

## Tema I - Introducción

Justificación de la ingeniería del software .....	1
Definición de la ingeniería del Software .....	1
¿Por qué necesitamos la Ing. Software? .....	1
Definiciones .....	2
¿Qué es el Software? .....	2
¿Qué es un Proceso Software? .....	2
¿Qué es un Modelo de Proceso Software? .....	3
Costes del Software .....	3
¿Qué son las herramientas CASE? .....	4
Características del Software .....	4
Diversidad del Software .....	4
Mitos del Software .....	4
Mitos de gestión de proyectos .....	4
Mitos del cliente .....	4
Mitos de los profesionales del software .....	5
Aspectos éticos del Software .....	5

---

# INTRODUCCIÓN

---

## Justificación de la ingeniería del software

### Definición de la ingeniería del Software

#### *Definición I*

*La aplicación de un enfoque sistemático, disciplinado y cuantificable para el desarrollo, operación y mantenimiento del software; es decir, la aplicación de la ingeniería al software.*

#### *Definición II*

*Disciplina de ingeniería que se ocupa de todos los aspectos de la producción de software desde las etapas iniciales de la especificación del sistema hasta el mantenimiento del sistema después de su puesta en funcionamiento.*

### ¿Por qué necesitamos la Ing. Software?

Porque el software es complejo y, por lo tanto, necesitamos equipos de personal cualificado y una disciplina de ingeniería. La ingeniería del software es necesaria de análisis, especificación, planificación, diseño, codificación, pruebas y mantenimiento.

# Definiciones

## ¿Qué es el Software?

El software es un conjunto de programas de ordenador, documentación asociada y datos. Una disciplina de ingeniería que trata con todos los aspectos de la producción de software de calidad.

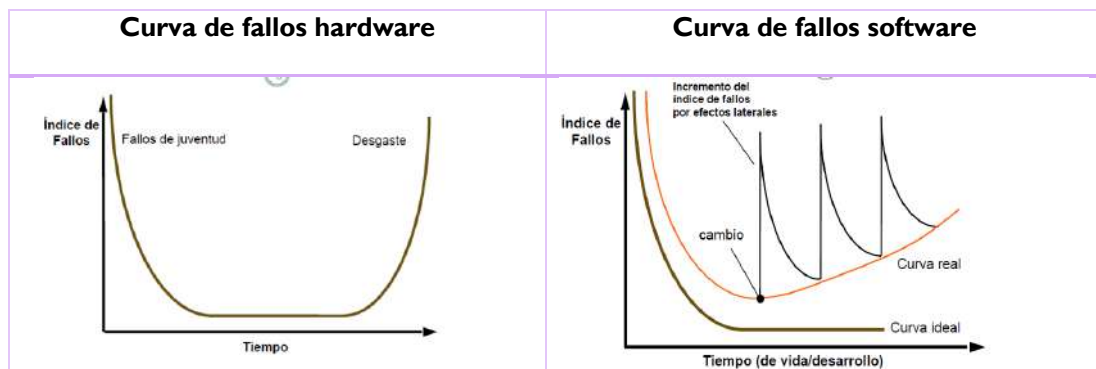
Es el establecimiento y uso de principios de ingeniería sólidos para obtener un software económico que sea confiable y funcione eficientemente en máquinas reales.

Los ingenieros software deben adoptar una forma de trabajo sistemática y organizada y utilizar herramientas y técnicas dependiendo del problema a resolver, las restricciones de desarrollo y los recursos disponibles.

Puede ser desarrollado para un cliente particular (de acuerdo con sus especificaciones) o un mercado general. Además, un software nuevo se puede crear:

- Desarrollando nuevos programas
- Reconfigurando sistemas software genéricos
- Reutilizando software existente

El software se desarrolla, no se fabrica, y además el software no se puede desgastar ni estropear por uso.



## ¿Qué es un Proceso Software?

Conjunto de actividades cuyo objetivo es el desarrollo o la evolución de un sistema software.

Actividades genéricas de un proceso Software:

- **Comunicación:** para saber qué quiere el cliente
- **Planificación:** descripción de las tareas técnicas a llevar a cabo
- **Modelado:** creación de modelos para entender los requisitos software y el diseño que los llevará a cabo
- **Construcción:** generación de código (programas) y su testeo para eliminar errores (software testing)
- **Implantación:** se entrega al cliente para su puesta en marcha

Formación  
Online  
Especializada

Clases Online  
Prácticas  
Becas

Escuela de  
LÍDERES

Jose María Girela  
Bim Manager.

