etcétera):

*Es de valor, pero ¿es necesaria?, ¿cuántas personas la usarán?, ¿cuántas veces la usarán en el año? ¿Es realmente útil? ¿Prefieres al equipo trabajando en esa “validación xxxx” -por ejemplo- que en esta otra historia de usuario que agrega más valor al negocio? ¿Podemos entregar esto para el final y hacer otra cosa más urgente sobre la que tenemos clara la necesidad de implementación?*

- **Estimable:** Que el equipo se sienta tranquilo y seguro estimándola.
- **Small (pequeñas):** Que no sean grandes, funcionalidades pequeñas.  
Que el tamaño de la historia de uso no supere los 3 a 4 días de trabajo de una persona, ya que historias más grandes quedan mal estimadas por más buena intención que tenga el equipo de desarrollo; el equipo se siente seguro con este tamaño de historia. Determine cuál es el rango para su equipo.  
Que tenga a lo sumo entre 3 y 7 criterios de aceptación, puesto que más de 7 criterios se puede dividir la historia y menos de 3 se puede agrupar con otra. Siguiendo esto, si es muy grande tiene que poder dividirse. La gran mayoría de las historias grandes se pueden dividir, esto permite más maniobrabilidad al equipo al momento de implementación.
- **Testeable (verifiable):** Que se le puedan realizar pruebas.

Historia de Usuario	
Número: 1	Usuario: Cliente
Nombre historia: Cambiar dirección de envío	Nombre breve y descriptivo.
Prioridad en negocio: Alta	Riesgo en desarrollo: Baja
Puntos estimados: 2	Iteración asignada: 1
Programador responsable: José Pérez	Descripción de la funcionalidad en forma de diálogo o monólogo del usuario describiendo la funcionalidad que desea realizar.
Descripción: Quiero cambiar la dirección de envío de un pedido.	
Validación: El cliente puede cambiar la dirección de entrega de cualquiera de los pedidos que tiene pendientes de envío.	Criterio de validación y verificación que permitirá considerar terminado y aceptable por el cliente el desarrollo de la funcionalidad descrita.



## CONCEPTOS RELACIONADOS

- ❖ **Pila de producto:** Instrumento metodológico del marco de trabajo Scrum, que se usa para listar las características (features) o funcionalidades del software a desarrollar, para priorizarlas de acuerdo a las necesidades del área de negocio. Su contenido se desarrolla a partir de las historias de usuario identificadas por el dueño de producto (Product Owner).
- ❖ **Temas:** Grande proyectos, peticiones globales sin más análisis ni detalle.  
*"Buscador de ofertas de trabajo", "Backoffice para agregar ofertas de trabajo".*
- ❖ **Epics:** "Super" historias de usuario, más concretas que los temas  
*"Sistema de búsqueda por texto libre de ofertas de trabajo", "Filtros que aplicar a las búsquedas", "Presentación listado-detalle de los resultados de las búsquedas".*
- ❖ **Historias de usuario:** Manera simple de describir una tarea que aporta valor.  
*"Como candidato quiero buscar en las ofertas de trabajo para ver cuáles me interesan", "Como candidato quiero poder encontrar ofertas filtradas para obtener solo las de mi zona, mi profesión y la remuneración que yo quiera."*
- ❖ **Tarea:** Las historias de usuario pueden ser divididas en diversas tareas por necesidades técnicas.  
*"Crear UI de presentación de resultados", "Crear los métodos de consulta a BBDD que retornen los resultados", "Mostrar mensaje si no se encuentran los resultados a los criterios de búsqueda."*

# Maquetado

## WIREFRAMING

**¿Qué es?** Wireframing es una manera de diseñar un sitio web en el nivel estructural analizando las necesidades del usuario, así como la información que llevará dicho sitio.

**¿Cuál es la diferencia entre maquetado y wireframing?** No son lo mismo. El maquetado es el término general, mientras que los desarrollos web son wireframing. El **maquetado** en general es un buen complemento a la descripción breve de las historias de usuario, siendo una forma gráfica de mostrar cómo se van a ir viendo las diferentes funcionalidades y pantallas del software.

**¿Cuándo se usan?** Los wireframes se usan al inicio de un proceso de desarrollo para establecer la estructura básica de una página antes de que el diseño visual y el contenido se agregue. En este sentido, es el diseño previo para mostrar los elementos principales y qué información se va a mostrar en la interfaz.

**¿Para qué se usan?** Su objetivo es proporcionar una comprensión visual de una página de un proyecto web para obtener la aprobación de los interesados y del equipo del proyecto, antes de que el desarrollo comience. Un wireframe también se puede utilizar para crear la navegación primaria y secundaria garantizando que la estructura cumple con las expectativas del interesado.

**¿Cómo se implementan?** Dibujados a mano o formados por medio de un software para proporcionar una entrega de la pantalla. Lo mejor es crearlos en HTML básico para un test de usabilidad de prototipo.

### ¿Cuáles son sus ventajas?

- Son más fáciles de modificar que, por ejemplo, un diseño completamente realizado, ya que no llevan todo el proceso creativo y, por lo tanto, se ahorra tiempo en la definición de elementos básicos.
- Desde una perspectiva práctica, los wireframes aseguran el contenido de la página y la funcionalidad, la posición correcta de cada elemento basándose en las necesidades del usuario.
- Un usuario va a entender más si le mostramos un esquema de la pantalla que un caso de uso.

### ¿Cuáles son sus desventajas?

- Los wireframes no incluyen el diseño visual; no siempre es fácil para el cliente comprender el concepto con elementos a blanco y negro.
- El diseñador también tendrá que traducir los wireframes en un diseño, así que la comunicación para apoyarlo es a menudo necesaria para explicar por qué los elementos de la página se colocan en tal posición.
- Cuando se agrega contenido, lo que inicialmente se muestra en espacios predefinidos puede no siempre ser el espacio final que ocupará la información, por lo que el diseñador y redactor tendrá que trabajar estrechamente para hacer este ajuste.

# diseño de software

El diseño es el proceso creativo de transformación del problema en una solución, es la actividad del ciclo de vida en la que se analizan los requisitos del software para desarrollar una descripción de la estructura interna y la organización del sistema que servirá de base para su construcción.



Es por esto que el **diseño de software** comprende la descripción de la arquitectura del sistema con el nivel de detalle suficiente para guiar su construcción:

- Descomposición del sistema.
- Organización entre los componentes del sistema.
- Interfaces entre los componentes.

**TRAZABILIDAD:** Desde un caso de uso tengo que poder encontrar su representación en diseño.

Hay una división en dos grandes grupos para la etapa de diseño:

1. **Diseño de la arquitectura del software:** Descripción de la arquitectura general, identificación de sus componentes y su organización y relaciones en el sistema.
2. **Diseño detallado del software:** Definición y estructura de los componentes y datos. Definición de los interfaces. Elaboración de las estimaciones de tiempo y tamaño (refinar y tener mayor precisión en cuanto al tiempo necesario y el tamaño de nuestro software ayuda a tomar decisiones más eficientes y menos erradas).

## RAZONES DEL DISEÑO DE SOFTWARE:

- Permite la descomposición del problema en partes y vistas de menor tamaño, más manejables para el trabajo intelectual del diseño de la solución.
- Permite el desarrollo de modelos que se pueden analizar para determinar si cumplen los distintos requisitos.
- Permite examinar soluciones alternativas.
- Los modelos se pueden utilizar para planificar el desarrollo de las actividades, y son el **punto de partida** para empezar las actividades de codificación y pruebas.

## VISTAS DEL DISEÑO:

Un sistema de software es una entidad que puede contemplarse o analizarse desde diferentes “vistas”. El diseño puede generar modelos para cada una de las diferentes vistas empleadas en su análisis (modelo físico, modelo de datos, modelo de procesos, etcétera), dependiendo de cómo se enfoque la atención:

- Distribución física del software entre los diferentes elementos del sistema.
- Descomposición en las diferentes funcionalidades que realiza.
- Estructuras de la información que gestiona.
- Entre otras.

## ¿Cuándo finaliza el proceso de diseño?

Se considera que el proceso de diseño se ha completado cuando:

- Todas las preguntas “cómo” tienen respuesta.
- La descripción del diseño de la arquitectura está completada.
- La revisión del diseño se ha completado y cada equipo/persona implicado está de acuerdo con el diseño.
- Los borradores de manuales para mantenimiento y administración están realizados.
- Se ha realizado la trazabilidad del diseño.
- Se ha revisado y verificado el diseño de la arquitectura.
- Se ha escrito la planificación de la integración del software.
- Se ha establecido la línea base del producto.

## ¿Cuál es la notación empleada?

Las **notaciones** pueden variar en función de las características de cada proyecto o de los conocimientos o preferencias de las personas u organización que lo realice.

A través del lenguaje de modelo empleado (UML, IDEF, Diagramas de flujo, etcétera) se consiguen realizar dos tipos de descripciones: descripciones estructurales y descripciones del comportamiento.

- **Descripciones estructurales**

- Lenguajes de descripción de arquitecturas (ADL): AADL, AESOP,
- CODE, MetaH, Gestalt, Modechart, UML, Unicon, Modechart, etc.
- Diagramas de clases y objetos
- Diagramas de componentes
- Diagramas entidad-relación
- Lenguajes de descripción de interfaz
- Etc.

- **Descripciones de comportamiento**

- Diagramas de actividad
- Diagramas de colaboración
- Diagramas de flujo de datos
- Diagramas de flujo
- Pseudo-código y lenguajes de diseño (PDL)
- Diagramas de secuencia
- Etc.

## ¿Qué estrategias se emplean para el diseño de software?

Las principales estrategias que suelen emplearse para el diseño del software son:

- **Diseño estructurado (orientado a funciones):** Aproximación clásica, centrada en la identificación y descomposición de las principales funciones del sistema hacia niveles más detallados.
- **Diseño orientado a objetos:** Aproximación más popular actualmente, sobre la que se han desarrollado numerosos métodos. A través del diseño orientado a objetos se desarrollan las especificaciones de sistemas como modelos de objetos (sistemas compuestos por conjuntos de objetos que interactúan entre ellos). Que, expuesta de forma muy básica, identifica a los nombres como objetos, a los verbos como los comportamientos que pueden ofrecer y a los adjetivos como sus métodos.



## DESCRIPCIÓN DEL DISEÑO DE SOFTWARE (SDD)

El resultado del proceso de diseño es la documentación denominada “**Descripción del Diseño de Software**”. Un estándar empleado para desarrollar esta documentación de forma normalizada es el IEEE Std. 1016-1998:

1.- Introducción	4.- Descripción de las dependencias
1.1 Propósito	4.1 Dependencias intermodulares
1.2 Alcance	4.2 Dependencias inter-procesos
1.3 Definiciones y acrónimos	4.3 Dependencias de los datos
2.- Referencias	5.- Descripción de interfaces
3.- Descomposición de la información	5.1 Interfaces entre módulos
3.1 Descomposición modular	5.1.1 Interfaz del módulo 1
3.1.1. Descripción del módulo 1	5.1.2 Interfaz del módulo 2
3.1.2. Descripción del módulo 2	5.2 Interfaces entre procesos
3.2 Descomposición de los procesos	5.2.1 Interfaz del proceso 1
3.2.1. Descomposición del proceso 1	5.2.2 Interfaz del proceso 2
3.2.2. Descomposición del proceso 2	6.- Diseño detallado
3.3 Descomposición de los datos	6.1 Diseño detallado de los módulos
3.3.1. Descripción de la entidad 1	6.1.1 Detalle del módulo 1
3.3.2. Descripción de la entidad 2	6.1.2 Detalle del módulo 2
	6.2 Diseño detallado de los datos
	6.1.1 Detalle de la entidad 1
	6.1.2 Detalle de la entidad 2

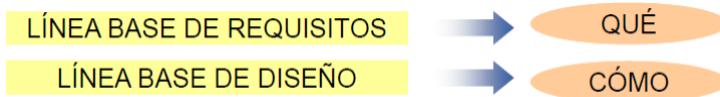
- Describe prácticas recomendadas para describir los diseños de software. Especifica la información que debe contener, y recomienda cómo organizarla.
- Puede emplearse en software comercial, científico o militar sin limitaciones por el tamaño, complejidad o nivel de criticidad.
- El estándar no establece ni limita determinadas metodologías de diseño, gestión de la configuración o aseguramiento de la calidad.

## PRÁCTICAS RECOMENDADAS EN EL IEEE

- **Trazabilidad del diseño:**
  - Comprobación de que el diseño incluye todos los requisitos.
  - Comprobación de que el diseño no incluye funciones adicionales no especificadas en el SRS.
  - Los resultados de la trazabilidad del diseño deben estar documentados para la reunión de revisión del diseño.
- **Reunión de revisión del diseño de la arquitectura:**
  - Revisión del diseño de la arquitectura.
  - Un equipo apropiado (usuarios, cliente, ingeniero de software) revisan el diseño.
  - Una vez aprobado este diseño se puede comenzar a realizar el diseño detallado.
- **Verificación del diseño de la arquitectura:**

El diseño se verifica contra el SRS:

  - El proceso de verificación analiza si el diseño es incompleto, incorrecto, inefficiente, difícil de mantener, presenta un interfaz de usuario difícil de utilizar o aprender, o la documentación es de baja calidad.
  - Se realiza un informe para documentar los posibles problemas encontrados y tomar nota de posibles incompatibilidades entre documentos.



## ARQUITECTURA DEL SOFTWARE

La arquitectura del software es el diseño de más alto nivel de la estructura de un sistema. Una **arquitectura de software**, también denominada arquitectura lógica, consiste en un conjunto de patrones y abstracciones coherentes que proporcionan un marco definido y claro para interactuar con el código fuente del software.

Se selecciona y diseña basada en los requerimientos y restricciones:

- **REQUERIMIENTOS:** Se deben tener en cuenta no solamente los de tipo funcional, también otros como la mantenibilidad, auditabilidad, flexibilidad e interacción con otros sistemas de información.
- **RESTRICCIONES:** Son aquellas limitaciones derivadas de las tecnologías disponibles para implementar sistemas de información. Unas arquitecturas son más recomendables de implementar con ciertas tecnologías mientras que otras tecnologías no son aptas para determinadas arquitecturas. Por ejemplo, no es viable emplear una arquitectura de software de tres capas para implementar sistemas en tiempo real.

La arquitectura de software, tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar cierto número de elementos arquitectónicos de forma adecuada para **satisfacer la mayor funcionalidad y requerimientos de desempeño de un sistema**, así como **requerimientos no funcionales**, como la confiabilidad.



## MODELOS O VISTAS

Toda arquitectura de software debe describir diversos aspectos del software. Generalmente, cada uno de estos aspectos se describe de una manera más comprensible si se utilizan distintos **modelos o vistas**.

Cada uno de estos modelos constituye una descripción parcial de una misma arquitectura y es deseable que exista cierto solapamiento entre ellos. Es así que cada paradigma de desarrollo exige diferente número y tipo de vistas o modelos para describir una arquitectura, dependiendo de su complejidad.

Existen al menos tres vistas absolutamente fundamentales en cualquier arquitectura:

- **La visión estática:** describe qué componentes tiene la arquitectura.
- **La visión funcional:** describe qué hace cada componente.
- **La visión dinámica:** describe cómo se comportan los componentes a lo largo del tiempo y como interactúan entre sí.

## LENGUAJE DE VISTAS Y MODELOS

Las vistas o modelos de una arquitectura de software pueden expresarse mediante uno o varios lenguajes. El más obvio es el lenguaje natural, pero existen otros lenguajes tales como los diagramas de estado, los diagramas de flujo de datos, etcétera.

Estos lenguajes son apropiados únicamente para un modelo o vista y existe cierto consenso en adoptar **UML** como lenguaje único para todos los modelos o vistas.

## ARQUITECTURAS MÁS COMUNES

No es necesario inventar una nueva arquitectura de software para cada sistema de información. Lo habitual es adoptar una arquitectura conocida en función de sus ventajas e inconvenientes para cada caso en concreto.

Las arquitecturas más universales son:

- **Descomposición modular:** el software se estructura en grupos funcionales muy acoplados.
- **Cliente-servidor:** el software reparte su carga de cómputo en dos partes independientes, pero sin reparto claro de funciones.
- **Arquitectura de tres niveles:** especialización de la arquitectura cliente-servidor donde la carga se divide en tres partes (o capas) con un reparto claro de funciones: una capa para la presentación (interfaz de usuario), otra para el cálculo (donde se encuentra modelado el negocio) y otra para el almacenamiento (persistencia). Una capa solamente tiene relación con la siguiente.

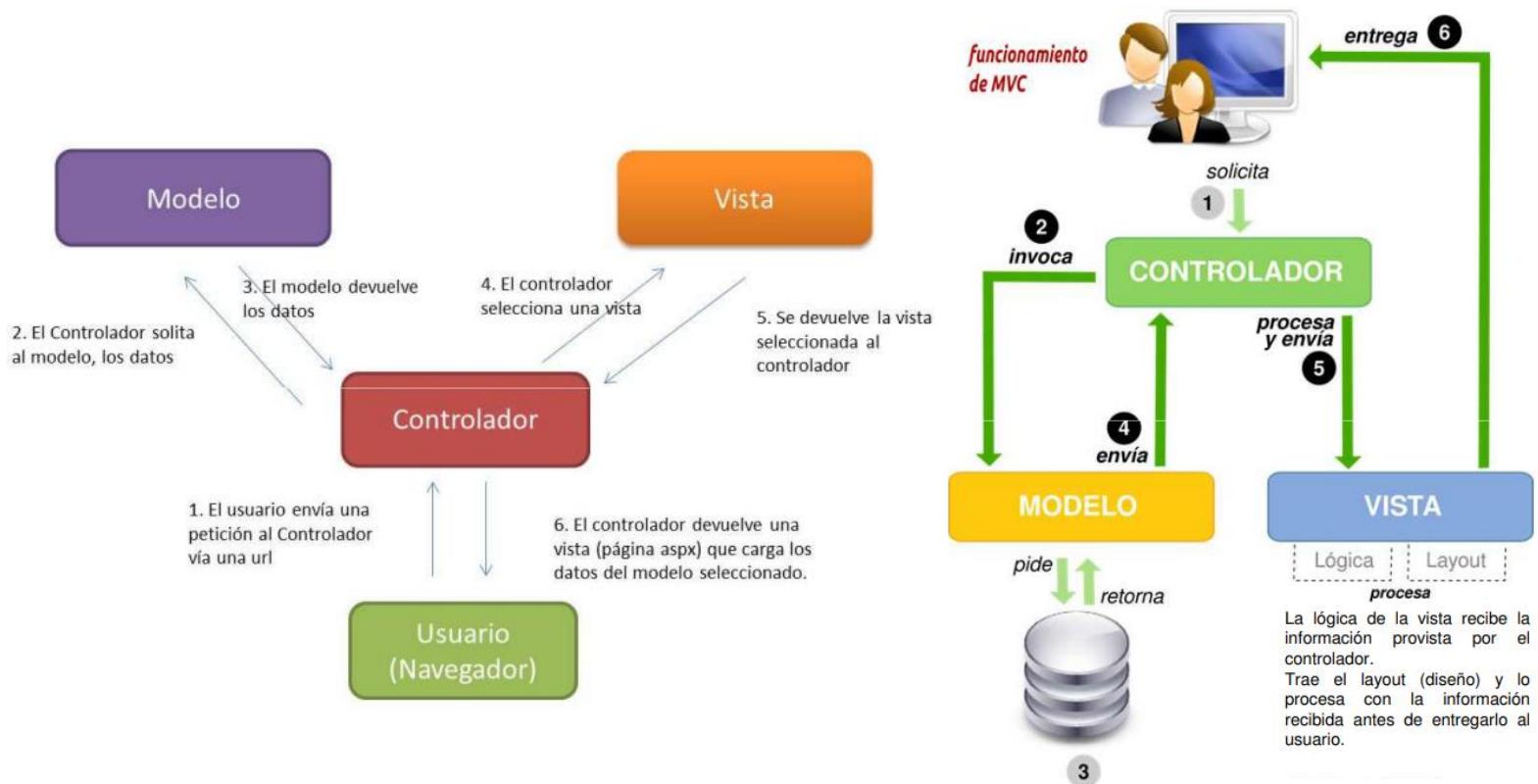
Otras arquitecturas afines menos conocidas son: **Modelo Vista Controlador**, en pipeline, entre pares, en pizarra, orientada a servicios, dirigida por eventos, máquinas virtuales.

## Modelo Vista Controlador (MVC)

El **modelo-vista-controlador (MVC)** es un patrón de arquitectura de software, que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario.

Para ello, MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador. Es decir, por un lado, define componentes para la representación de la información y, por otro lado, para la interacción del usuario.

Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.



## MVC EN APLICACIONES WEB

Aunque originalmente MVC fue desarrollado para aplicaciones de escritorio, ha sido ampliamente adaptado como arquitectura para diseñar e implementar aplicaciones web en los principales lenguajes de programación.

Se han desarrollado multitud de frameworks, comerciales y no comerciales, que implementan este patrón; estos frameworks se diferencian básicamente en la interpretación de como las funciones MVC se dividen entre cliente y servidor. Ejemplos de frameworks son JavaScriptMVC, Backbone o jQuery.

## Arquitectura Orientada a Servicios (SOA)

La **Arquitectura Orientada a Servicios** significa integración a través de sistemas diversos. Utiliza protocolos estándar e interfaces convencionales –usualmente Web Services– para facilitar el acceso a la lógica de negocios y la información entre diversos servicios.

SOA nos brinda los principios y la guía para transformar el conjunto de recursos de TI de la compañía (los cuales son, por lo general, heterogéneos, distribuidos, inflexibles y complejos) en recursos flexibles, integrados y simplificados, que pueden ser cambiados y compuestos para alinearse más fácilmente con los objetivos del negocio.

SOA es un conjunto de patrones de construcción de las nuevas aplicaciones de la empresa –más dinámicas y menos dependientes.



## MODELO ORIENTADO A OBJETOS

- **HISTORIA:** La década de los 90 fue la era de la programación orientada a objetos. Los usuarios demandan programas y entornos de trabajo simples y fáciles de usar, lo cual implica un mayor número de líneas de código que es necesario organizar, gestionar y mantener.
- **VENTAJAS:** Proporciona mejores herramientas para:
  - ✓ Obtener un modelo del mundo real cercano a la perspectiva del usuario.
  - ✓ Interaccionar fácilmente con un entorno de computación, empleando metáforas familiares.
  - ✓ Facilitar la modificación y la extensión de los componentes sin codificar de nuevo desde cero.
- **DIFERENCIA CON PROGRAMACIÓN TRADICIONAL:** La programación tradicional está orientada a los procedimientos, en la programación orientada a objetos las entidades centrales son los datos (objetos).
- **CÓMO FUNCIONA:**
  - ✓ Los objetos se comunican entre sí mediante el uso de mensajes y el conjunto de objetos que responden a los mismos mensajes se implementan mediante clases.
  - ✓ La clase describe e implementa todos los métodos que capturan el comportamiento de sus instancias.
  - ✓ La implementación está totalmente oculta (encapsulada) dentro de la clase, de modo que puede ser extendida y modificada sin afectar al usuario.
  - ✓ Una clase es como un módulo. Sin embargo, también es posible extender y especializar una clase (mecanismo de herencia).

## TERMINOLOGÍA COMÚN Y PRINCIPIOS BÁSICOS

<b>MODULARIZACIÓN</b>	Módulos fáciles de manejar y que comprenden las estructuras de datos y las operaciones permisibles.
<b>ENCAPSULADO</b>	Distingue entre la interface a un objeto (qué es lo que hace), de la implementación (cómo lo hace).
<b>TIPOS DE DATOS ABSTRACTOS</b>	Agrupa todos los objetos que tienen la misma interface y los trata como si fueran del mismo tipo.
<b>HERENCIA</b>	Reutilización, ya que permite definir nuevos tipos en funciones de otros tipos. El nuevo tipo hereda las estructuras de datos y los métodos del tipo precedente.
<b>MENSAJES</b>	Un objeto lleva a cabo sus acciones cuando recibe un mensaje concreto, codificado de una forma simple, estándar e independiente de cómo o dónde está implementado el objeto.
<b>POLIMORFISMO</b>	Diferentes objetos responden al mismo mensaje. El sistema determina en tiempo de ejecución qué código invocar dependiendo del tipo de objeto (técnicas de Overloading y Dynamic binding).



## OBJETO

Concepto, abstracción o cosa que tiene un cierto significado para una aplicación.