## ¿Qué es un Modelo de Proceso Software?

Es una representación simplificada de un proceso de software, presentado desde una perspectiva específica. Algunos ejemplos de perspectiva son:

- Perspectiva del flujo de trabajo – *la secuencia de actividades*
- Perspectiva del flujo de datos – *flujo de información*
- Perspectiva de rol/acción – *quién hace qué*

Ejemplos de modelos de proceso software:

- Modelo en cascada
- Desarrollo iterativo
- Desarrollo software basado en componentes
- Métodos ágiles
- Proceso unificado

## Costes del Software

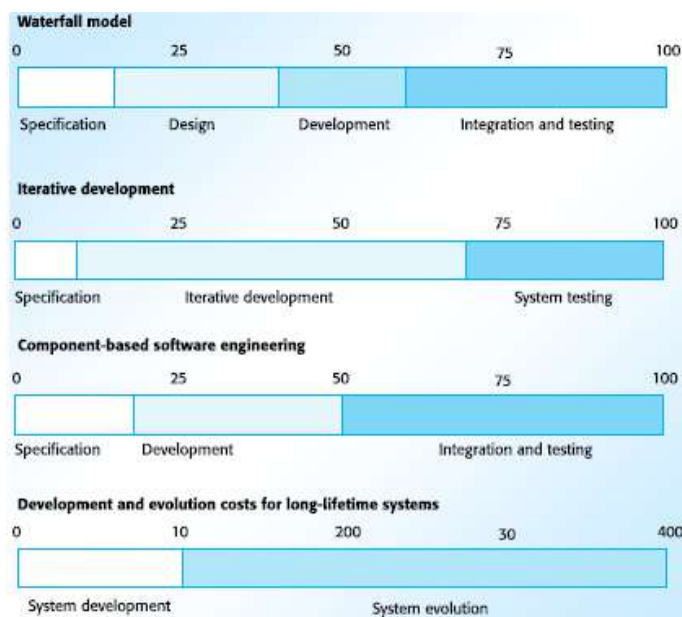
El coste del software frecuentemente predomina en coste global de un sistema informático. De hecho, el coste del mantenimiento del software es mayor que el de la etapa de desarrollo (aproximadamente el 60% de los costes son costes de desarrollo, el 40% son costes de pruebas).

Para software personalizado, los costes de evolución suelen superar a los costes de desarrollo.

Los costes varían dependiendo del tipo de sistema a desarrollar y de los requisitos para ciertos atributos del sistema como el rendimiento o la fiabilidad. Además, su distribución depende del modelo de desarrollo que utilice.

Es necesario producir software con un coste adecuado.

**Distribución de costes por actividad:**



## ¿Qué son las herramientas CASE?

Son herramientas software destinadas a proporcionar soporte para las actividades de un proceso de software. Dos tipos:

- **Upper-CASE:** Herramientas de soporte de las actividades iniciales del proceso (requisitos, modelado, diseño). *Ejemplo: MagicDraw*
- **Lower-CASE:** Herramientas de soporte de actividades posteriores (programación, depuración, pruebas). *Ejemplo: IDEs*

## Características del Software

Un buen software debe proporcionar la funcionalidad requerida por el cliente y además tener las siguientes características:

- **Mantenibilidad:** Para poder evolucionar para satisfacer nuevas necesidades.
- **Fiabilidad:** Debe estar libre de errores.
- **Eficiencia:** Los recursos no han de ser malgastados.
- **Aceptación:** Por los usuarios finales (no por los desarrolladores).

## Diversidad del Software

- Software de sistemas (p.ej. compiladores, SOs, drivers, etc)
- Software de línea de productos (p.ej. paquetes ofimáticos)
- Sistemas empotrados
- Software de ingeniería y ciencias
- Software de inteligencia artificial
- Muchos más.

## Mitos del Software

### Mitos de gestión de proyectos

*“Si vamos mal de tiempo, podemos añadir más programadores para avanzar más rápido” (horda de mongoles)*

*“Si decidimos subcontratar el proyecto a un tercero podemos confiar en que lo harán bien”*

*“Para saber lo que hay que hacer podemos confiar en los libros que contienen estándares y procedimientos para construir software”*

### Mitos del cliente

*“Una simple especificación de objetivos es suficiente para empezar a trabajar –los detalles se incluirán con posterioridad”*

*“Los requisitos de un sistema software cambian continuamente, pero es fácil hacer cambios en el software porque éste es flexible”*

## Mitos de los profesionales del software

*“Una vez que el programa compila y funciona, ya hemos terminado nuestro trabajo”*