Se presentan como nombres propios o referencias específicas en la descripción o discusión de un problema:

- Algunos objetos tienen una entidad real (por ejemplo, un auto).
- Otros son entidades conceptuales (por ejemplo, la fórmula para resolver una ecuación de segundo grado).
- Existen objetos que se introducen por razones de implementación y carecen de equivalencia en la realidad física (por ejemplo, un árbol binario).

☞ **TRAZABILIDAD:** Cada objeto tiene existencia propia y puede ser identificado. Se ha definido la identidad como: *“aquella propiedad de un objeto que lo distingue del resto de objetos”* y es una noción muy importante de la tecnología orientada a objetos.

En la etapa de análisis o en el mismo diseño a alto nivel, es posible dar por supuesto que los objetos tienen identidad.

En el diseño detallado o ya en la implementación, es necesario adoptar una metodología para implementar dicha identidad.

☞ **DIAGRAMAS DE INSTANCIA:** Los objetos son instancias de una clase. Éstos y sus relaciones se describen mediante diagramas de instancia:

- unArbolBinario pertenece a la clase ArbolBinario y no se han especificado los valores de los atributos.
- El objeto IAH pertenece a la clase Aeropuerto y tiene los valores IAH, Intercont y Central.

<b>barcelona:Ciudad</b> nombreCiudad=Barcelona población=6000000	<b>sim143:Simulación</b> descripción=Normal fechaEjecución=14-Dic-1999 convergencia=false
<b>unArbolBinario:ArbolBinario</b>	
<b>IAH:Aeropuerto</b> códigoAeropuerto=IAH nombreAeropuerto=Intercont zonaHoraria=Central	<b>MAD:Aeropuerto</b> códigoAeropuerto=MAD nombreAeropuerto=Barajas zonaHoraria=MET

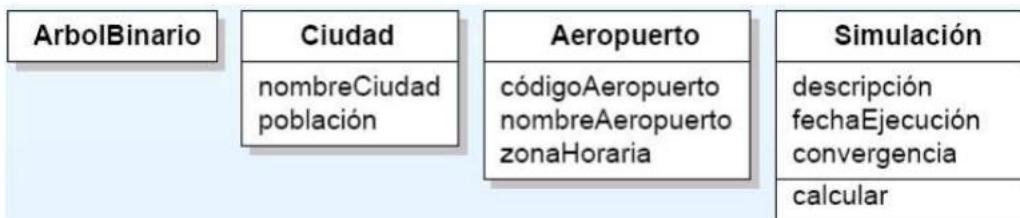


## CLASES

Descripción de un grupo de objetos con propiedades similares, comportamiento y semántica común, y que establecen el mismo tipo de relaciones con otros objetos.

Las **clases** proporcionan un mecanismo para compartir la estructura entre objetos similares. Algunas clases tienen una contrapartida real (persona y empresa) pero otras son entidades conceptuales (ecuación algebráica). Además, existen clases que son sólo artefactos de una implementación específica (árbol binario).

- Como los objetos, las clases y sus relaciones se describen mediante **diagramas de clases**:



- Del mismo modo que una clase describe un conjunto de objetos, un diagrama de clases describe un conjunto de diagrama de instancias. Un diagrama de clases permite representar de forma abstracta el conjunto de diagrama de instancias, documentando la estructura de los datos.



## VALORES Y ATRIBUTOS

Un **valor** es un trozo de información (dato), mientras que un **atributo** es una propiedad de una clase a la que se le asigna un nombre y que contiene un valor para cada objeto de la clase.

- Los modelos orientados a objetos se construyen sobre estructuras: clases y relaciones. Los atributos tienen una importancia menor y se utilizan para elaborar las clases y las relaciones.
- DIFERENCIAS:** Es importante no confundir valores y objetos: Los objetos tienen identidad mientras que los valores no.



## OPERACIONES Y MÉTODOS

Una **operación** es una función o procedimiento que se puede aplicar por un objeto o sobre un objeto. Cada operación actúa sobre un objeto que es su argumento implícito.

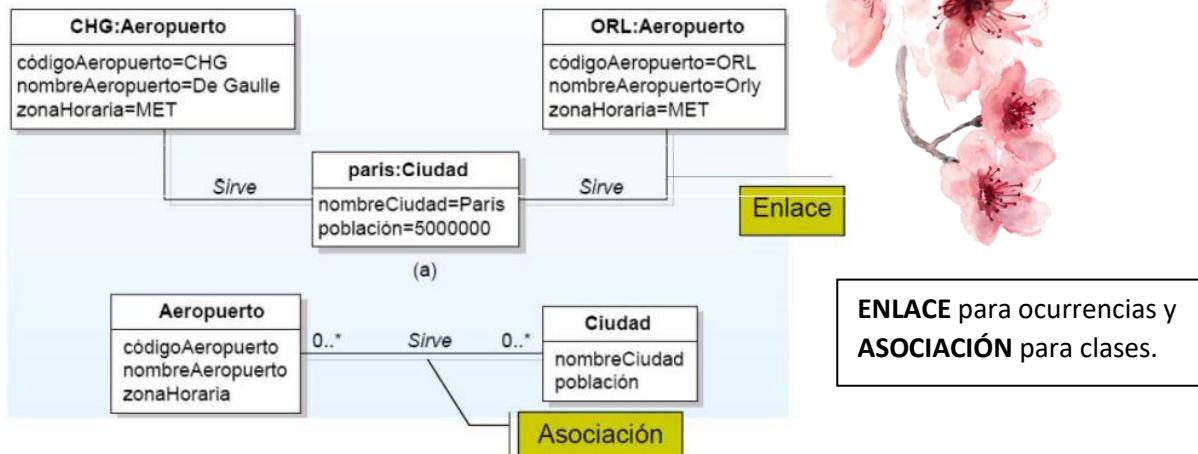
- Al nombre de la operación se le pueden añadir detalles opcionales tales como la lista de argumentos o el tipo del resultado.
- Algunas operaciones son polimórficas, esto es, aplicables a muchas clases. La implementación de una operación para una clase concreta se denomina **método**.
- DIFERENCIAS:** Las operaciones pueden ser polimórficas, los métodos son siempre para clases singulares. Esto es, un método es una operación pero no toda operación es un método.

<b>Simulación</b>
descripción
fechaEjecución
convergencia
<b>calcular</b>

La clase Simulación contiene la operación calcular cuyo argumento implícito es un objeto de la clase Simulación.



# ENLACE Y ASOCIACIÓN

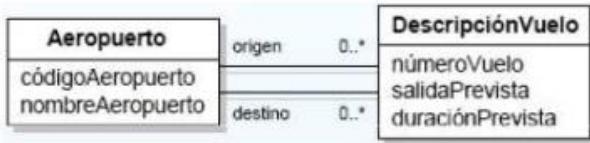


- La **multiplicidad** especifica el número de instancias de una clase que se puede relacionar con una instancia de la clase relacionada:



- Un **rol** es uno de los extremos de una asociación al que se le puede asignar un nombre. Los nombres del rol son especialmente útiles en el recorrido de las asociaciones debido a que se pueden tratar como pseudoatributos:

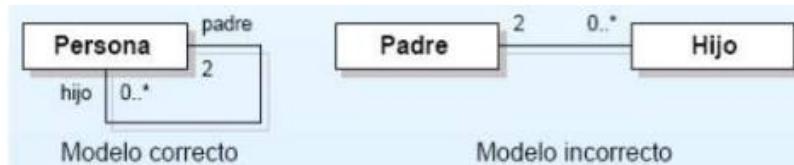
- una DescripciónVuelo.origen hace referencia al aeropuerto de origen del vuelo.
- una DescripciónVuelo.destino hace referencia al aeropuerto en el que finaliza el vuelo.



Ya que el rol es un pseudo-atributo, el nombre del rol no puede entrar en contradicción con cualquier otro atributo o rol de la clase que lo origina.

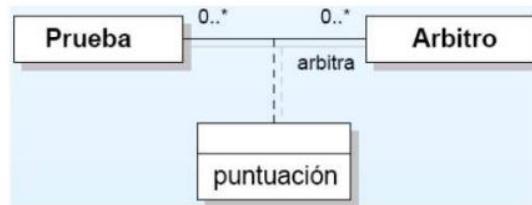
Los nombres de rol son opcionales si el modelo no es ambigüo. La ambigüedad puede aparecer cuando:

- Se dan múltiples asociaciones entre las mismas clases, como en la figura anterior.
- Cuando la asociación tiene lugar entre objetos de la misma clase (asociación reflexiva).



- El **atributo de un enlace** es una propiedad de una asociación con nombre que describe un valor contenido por cada enlace de la asociación.

Un atributo de enlace es una propiedad de la asociación que no se puede adscribir a ninguna de las clases asociadas.



#### EJEMPLO:

Un árbitro asigna una puntuación a los esfuerzos realizados por un competidor de una prueba de decatlón. La puntuación es el atributo de la asociación entre las clases Prueba y Árbitro.

Las asociaciones muchos-a-muchos proporcionan el argumento más convincente y la razón fundamental de los atributos de enlace:

- Un atributo de este tipo es sin duda alguna del enlace y no puede asignarse a ninguno de los dos objetos relacionados. La puntuación depende tanto del atleta como del árbitro.



## AGREGACIÓN

La **agregación** es una relación del tipo parte-todo entre dos clases. UML permite definir dos tipos de agregación:

- **Agregación simple:** también denominada agregación compartida. Permite modelar una relación parte-todo en la cual un objeto es propietario de otro objeto, pero no en exclusividad. El objeto poseído puede ser a su vez poseído por otros objetos.

No cambia el significado de la navegación entre el todo y sus partes. No liga la vida del todo y las partes.

**GRÁFICA:** UML representa la agregación simple mediante una figura en forma de diamante hueco sobre el extremo de la línea de la asociación que termina en la clase propietaria:



- **Composición:** También denominada en ocasiones agregación fuerte o agregación compuesta. Permite modelar una relación parte-todo en la que un objeto es propietario en exclusiva del otro objeto.

Representa una fuerte relación de pertenencia y vidas coincidentes de la parte y el todo. Las partes se crean después del objeto al que pertenecen y una vez creadas, viven y mueren con él.

Una parte puede formar parte de sólo un objeto compuesto. El objeto compuesto debe gestionar la creación y destrucción de las partes.

**GRÁFICA:** La composición se representa mediante un diamante sólido sobre el extremo de la línea de la asociación que termina en la clase propietaria.

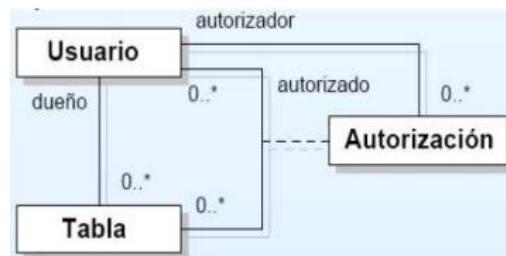




# ASOCIACIÓN

Una **clase de asociación** es una asociación cuyos enlaces pueden participar en asociaciones posteriores.

Una clase de asociación tiene características de asociación y de clase. Como el enlace de una asociación, las instancias de una clase de asociación obtienen su identidad de las instancias de las clases que las constituyen. Como una clase, pueden participar en asociaciones.



## ASOCIACIONES TERNARIAS:

- El **grado de asociación** es el número de roles de cada enlace. La gran mayoría de las asociaciones son binarias o binarias cualificadas.
- Una asociación ternaria es una asociación con tres roles que no pueden representarse mediante asociaciones binarias. Estas son poco frecuentes y es muy difícil encontrar asociaciones de grado mayor.



# GENERALIZACIÓN

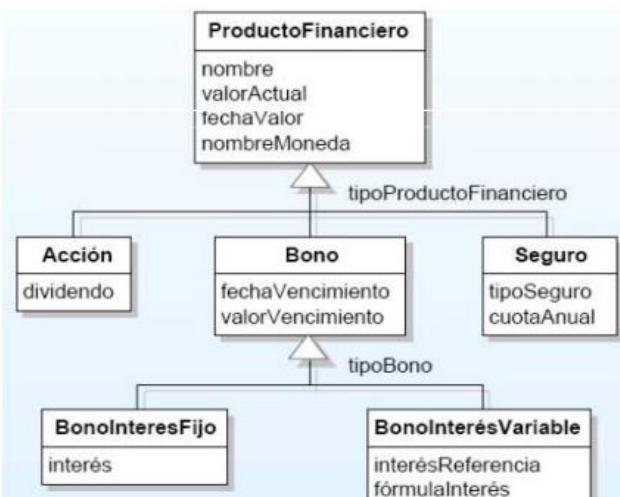
Una **generalización** es la relación entre una clase (la superclase) y una o más variaciones de esta clase (las subclases).