*“Hasta que no se tenga el programa en funcionamiento no se puede determinar su calidad”*

*“El único resultado válido de un producto software es el programa resultante”*

*“Los ingenieros software tienden a escribir grandes cantidades de documentación que no sirve para nada, excepto para ralentizar la marcha del proyecto”*

## Aspectos éticos del Software

Aspectos éticos a considerar:

- Confidencialidad
- Competencia
- Derechos de propiedad intelectual
- Mal uso de los ordenadores

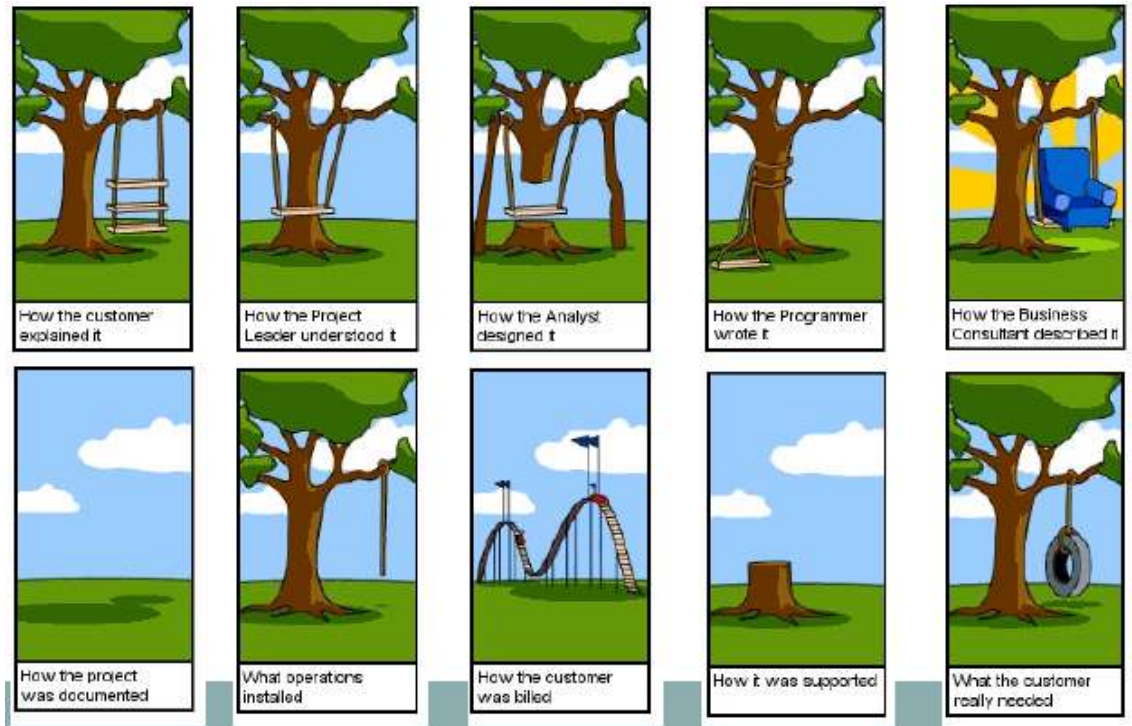
## Tema 2 – Procesos Software

Introducción.....	2
La paradoja del columpio.....	2
Procesos del software.....	3
Actividades dentro de un proceso software .....	4
Actividades genéricas de un proceso software.....	4
Modelos de proceso software .....	4
Tipos de modelos software.....	5
Modelos clásicos .....	5
Modelo en cascada .....	5
Modelos de prototipado.....	5
Desarrollo en espiral .....	6
Desarrollo incremental .....	7
Modelos especializados .....	8
Desarrollo basado en componentes .....	8
Métodos formales.....	8
Modelos ágiles .....	8
Programación extrema (XP).....	9
Scrum .....	10

# PROCESOS SOFTWARE

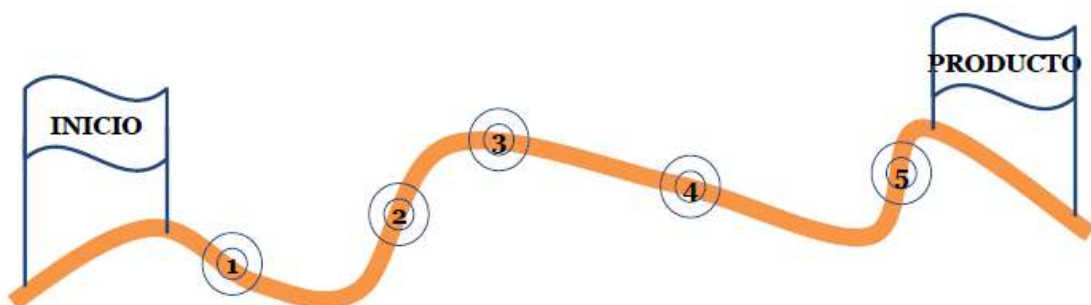
## Introducción

### La paradoja del columpio



Los procesos software constituyen un conjunto de actividades cuya meta es el desarrollo del software (producto).