## CARACTERÍSTICAS:

- Organiza las clases de acuerdo con sus similitudes y diferencias.
- Proporciona estructura a la descripción de los objetos.
- La superclase contiene los atributos, operaciones, diagramas de estado y asociaciones comunes.
- La subclase añade atributos, operaciones, diagramas de estado y asociaciones específicos.
- Una instancia de una subclase es también una instancia de todas sus superclases.

**ESPECIALIZACIÓN:** Proporciona otra perspectiva de la estructura del sistema. Tiene el mismo significado que generalización, pero aporta una visión de arriba abajo.

**La generalización** es la relación estructural que permite la existencia del mecanismo de **HERENCIA**.  
Una subclase hereda los atributos, operaciones, diagramas de estado y asociaciones de su superclase.  
Las propiedades heredadas pueden reutilizarse o redefinirse en la subclase, y la subclase puede definir nuevas propiedades no presentes en la superclase.  
El caso en el que una subclase tiene múltiples superclases



La **GENERALIZACIÓN SIMPLE** (o **HERENCIA SIMPLE**) organiza las clases en una jerarquía en la que cada subclase tiene una única superclase inmediata.



# PATRONES "GRASP"

**RESPONSABILIDAD:** "Contrato u obligación de un tipo o clase" Las responsabilidades se relacionan con las obligaciones de un objeto respecto a su comportamiento. Tienen dos categorías:

- **RESPONSABILIDADES DE HACER:**
  - Hacer algo en uno mismo.
  - Iniciar una acción en otros objetos.
  - Controlar y coordinar actividades en otros objetos.
- **RESPONSABILIDADES DE CONOCER:**
  - Estar enterado de los datos privados encapsulados.
  - Estar enterado de la existencia de objetos conexos.
  - Estar enterado de cosas que se pueden derivar o calcular.

## PATRON

El **patrón** es una pareja de problema/solución con un nombre, y que es aplicable a otros contextos, con una sugerencia sobre la manera de usarlo en situaciones nuevas.

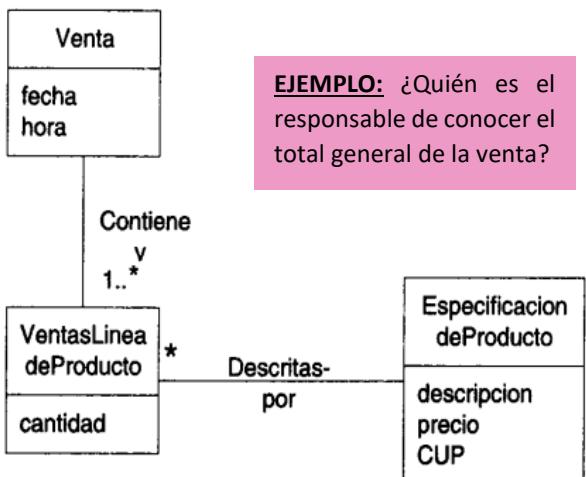
"GRASP" es acrónimo de "General Responsibility Assignment Software Patterns" (Patrones de Software para la Asignación General de Responsabilidad). Los **patrones "GRASP"** describen los principios fundamentales de diseño de objetos para la asignación de responsabilidades.

Se pueden destacar 5 patrones principales que son: *experto, creador, alta cohesión, bajo acoplamiento, controlador*. Otros 4 patrones GRASP adicionales son: fabricación pura, polimorfismo, indirección, no hables con extraños.

### 1 - EXPERTO

La responsabilidad de realizar una labor es de la clase que tiene o puede tener los datos involucrados (atributos). Una clase contiene toda la información necesaria para realizar la labor que tiene encomendada.

(!!!) Hay que tener en cuenta que esto es aplicable mientras estemos considerando los mismos aspectos del sistema: lógica de negocio, persistencia a la base de datos e interfaz de usuario.



**EJEMPLO:** ¿Quién es el responsable de conocer el total general de la venta?

Clase	Responsabilidad
Venta	Conoce el total de la venta.
VentasLíneaDeProducto	Conoce el subtotal de la línea del producto.
EspecificaciónDeProducto	Conoce el precio del producto.

Diagrama detallado de las responsabilidades:

- Venta: Conoce el total de la venta.
- VentasLíneaDeProducto: Conoce el subtotal de la línea del producto.
- EspecificaciónDeProducto: Conoce el precio del producto.

Relaciones entre las clases:

- Venta contiene VentasLíneaDeProducto.
- VentasLíneaDeProducto contiene EspecificaciónDeProducto.

## 2 - CREADOR

Se asigna la responsabilidad de que una clase B cree un objeto de la clase A solamente cuando:

- B contiene a A.
- B es una agregación (o composición) de A.
- B almacena a A.
- B tiene los datos de inicialización de A (datos que requiere su constructor).
- B usa a A.

La creación de instancias es una de las actividades más comunes en un sistema orientado a objetos. Si se asignan bien, el diseño puede soportar un bajo acoplamiento, mayor claridad, encapsulación y reutilización.

**EJEMPLO:** ¿Quién debería encargarse de crear una instancia VentasLineadeProducto? Deberíamos buscar una clase que agregue, contenga y realice otras operaciones sobre este tipo de instancias. → ¿Quién debería ser el responsable de crear una nueva instancia de una clase?

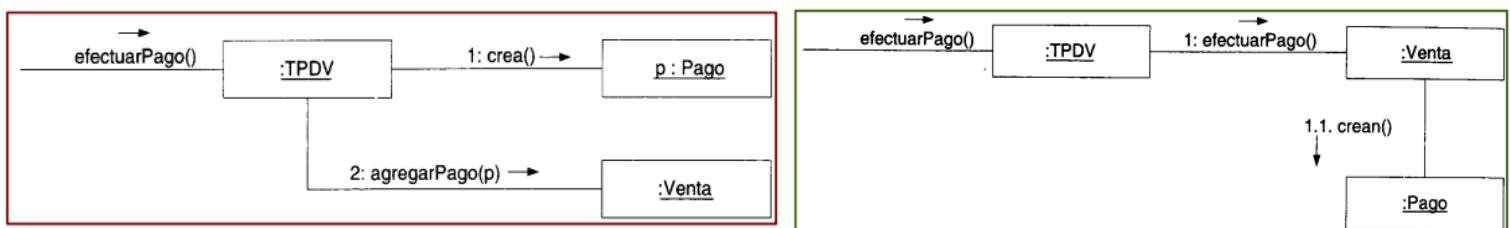


## 3 - BAJO ACOPLAMIENTO

Debe haber pocas dependencias entre las clases. Si todas las clases dependen de todas, ¿cuánto software podemos extraer de un modo independiente y reutilizarlo en otro proyecto? Uno de los principales síntomas de un mal diseño y alto acoplamiento es una herencia muy profunda.

Siempre hay que considerar las ventajas de la delegación respecto de la herencia.

**EJEMPLO:** Necesitamos crear una instancia de Pago y asociarla a Venta → ¿Qué clase se encargará de esto?



- El **diseño 1**, donde la instancia TPDV crea Pago, incorpora el acoplamiento de TPDV a Pago.
- El **diseño 2**, donde la Venta realiza la creación de un Pago, no incrementa el acoplamiento. Tomando como única perspectiva la del acoplamiento, el diseño 2 es preferible porque se conserva un menor acoplamiento global.

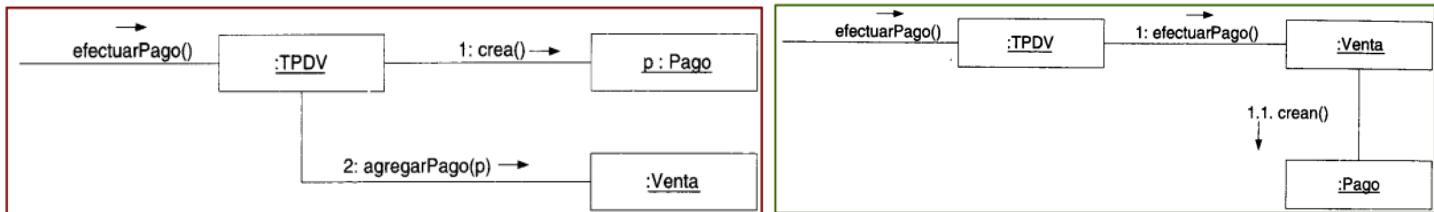
Este es un ejemplo donde dos patrones —Bajo Acoplamiento y Creador— pueden sugerir soluciones distintas. En la práctica, el grado de acoplamiento no puede considerarse aisladamente de otros principios como Experto y Alta Cohesión. Sin embargo, es un factor a considerar cuando se intente mejorar un diseño.

## 4 - ALTA COHESIÓN

Cada elemento de nuestro diseño debe realizar una labor única dentro del sistema, no desempeñada por el resto de los elementos y auto-identifiable.

- Ejemplos de una baja cohesión son clases que hacen demasiadas cosas.
- Ejemplos de buen diseño se producen cuando se crean los denominados “paquetes de servicio” o clases agrupadas por funcionalidades que son fácilmente reutilizables (bien por uso directo o por herencia).

**EJEMPLO:** Necesitamos crear una instancia de Pago y asociarla a Venta → ¿Qué clase se encargará de esto?



- Con la asignación de responsabilidades del **diseño 1**, la clase TPDV asume parte de la responsabilidad de realizar la operación del sistema.  
**PERO:** En este ejemplo aislado, lo anterior es aceptable. Pero si seguimos haciendo que la clase TPDV se encargue de efectuar la mayor parte del trabajo que se relaciona con un número creciente de operaciones del sistema, se irá **saturando** con tareas y terminará por **perder la cohesión**.
- El **diseño 2** delega a la Venta la responsabilidad de crear el pago, que soporta una mayor cohesión de TPDV. El segundo diseño brinda soporte a una alta cohesión y a un bajo acoplamiento.

## 5 - CONTROLADOR

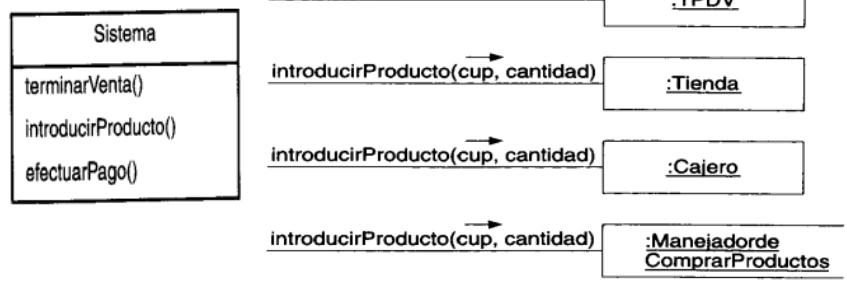
Asignar la responsabilidad de controlar el flujo de eventos del sistema, a clases específicas. Esto facilita la centralización de actividades (validaciones, seguridad, etc.).

El controlador no realiza estas actividades, las delega en otras clases con las que mantiene un modelo de alta cohesión. Un error muy común es asignarle demasiada responsabilidad y alto nivel de acoplamiento con el resto de los componentes del sistema.

**EJEMPLO:** En la aplicación del punto de venta se dan varias operaciones del sistema. → ¿Quién debería encargarse de atender los eventos del sistema?

Existen cuatro opciones válidas:

- El “sistema” global (controlador de fachada).
- La empresa u organización global (controlador de fachada).
- Algo en el mundo real que es activo y que pueda participar en la tarea (controlador de tareas).
- Un manejador artificial de todos los eventos del sistema de un caso de uso, generalmente denominado “Manejador<NombreCasoUso>” (controlador de caso de uso).





# DIAGRAMAS DE ESTADO

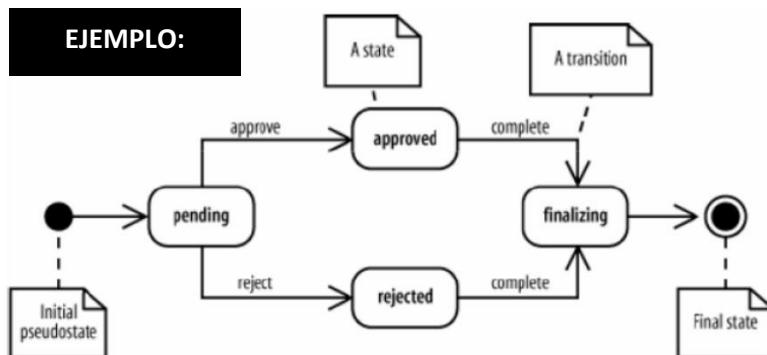
Muestra la **secuencia de estados** por los que pasa un caso de uso, un objeto a lo largo de su vida, o bien todo el sistema. En él se indican qué eventos hacen que se pase de un estado a otro y cuáles son las respuestas y acciones que genera.

- **¿PARA QUÉ SIRVEN?** Son útiles para modelar la vida de un objeto.
- **¿QUÉ MUESTRAN?** Un diagrama de estados muestra el flujo de control entre estados (en qué estados posibles puede estar “cierto algo” y cómo se producen los cambios entre dichos estados).

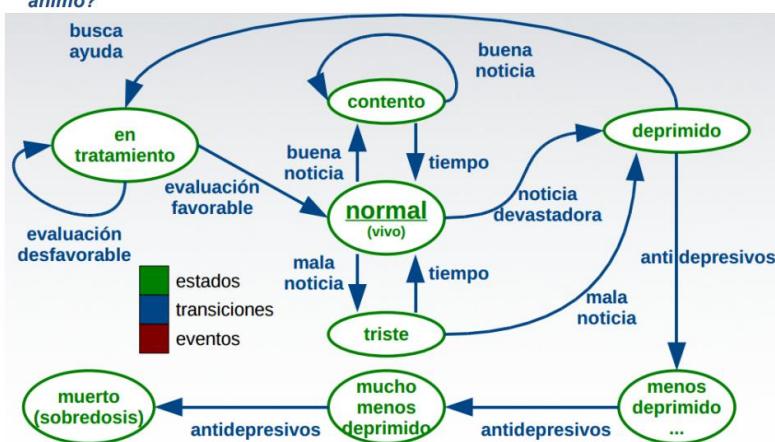
→ **ESTADO** Es una **condición o situación** en la vida de un objeto durante la cual **satisface** una condición, realiza alguna **actividad** o **espera** algún evento.

→ **EVENTO** Es la especificación de un **acontecimiento significativo** que ocupa un lugar en el **tiempo** y en el **espacio**. Es la aparición de un estímulo que puede (o no) activar una transición de estado.

→ **TRANSICION** Es una **relación** entre dos estados que indica que un objeto que esté en el **primer estado** realizará ciertas **acciones** y **entrará** en el **segundo estado** cuando ocurra un **evento** especificado y se satisfagan unas **condiciones** especificadas.



¿En qué estado (de ánimo) se encuentra usted y como cambia su estado de ánimo?



El **diagrama de estado** es un grafo cuyos nodos son estados y los arcos dirigidos son transiciones etiquetadas con los nombres de los eventos.

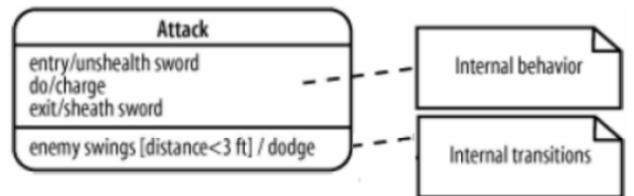
Un **estado** se representa como una caja redondeada con el nombre del estado en el interior.

Una **transición** se representa como una flecha desde el estado origen al estado destino.

La caja de un estado puede tener 1 o 2 compartimentos. En el primer compartimento aparece el nombre del estado. El segundo compartimento es opcional, y en él pueden aparecer acciones de entrada, de salida y acciones internas.

## ► **COMPORTAMIENTO** Describe las acciones que se producen mientras que el sistema se encuentra en un estado:

- **Entry/behavior:** Acción que se realiza cuando se llega a un estado.
- **Do/behavior:** Actividad que se ejecuta mientras se está en un estado.
- **Exit/behavior:** Acciones que se ejecutan cuando se abandona un estado.
- **Transiciones internas:** Se formulan como trigger[guard]/behavior. Se ejecuta cuando se cumple una condición de guarda.



"Entry" y "Exit" hacen que pase de un estado a otro, mientras que "Do" no tiene transición.



## INTERFAZ GRÁFICA DE USUARIO (GUI)

Una **interfaz gráfica de usuario** es un programa o entorno que gestiona la interacción con el usuario basándose en relaciones visuales como íconos, menús o un puntero.

- **¿CÓMO SURGIERON?** Surgieron en los años 70 (por Xerox para la investigación en las universidades, no con un fin comercial) pero se hicieron famosas en los 80 gracias a Apple y sus ordenadores (sí comerciales, con mucha relevancia), posteriormente copiadas por Microsoft con sus "Windows".
- **¿POR QUÉ ES FUNDAMENTAL?** Los usuarios de sistema frecuentemente juzgan un sistema por su interfaz. Un diseño de interfaz pobre puede provocar que el usuario cometa errores catastróficos. El diseño de una interfaz de usuario pobre es la razón por la cual muchos sistemas no son usados.
- En un sistema bien diseñado, los elementos que componen la interfaz son funcionalmente independientes y están conectados de forma indirecta al programa. De esta forma, se puede personalizar con "skins" a gusto del usuario.
- **¿CUÁLES SON SUS VENTAJAS?**
  - **Son fáciles de aprender y usar.** Usuarios sin experiencia pueden aprender el uso del sistema rápidamente.
  - El usuario puede **cambiar rápidamente desde un proceso a otro** y puede **interactuar con diferentes aplicaciones a la vez**. La información aparece visible en su propia ventana cuando la atención cambia.
  - **Rápido, interacción de pantalla completa** es posible con **acceso inmediato** a cualquier cosa sobre la pantalla.

## PRINCIPIOS DE DISEÑO:

- **Familiaridad:** La interfaz debe utilizar término y conceptos que se toman del usuario, de la experiencia de las personas que más utilizan el sistema. Por ejemplo, el tango-gestión (los usuarios usaban las teclas F1, F2, F3 para todo; el software se hizo en base a esa costumbre para evitar que tengan que aprender todo de nuevo).
- **Consistencia:** Siempre que sea posible, la interfaz debe ser consistente en el sentido de que las operaciones comparables se activan de la misma forma.
- **Recuperabilidad:** La interfaz debe incluir mecanismos para permitir a los usuarios recuperarse de los errores. Un ejemplo puede ser el ctrl + z, se debe poder retroceder ante el error para acortar el margen de error significativamente.
- **Mínima sorpresa:** El comportamiento del sistema no debe provocar sorpresa a los usuarios.
- **Guía al usuario:** Cuando los errores ocurren, la interfaz debe proveer retroalimentación significativa y características de ayuda sensible al contexto.
- **Diversidad:** La interfaz debe proveer características de interacción apropiada de usuarios para los diferentes tipos de usuarios del sistema (es decir, accesibilidad).

## MODOS DE INTERACCIÓN DE USUARIO:

- **Manipulación directa:** El usuario interactúa directamente con los objetos de la pantalla. Requiere de un dispositivo apuntador. Ejemplo: videojuegos, sistemas CAD.
- **Selección de menús:** El usuario selecciona un comando de una lista de posibilidades (un menú). Ejemplo: la mayoría de los sistemas de propósito general.
- **Relleno de formularios:** El usuario rellena los campos de un formulario. Ejemplo: Procesamiento de préstamos personales.
- **Lenguajes de comandos:** El usuario emite un comando especial y los parámetros asociados para indicarle al sistema qué hacer. Ejemplo: Sistemas operativos.
- **Lenguaje natural:** El usuario emite un comando en lenguaje natural, que se analiza y traduce a comandos del sistema. Ejemplo: Sistemas de recuperación de información.

## PRESENTACIÓN DE LA INFORMACIÓN

Todos los sistemas interactivos tienen que proporcionar alguna forma de presentar la información a los usuarios. Una buena pauta de diseño es mantener separado el software requerido para la presentación de la información misma. Para esto se utiliza el enfoque “MVC: Modelo – Vista – Controlador”.

Para encontrar la mejor presentación de la información, hay que tener en cuenta:

1. ¿Información precisa o relaciones entre los valores de los datos?
2. Frecuencia de cambio de los valores de la información.
3. ¿El usuario debe llevar a cabo alguna acción en respuesta a los cambios de información?
4. ¿El usuario necesita interactuar con la información visualizada?
5. ¿información textual o numérica?

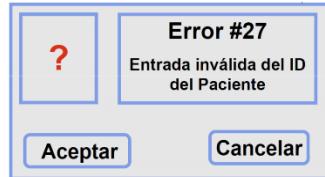
### ☞ RECOMENDACIONES DE SCHNEIDERMAN: USO DEL COLOR

1. Limitar el número de colores utilizados y ser conservador al momento de utilizarlos.
2. Utilizar un cambio de color para mostrar un cambio en el estado del sistema.
3. Utilizar el código de colores para apoyar la tarea que los usuarios están tratando de llevar a cabo.
4. Utilizar el código de colores en forma consistente y consciente.
5. Ser cuidadoso al momento de usar pares de colores (combinaciones de colores).

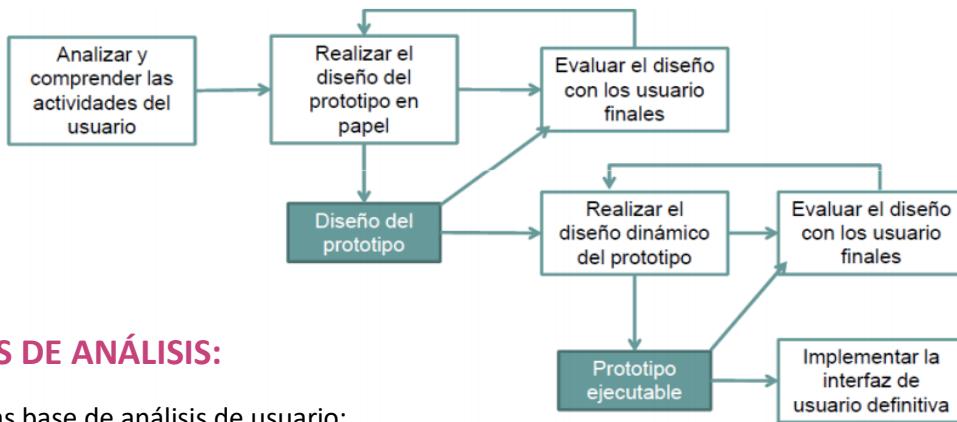
- ☞ **SOPORTE AL USUARIO (GUÍA AL USUARIO):** Se entiende por soporte al usuario a los mensajes producidos por el sistema en respuesta a las acciones de los usuarios, sistemas de ayuda en línea (contextual) y la documentación suministrada con el sistema.

Los mensajes deben tener ciertos factores de diseño:

- **Contexto:** El sistema guía del usuario debe estar pendiente de lo que hace el usuario y ajustar el mensaje de salida al contexto actual.
- **Experiencia:** Al aumentar la familiaridad de los usuarios con el sistema, también se aumenta su molestia por mensajes largos y sin “significado”. Sin embargo, los principiantes tienen dificultades en comprender mensajes concisos del problema.
- **Nivel de habilidad:** Los mensajes se deben ajustar a las habilidades del usuario, así como a su experiencia. Los mensajes para las diferentes clases de usuario se pueden expresar de diferentes formas dependiendo de la terminología que el lector utiliza.
- **Estilo:** Los mensajes deben ser positivos en lugar de negativos, deben estar escritos en estilo lingüístico activo y no en pasivo, no deben ser insultantes o tratar de ser chistosos (ser directos, pero mantener formalidad y respeto).
- **Cultura:** En la medida de lo posible, el diseñador de mensajes debe estar familiarizado con la cultura del país donde el sistema se vende. Hay diferencias culturales entre Europa, Asia y América (mensaje adecuado en una cultura puede ser no aceptado en otra).



## PROCESO DE DISEÑO DE LA INTERFAZ DE USUARIO



### TECNICAS DE ANÁLISIS:

Tres técnicas base de análisis de usuario:

- **Análisis de tareas:** Se centran en el individuo y su trabajo. La más conocida es el HTA (Análisis Jerárquico de Tareas): tareas de alto nivel se subdividen en subtareas, y se hacen planes indicando qué pasaría en una situación específica.
- **Entrevistas y cuestionarios.**
- **Relleno de formularios:** Es una técnica de apoyo a la Ingeniería de Requerimientos. Los etnógrafos observan de cerca cómo trabajan las personas, cómo se relacionan entre sí y cómo utilizan los recursos en su lugar de trabajo.

### PROTOTIPADO DE LA INTERFAZ DE USUARIO:

- Implicar al usuario en el proceso de diseño y desarrollo es un aspecto fundamental del **diseño centrado en el usuario**, un criterio de diseño para sistemas interactivos.
- **¿CUÁL ES SU PROPÓSITO?** Es permitir a los usuarios adquirir una experiencia directa con la interfaz.
- Debe ser un proceso en etapas con prototipos iniciales basados en versiones en papel de la interfaz que, después de una evaluación y retroalimentación inicial, se utilizan como base para prototipos automatizados.

#### ENFOQUES PARA EL PROTOTIPADO DE INTERFACES DE USUARIO:

1. **Enfoque dirigido por secuencias de comandos:** Se crean pantallas con elementos visuales (botones, menús, etc.) y se asocia secuencias de comandos a estos elementos. Cuando el usuario interactúa, les muestra los resultados de sus acciones. Ej: Macromedia Director.
2. **Lenguajes de programación visuales.**
3. **Prototipo basado en Internet.**

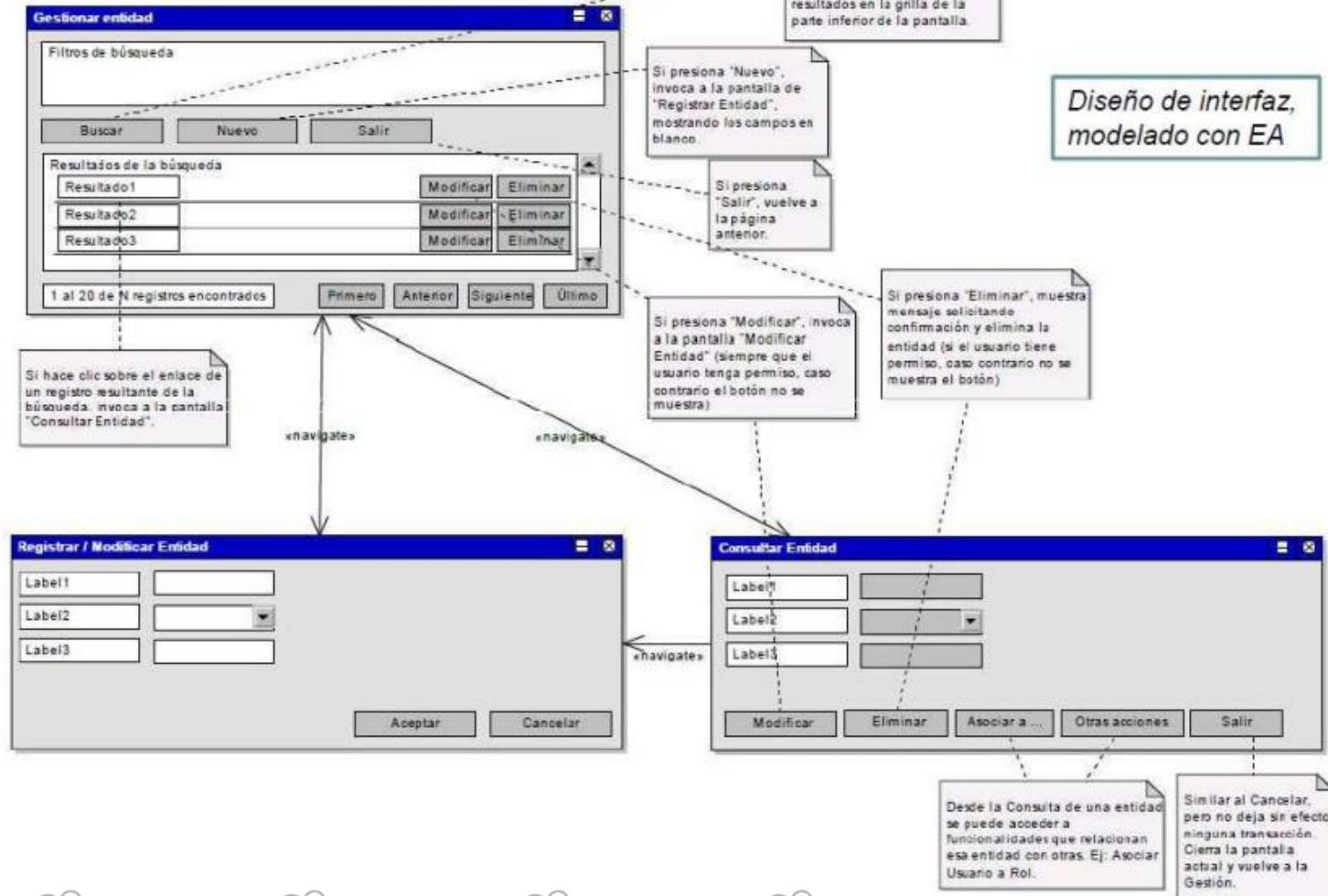
### EVALUACIÓN DE LA INTERFAZ:

Es el proceso de evaluar la forma en que se utiliza una interfaz y verificar que cumple los requerimientos del usuario. Es parte del proceso de Verificación y Validación del Software.

#### ENFOQUES PARA EL PROTOTIPADO DE INTERFACES DE USUARIO:

1. Cuestionarios sobre opinión de los usuarios de la interfaz.
2. Observación de los usuarios cuando trabajan con el sistema.
3. "Instantáneas" de videos del uso típico del sistema.
4. Inclusión de código en el software que recopile información de recursos más utilizados y errores más comunes.

Diseño de un ABM genérico.



## Verificación y validación

La **Verificación y Validación (V & V)** es el nombre dado a los procesos que permiten asegurar que el programa desarrollado satisface su especificación y brinda la funcionalidad esperada por las personas que pagan por el software. Los dos conceptos se relacionan pero son diferentes, esto se ve de la siguiente manera:

- **Validación:** ¿Estamos construyendo el producto correcto?

Mostrar que un programa o producto intermedio de software cumple con su especificación.

- **Verificación:** ¿Estamos construyendo el producto correctamente?

Mostrar que el software hace lo que el usuario requiere. Ejemplo: Las pruebas deben ser validadas por los programadores, que van a ser los usuarios de las mismas.

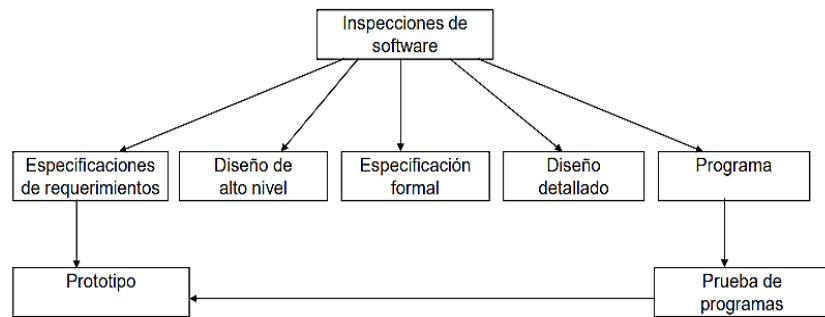
☞ **OBJETIVO DE V & V:** Asegurar que el software sea lo suficientemente bueno para su uso pretendido. No se busca perfección porque todo software va a tener al menos un fallo en su ciclo de vida (generalmente son muchos), por eso existe el mantenimiento.

❖ **NIVEL DE CONFIANZA REQUERIDO:** Los factores que indiquen si es suficientemente bueno van a depender de:

- **Función del software:** Depende de la criticidad del software para una organización.
- **Expectativas del usuario:** Los usuarios están dispuestos a aceptar un cierto número de fallas cuando el beneficio a obtener sea mayor.
- **Entorno de mercado:** Tener en cuenta los programas competidores. Y si los clientes no están dispuestos a pagar un precio alto por el software, entonces pueden estar dispuestos a tolerar más defectos.

❖ **APROXIMACIONES COMPLEMENTARIAS:**

1. Las **inspecciones** del software: Analizan y comprueban las representaciones del sistema tales como la ERS, los diagramas de diseño, el código fuente del programa. Pueden usarse en todas las etapas del proceso. Son técnicas estáticas.
2. Las **pruebas** de software: Implican ejecutar una implementación del software con datos de prueba. Se examinan las salidas y el entorno para comprobar su correcto funcionamiento. Son técnicas dinámicas.

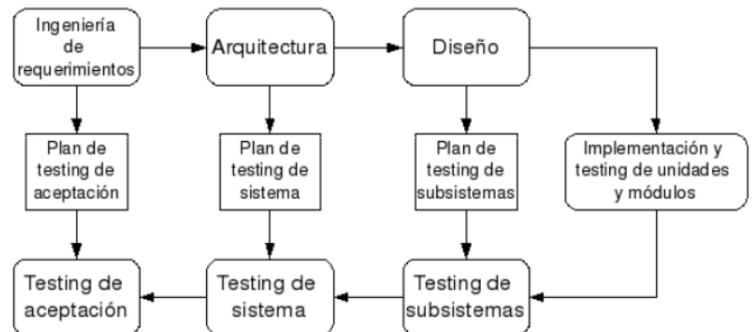


❖ **PLANIFICACIÓN DE V&V:**

Inicia en las primeras etapas del ciclo de desarrollo. Un ejemplo es el **modelo en "V"**, que muestra como los planes de pruebas se derivan de las especificaciones y el diseño. Éste se usa para sistemas bastante críticos, que deberían llegar al usuario con prácticamente CERO errores.

El esfuerzo dedicado a V&V depende del tipo de sistema en desarrollo y de la experiencia organizacional. Las **actividades** de la gestión de V&V comprenden:

- Utiliza estándares y procedimientos para las inspecciones y pruebas del software.
- Define checklists para las inspecciones.
- Define el **plan de pruebas (PP)**.



### CONTENIDO DEL PLAN DE PRUEBAS

- **Descripción del proceso de pruebas:** fases de pruebas a realizar.
- **Requerimientos a probar:** casos de uso, por ejemplo.
- **Productos de software a probar.**
- **Calendarización de las pruebas:** que debe ir integrada con la calendarización de las otras actividades del proceso de desarrollo.
- **Procedimientos de registro de las pruebas:** registración de defectos, asignación, tiempos estimados de corrección, etcétera.
- **Requerimientos de hardware y software.**
- **Restricciones:** restricciones que afectan al proceso de pruebas, por ejemplo, de personal dedicado.



#### ● VENTAJAS DE LAS INSPECCIONES:

- No requieren que el programa se ejecute; incluso pueden realizarse antes que se implemente.
- No reemplazan a las pruebas, sino que son un complemento a utilizar en etapas previas. Sin embargo, son más efectivas y menos costosas que las pruebas:
  - **Inspecciones informales:** descubren 60% de errores.
  - **Inspecciones formales (Cuarto Limpio):** descubren 90% de errores.
  - Se pueden descubrir problemas con algunos requerimientos no funcionales: ajuste a estándares, portabilidad, mantenibilidad.
  - Varios defectos se detectan en una sola sesión de inspección.
  - Reutilizan el conocimiento del dominio y del lenguaje de programación (esto último sí se trata de inspecciones de código).
- Las “**revisiones o inspecciones de pruebas**” pueden descubrir problemas en éstas y ayudar a diseñar formas más efectivas para probar el sistema.
- Las inspecciones siempre requieren sobrecostos al inicio, y sólo conducen a ahorros en los costos después que los equipos de desarrollo adquieran experiencia.

#### ● DESVENTAJAS DE LAS INSPECCIONES:

No pueden validar el comportamiento dinámico del software. A veces no es práctico inspeccionar “todo” un sistema completo, a nivel “sistema” se utilizan las pruebas.

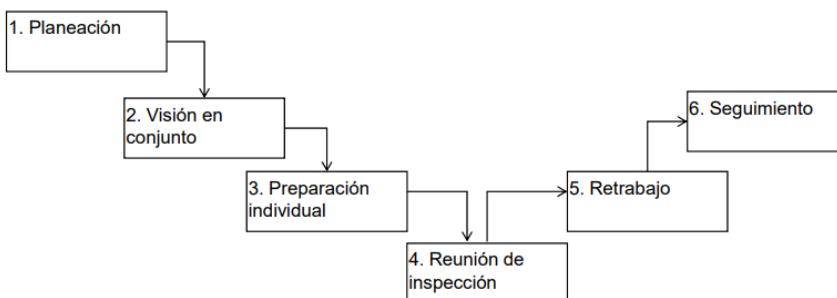
## INSPECCIONES DE PROGRAMA

- Son revisiones cuyo objetivo es la detección de defectos en el programa (código).
- Un equipo con miembros con diferentes conocimientos debe hacer una revisión línea por línea del código fuente de programa. El equipo de personas necesita 4 roles: autor, lector, probador y moderador.
  - Lector lee en voz alta al equipo el código.
  - Probador inspecciona el código desde una perspectiva de prueba.
  - Moderador organiza el proceso.

#### ● PRECONDICIONES AL PROCESO DE INSPECCIÓN:

- Existir especificación precisa del código a inspeccionar.
- Miembros del equipo familiarizados con estándares organizacionales.
- Versión actualizada y sintácticamente correcta del código disponible.

#### ● PROCESO DE INSPECCIÓN:



- No más de 2 horas. Dedicada exclusivamente a detectar defectos, anomalías y discordancias con los estándares. No sugerir formas de corrección o cambios. Checklist de errores comunes (revisado previamente por expertos en el tema).
- Es importante obtener experiencia a nivel organización de cada proceso de inspección (lecciones aprendidas).

- Las inspecciones son un proceso público de detección de errores, a diferencia de los procesos privados de pruebas. Los errores cometidos se muestran a todo el equipo de programación.

Clase de defecto	Comprobación de inspección
Defectos de datos	<ul style="list-style-type: none"> <li>- Se inicializan todas las variables antes de que se utilicen sus valores?</li> <li>- Tienen nombres todas las constantes?</li> <li>- Límites de arreglos/matrizes?</li> <li>- Separadores en cadenas?</li> <li>- Buffers o variables que se desborden?</li> </ul>
Defectos de control	<ul style="list-style-type: none"> <li>-Para cada sentencia condicional, es correcta la condición?</li> <li>- Se garantiza que termina cada bucle?</li> <li>- Están puestas correctamente entre llaves las sentencias compuestas?</li> <li>- En las sentencias case, se tienen en cuenta todos los posibles casos?</li> </ul>
Defectos de entrada / salida	<ul style="list-style-type: none"> <li>-Se utilizan todas las variables de entrada?</li> <li>- Se les asigna un valor a todas las variables de salida?</li> <li>- Pueden provocar errores los datos de entrada no esperados?</li> </ul>
Defectos de interfaz	<ul style="list-style-type: none"> <li>-Concuerdan los tipos de parámetros?</li> <li>- Están en el orden correcto los parámetros?</li> <li>- Las llamadas a funciones y métodos, tienen el número correcto de parámetros?</li> </ul>
Defectos de gestión de almacenamiento	<ul style="list-style-type: none"> <li>- Si se utiliza almacenamiento dinámico, se asigna correctamente el espacio de memoria?</li> <li>- Se limpian todas las variables al salir de la aplicación?</li> </ul>
Defectos de manejo de excepciones	<ul style="list-style-type: none"> <li>- Se tienen en cuenta todas las condiciones de error posibles?</li> </ul>

## ANÁLISIS ESTÁTICO AUTOMATIZADO

Son herramientas que **rastrean el código fuente de un programa y detectan posibles fallas y anomalías**. No requieren que el programa se ejecute. Son un complemento de los recursos de detección de errores provistos por el compilador del lenguaje.

El análisis estático hace **comprobaciones** de flujo de control, utilización de los datos, interfaces, flujo de información y trayectoria (análisis semántico, identifica todas las posibles trayectorias del programa).

➤ Ejemplo de analizador estático:

- **LINT**, bajo Linux, para el lenguaje C.
- Analizador de código **Zend**. Es indispensable para cualquier programación seria en PHP puesto que ayuda a los desarrolladores a analizar el código fuente estático para hacer cumplir las buenas prácticas de codificación y análisis de código PHP.

El analizador de código/análisis semántico logra esta funcionalidad, tratando de conciliar el código problemático y la localización de código inalcanzable (código que se ha definido pero no se usa o con variables de vacío).

\* **QUICK FIX**: La opción de análisis semántico Quick Fix le permite cambiar fácilmente el código problemático de acuerdo a las sugerencias proporcionadas por el mecanismo de análisis semántico.

```

1 <?
2 abstract class A {
3     abstract function foo ();
4 }
5 class B extends A {
6 }
7
  
```

1 method(s) to implement:  
- A.foo()

Add unimplemented methods  
Make type 'B' abstract  
Rename in workspace (Alt+Shift+R direct access)



**MÉTODOS FORMALES:** Se basan en representaciones matemáticas del software (especificación formal). Los métodos formales pueden utilizarse en diferentes etapas de la V&V:

- ❖ Puede desarrollarse una **especificación formal** y luego analizarse matemáticamente en busca de inconsistencias.
- ❖ Puede verificarse que el **código** de un software es consistente con su especificación. Esto requiere una especificación formal a la que luego se le aplica un proceso de desarrollo transformacional (Sala Limpia):

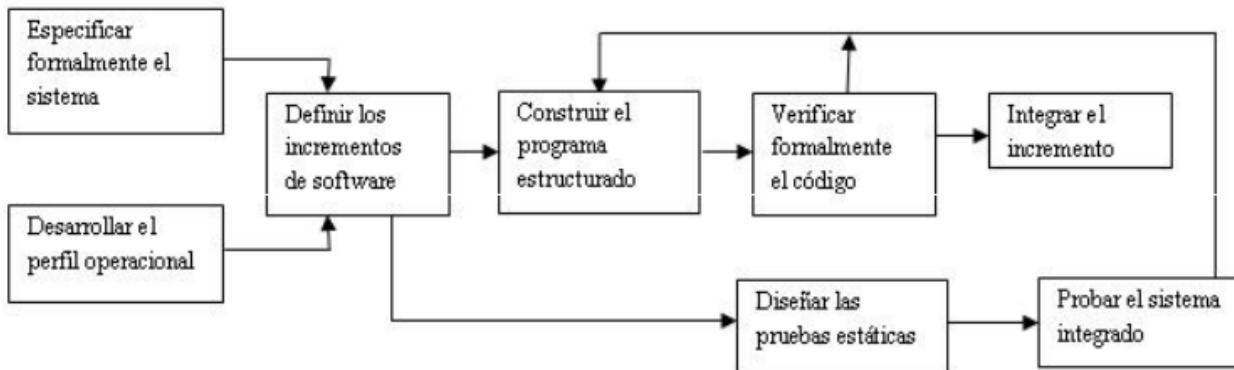
### PROCESO DE DESARROLLO SALA LIMPIA



Es un proceso de desarrollo que utiliza métodos formales para soportar inspecciones de software rigurosas. El objetivo de este desarrollo de software es obtener software con cero defectos.

La aproximación de Sala Limpia se basa en cinco estrategias:

1. **ESPECIFICACIÓN FORMAL:** El software que se va a desarrollar se especifica formalmente.
2. **DESARROLLO INCREMENTAL:** El software se divide en incrementos que se desarrollan y se validan utilizando el proceso de Sala Limpia
3. **PROGRAMACIÓN ESTRUCTURADA:** El proceso de desarrollo del programa es un proceso de pasos de la especificación. El objetivo es transformar la especificación para la creación del código del programa.
4. **VERIFICACIÓN ESTÁTICA:** El software se verifica estáticamente usando inspecciones de software rigurosas.
5. **PRUEBAS ESTADÍSTICAS DEL SISTEMA:** El incremento del software integrado es probado estadísticamente para determinar su fiabilidad.



# Pruebas:

El **testing** es una etapa que comienza al terminar de codificar y consiste en probar que el software funciona. Testing es igual a **calidad de producto** y de **proceso**, por lo cual el tester es el enemigo del programador (evidencia los errores y critica).

- El testing todavía no es parte fundamental de la formación profesional.
- El desarrollo de software es caro.
- Más del 50% del esfuerzo de desarrollo frecuentemente lo ocupa la tarea de testing.
- El proceso de testing todavía es muy inmaduro.

Un software de calidad debe estar bien diseñado, bien elaborado, bien documentado y **bien probado**. Llamamos **probar** al *proceso de detectar diferencias (errores) entre el comportamiento esperado y el comportamiento actual del sistema*.

Otras definiciones de probar entablan:

- Establecer fehacientemente que un programa hace lo que se supone que tiene que hacer (Bill Hetzel, 1975).
- El proceso de ejecutar un programa o sistema con el objeto de encontrar errores (Myers, 1979).
- Una medida de la calidad del software.
- El proceso de ejecutar un programa utilizando un caso de prueba construido expresamente para asegurar que la conducta exhibida es la esperada.
- “El testeo de software es comparable, en su esencia, al trabajo del Editor para un Escritor de libros, pues quien desarrolla está inmerso en el proyecto y necesita de una vista objetiva para perfeccionar su producto.” Michael Albers.
- “Testing es una disciplina profesional que requiere personas capacitadas y entrenadas.” Edward Kit.

## ❖ ¿DE QUÉ SIRVE PROBAR?

Es una de las barreras para evitar que sistemas de baja calidad lleguen a los usuarios finales.

## ❖ ¿CUÁL ES EL OBJETIVO DE PROBAR?

- Encontrar errores.
- Detectar los errores del sistema antes que lo haga el usuario final.
- Asegurar la implementación de toda su funcionalidad.
- Prevenir potenciales defectos futuros (proceso de mejora).
- Nos lleva a asegurar la satisfacción del cliente.

## ❖ ¿QUÉ PASA SI NO PROBAMOS?

Pérdidas de tiempo, económicas e intangibles (confiabilidad, imagen, etcétera).

Consorcio de Informática Sostenible explicó que podían existir tanto como 20 o 30 fallos o bugs por cada 1000 líneas de código en la mayoría de las aplicaciones de software. El Consorcio Cutter comentó que, de las firmas de software que encuestaron, el 32% admitió haber lanzado software con una gran cantidad de defectos; el 38% dijo que su software carecía de un programa de garantía adecuado, y el 27% comentó que no realizaban revisiones formales de calidad de su software.

## ❖ ¿CUÁLES SON LAS DESVENTAJAS DE PROBAR?

Requieren que el software se desarrolle (prototipo o producto final) y debe ejecutarse reiteradas veces. Por lo general, se encuentra una falla por cada prueba.

## ❖ ¿PARA QUÉ INVOLUCRAR EL TESTING DE MANERA TEMPRANA?

Dar visibilidad de manera temprana al equipo de cómo se va a probar el producto. Disminuir los costos de correcciones de defectos.

## PROCESO GENERAL DE PRUEBAS:

Inicia con las pruebas de las unidades de programas individuales como funciones u objetos. Despues éstas se integran en subsistemas y sistemas y se prueban las interacciones de estas unidades. Finalmente, despues de completar el desarrollo del software, los clientes llevan a cabo pruebas de aceptación para verificar que el sistema se desempeñe conforme a lo especificado.

- **CICLO 0:** Prueba individual de cada componente a cargo del desarrollador.
- **CICLO 1 Y SUCESIVOS:** Prueba individual de cada componente a cargo del equipo de testing.
- **PRUEBAS DE INTEGRACIÓN:** Siempre a cargo de equipo de testing, una vez fallidos todos los casos de pruebas individuales (o unitarios).

Sommerville sostiene que el límite entre pruebas unitarias y de integración es difuso, ya que los sistemas orientados a objetos difieren de los orientados a funciones por dos razones:

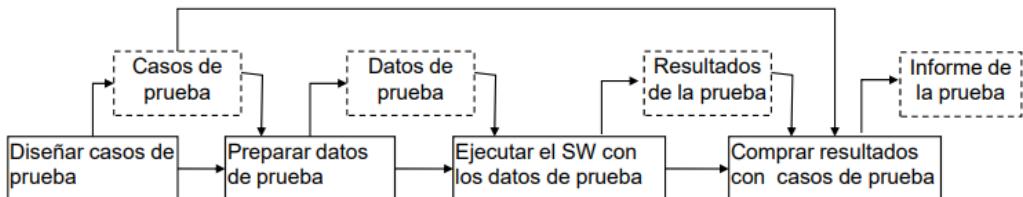
1. En sistemas OO no existe una clara distinción entre las unidades básicas del programa (funciones) y las colecciones de estas unidades (módulos).
2. No existe clara jerarquía de objetos.

Sin embargo, se pueden definir unidades de partición a utilizar en todas las etapas de desarrollo, incluyendo el testing, como son los "casos de uso", y definir pruebas unitarias por CU.

Las pruebas exhaustivas, donde se prueba cada posible secuencia de ejecuciones del programa, no son prácticas. Se debe llevar a cabo pruebas sobre un subconjunto de posibles casos de prueba. (Definir políticas organizacionales para pruebas, tener en cuenta experiencia en la utilización del sistema).

## PRUEBAS DE DEFECTOS

- ☞ **OBJETIVO:** Exponer los defectos latentes en un sistema de software antes de entregar el sistema.
- ☞ Lo anterior va en contraposición con las **pruebas de aceptación**, en las que se pretende demostrar que un sistema cumple con su especificación.
- ☞ Prueba de defecto exitosa: expone un defecto = el sistema se desempeña incorrectamente.



## PRUEBAS DE CAJA NEGRA O FUNCIONALES

Se derivan de la especificación de requerimientos. El comportamiento del sistema está determinado por sus entradas y las salidas esperadas.

Entre estas se tiene el **monkey testing**, técnica en la que se definen diferentes hitos a ser probados, que pueden ir desde elementos de carácter unitario hasta elementos cuya funcionalidad es proporcionada por varias capas de la arquitectura de la aplicación (por ejemplo, los campos de un formulario), a los que se les proporciona un juego de entrada de datos aleatorio que puede tener acotado o no el patrón de generación de los mismos.

## PRUEBAS DE CAJA BLANCA O ESTRUCTURALES

Son las opuestas a las anteriores. Se conoce la estructura del software a probar. Se utilizan principalmente para probar ciertas porciones de código.

## PRUEBAS DE INTEGRACIÓN

☞ **PRUEBAS DEL SISTEMA:** Implican integrar varios componentes y probar el sistema integrado. Existen dos fases de pruebas de sistema.

1. **Pruebas de integración:** El equipo de testing tiene acceso al código del sistema. Cuando descubre un problema, se intenta localizar el componente que lo origina y que debe ser depurado. El objetivo es encontrar defectos en el sistema.
2. **Pruebas de entrega:** Se prueba la versión al entregar al cliente, sin disponer del código. Son pruebas de “caja negra”, en las que el equipo de testing se encarga de demostrar si el sistema funciona correctamente o no.
  - Cuando participa el cliente en estas pruebas, se denominan **Pruebas de Aceptación**. Si la entrega es lo suficientemente buena, el cliente la acepta para su uso.

Los **componentes** a integrar pueden ser de **distintos tipos**:

- Componentes comerciales.
- Componentes reutilizados que han sido adaptados.
- Componentes nuevos.

La integración implica identificar grupos de componentes que proporcionan alguna funcionalidad del sistema e integrarlos añadiendo código para hacer que funcionen conjuntamente.

Existen dos **estrategias de integración**: descendente y ascendente (todo separado, las unidades más chiquitas y lo empiezo a unir hacia arriba). En la práctica, suele utilizarse una mezcla de ambas.

## AUTOMATIZACIÓN DE LAS PRUEBAS



Herramientas de automatización de pruebas ofrecen una serie de facilidades y su uso puede reducir significativamente el costo de las pruebas.

- **FRAMEWORKS DE PRUEBAS:** (marcos de trabajo) Es un conjunto integrado de herramientas para soportar el proceso de pruebas.
- **JUnit:** Es un conjunto de clases que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, Junit devolverá un fallo en el método correspondiente.

JUnit es también un medio de controlar las **pruebas de regresión**, necesarias cuando parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.

# mantenimiento

El mantenimiento implica administrar el proceso de cambios en el sistema y es la parte más costosa del ciclo de vida del software (60-90% del coste total y creciente a medida que se está manteniendo – afectado por factores técnicos y no-técnicos; 2 a 100 veces mayores que los costos del desarrollo). Se define, formalmente, como: “*la modificación de un producto software después de su entrega al cliente o usuario para corregir defectos, para mejorar el rendimiento u otras propiedades deseables, o para adaptarlo a un cambio de entorno.*” (IEEE 83).

Es un proceso inevitable ya que los requerimientos del sistema a menudo cambian cuando el sistema está siendo desarrollado, debido a que los sistemas se encuentran fuertemente acoplados con el **ambiente**. Cuando un sistema es instalado en un ambiente que cambia, el ambiente produce cambios en los requerimientos. **DEBEN** ser mantenidos si se quiere que sean útiles en el ambiente.

## • DESVENTAJAS DEL MANTENIMIENTO:

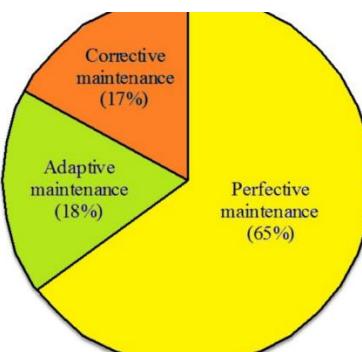
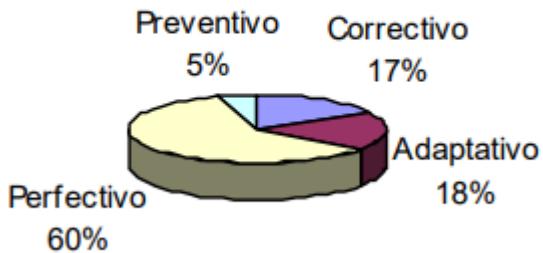
- Oportunidades de desarrollo que se pierden y perjuicio en otros proyectos de desarrollo cuando el equipo tiene que dejarlos, total o parcialmente, para atender peticiones de mantenimientos.
- Insatisfacción del cliente cuando no se puede atender en un tiempo aceptable una petición de reparación que parece razonable.
- Los errores ocultos que se introducen al cambiar el software durante el mantenimiento reducen la calidad global del producto. Intentando arreglar se puede producir otro error, no siempre es perfecto el proceso.

## • TIPOS DE MANTENIMIENTO:

- **Correctivo:** Cambiar al sistema de forma que corrija deficiencias y así cumpla con sus requerimientos. Suele ser el más común.
- **Adaptativo:** Cambiar al sistema para que cumpla con nuevos requerimientos.
- **Perfectivo:** Cambios al sistema que permiten que los requerimientos sean más efectivos. Implica el mantenimiento de ampliación y el mantenimiento de eficiencia.
- **Preventivo:** Consiste en la modificación del software para mejorar las propiedades de dicho software (por ejemplo: aumentando su calidad o mantenibilidad) sin alterar sus especificaciones funcionales:
  - incluir sentencias que validen los datos de entrada,
  - reestructurar los programas para mejorar su legibilidad,
  - incluir nuevos comentarios.

Dentro del software preventivo encontramos el mantenimiento para la **reutilización**, el cual trata de mejorar la propiedad de reusabilidad del software.

## Coste de mantenimiento



El mantenimiento preventivo es opcional. Algunos autores no lo mencionan, pero es importante tenerlo en cuenta.

Usualmente es más caro añadir funcionalidad después de que el sistema ha sido desarrollado, en vez de cuando se está diseñando.

Categoría	Actividad	% Tiempo
Comprensión del sw. y de los cambios a realizar	Estudiar las peticiones	18%
	Estudiar la documentación	6%
	Estudiar el código	23%
Modificación del sw.	Modificar el código	19%
	Actualizar la documentación	6%
Realización de pruebas	Diseñar y realizar pruebas	28%

Nótese cómo la comprensión del software y de los cambios supone casi un 50% del coste total de mantenimiento.

#### • DIFICULTADES DEL MANTENIMIENTO:

- Aplicaciones antiguas heredadas (legacy code):
  - o Restricciones de tamaño y espacio de almacenamiento.
  - o Herramientas desfasadas, sin métodos.
  - o Una o varias migraciones a nuevas plataformas.
  - o Múltiples modificaciones para adaptarlos o mejorarllos.
  - o Desarrolladores no localizables.

¿Desechar el software y reescribirlo? No factible al tener una gran carga financiera de su desarrollo y la necesidad de amortización (se obtiene un software que sigue funcionando con baja calidad).

- Ausencia de métodos (se realiza de forma ad hoc).
- Cambio tras cambio, los programas tienden a ser menos estructurados, tener estructura pobre.
- Ausencia de documentación.
- Falta de documentación. No hay captura adecuada de requisitos (mayores esfuerzos de mantenimiento futuros) o no existen registros de pruebas (imposibilidad de pruebas de regresión).
- Problemas de gestión, considerado “trabajo poco creativo” entre el personal de desarrollo y asignado a las personas con menos experiencia que no saben cómo realizar el mantenimiento adecuadamente.
- Una administración de configuraciones inadecuada, a menudo implica que en un momento dado no se conozca cuál sea la versión del sistema más reciente o más funcional.

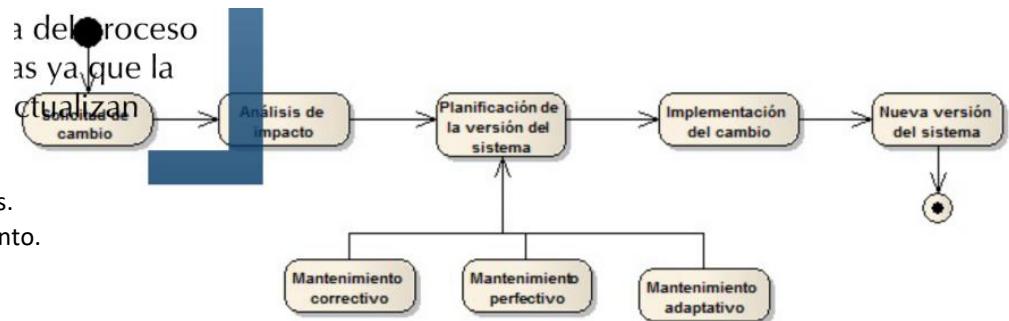


#### • MOTIVACIÓN PARA EL MANTENIMIENTO:

- Relacionar el desarrollo del software a las metas organizacionales-racionalidad del mantenimiento.
- Relacionar compensaciones al desempeño organizacional.
- Integrar el mantenimiento con el desarrollo.
- Crear un presupuesto para un mantenimiento preventivo.
- Elaborar planes para el mantenimiento en fases iniciales del proceso de desarrollo.
- Realizar esfuerzos y planes para llevar a cabo una mantenibilidad de programas.

### • PROCESO DEL MANTENIMIENTO:

1. Implementación del proceso.
2. Análisis del problema y modificaciones.
3. Revisión y aceptación del mantenimiento.
4. Migración.
5. Retirada del software.



El mantenimiento se debe a cambios pedidos por los clientes o por los requerimientos del mercado. Los cambios normalmente se atienden en orden de pedido y se implementan en una nueva versión del sistema.

Los programas algunas veces necesitan ser reparados sin que se realice una iteración completa del proceso (ciclo de vida), lo cual provoca problemas ya que la documentación y los programas se desactualizan.

### • MÉTODOS DE MANTENIMIENTO:

- **Reingeniería:** examen y modificación del sistema para reconstruirlo en una nueva forma.
- **Ingeniería inversa:** análisis de un sistema para identificar sus componentes y las relaciones entre ellos, así como para representaciones del sistema en otra forma o en un nivel de abstracción más elevado.
- **Reestructuración del software:** consiste en la modificación del software para hacerlo más fácil de entender y cambiar o hacerlo menos susceptible de incluir errores en cambios posteriores.
- **Transformación de programas:** técnica formal de transformación de programas.



### • DOCUMENTACIÓN DEL MANTENIMIENTO:

- Documento de requerimientos.
- Descripción de la arquitectura del sistema.
- Documentación del diseño de programas.
- Listado de códigos fuente.
- Planes de prueba y reportes de validación.
- Guía de mantenimiento del sistema.

**MÉTRICAS:** Son mediciones de las características de los programas que podrían permitir una predicción de la mantenibilidad.

- Complejidad de los datos.
- Acoplamiento.
- Documentación del código.
- Número de peticiones de acciones correctivas.
- Tiempo promedio requerido para el análisis de impacto.
- Tiempo promedio que se toma implementar una petición de cambios.

### CONCLUSIONES:

1. Los tres tipos de mantenimiento son perfecto, adaptivo y correctivo.
2. Los costos de mantenimiento usualmente exceden los costos de desarrollo para sistemas grandes de gran duración.
3. Los esfuerzos dedicados a mejorar la mantenibilidad pueden ser benéficos en costo/beneficio a largo plazo.
4. La documentación debe incluir documentos de requerimientos, diseño y validación.