*El proceso del software es como un “mapa de carreteras” o una “hoja de ruta” que marca los pasos que deben seguirse para la obtención del software.*





Existen diferentes procesos de software. No obstante, algunas actividades básicas son comunes a todos ellos.

Para afrontar la **resolución de cualquier problema** hay que plantearse:

- Entender el problema.
- Pensar una solución.
- Llevar a cabo un plan para aplicar la solución.
- Examinar el resultado y comprobar que el problema se ha resuelto de forma satisfactoria.

## Procesos del software

**Entender el problema** es tarea de **INGENIERÍA DE REQUISITOS**.

**Pensar en una solución** es tarea de **DISEÑO SOFTWARE**.

**Llevar a cabo un plan** es tarea de la **IMPLEMENTACIÓN DEL SOFTWARE**.

**Comprobar que se ha resuelto el problema** es tarea de las fases de **VALIDACIÓN Y EVOLUCIÓN**.

INGENIERÍA DE REQUISITOS	DISEÑO SOFTWARE
<ul style="list-style-type: none"><li>• Estudio de viabilidad.</li><li>• Obtención y análisis de requisitos.</li><li>• Especificación (modelado del análisis).</li><li>• Validación de requisitos.</li></ul>	<ul style="list-style-type: none"><li>• La estructura del SW que se va a implementar</li><li>• Los datos que forman parte del sistema</li><li>• Las interfaces entre los componentes del sistema</li><li>• Los algoritmos utilizados</li></ul>

IMPLEMENTACIÓN DEL SOFTWARE	VALIDACIÓN Y EVOLUCIÓN
<ul style="list-style-type: none"><li>• Se debe planificar:<ul style="list-style-type: none"><li>○ Tareas.</li><li>○ Recursos.</li><li>○ Entregables (deliverables).</li><li>○ Calidad del software.</li></ul></li><li>• Objetivo: se debe producir software que cumpla su especificación.</li></ul>	<ul style="list-style-type: none"><li>• El software debe funcionar y hacer lo que el cliente desea</li><li>• El software debe evolucionar para satisfacer:<ul style="list-style-type: none"><li>○ Necesidades cambiantes del cliente</li><li>○ Cambios en el entorno<ul style="list-style-type: none"><li>▪ Técnicos</li><li>▪ Sociales</li><li>▪ Legales</li><li>▪ Etc.</li></ul></li></ul></li></ul>



## Gana dinerito extra.

Recomienda a tus negocios favoritos que se anuncien en Wuolah y llévate **50€**.

Te daremos un código promocional para que puedan anunciarse desde 99€.

1 Ve a tu negocio favorito • 2 Dale tu código de promo • 3 Diles que nos llamen o nos escriban.



## Actividades dentro de un proceso software

### Actividades genéricas de un proceso software

- **Comunicación:** para saber qué quiere el cliente.
- **Planificación:** descripción de las tareas técnicas a llevar a cabo.
- **Modelado:** creación de modelos para entender los requisitos software y el diseño que los llevará a cabo.
- **Construcción:** generación de código (programas) y su testeo para eliminar errores (software testing).
- **Implantación:** entrega al cliente para su puesta en marcha.
- **Especificación:** Qué debe hacer el sistema y las restricciones a considerar.
- **Desarrollo:** Producir el sistema software.
- **Validación:** Verificar que el software es lo que quiere el cliente.
- **Evolución:** Modificar el software en respuesta a necesidades cambiantes.
- **Gestión del proyecto.**
- **Gestión de riesgos.**
- **Control de calidad.**
- **Revisiones técnicas.**
- **Gestión de configuraciones.**

## Modelos de proceso software

También son llamados modelos de ciclo vida del software. Son una representación abstracta (simplificada) de un proceso software, propuestos inicialmente para **poner orden al caos del desarrollo software**.

Debido a la amplia diversidad de tipos de aplicaciones software:

- No existe un proceso de software ideal.
- Hay modelos distintos que se adaptan al tipo de software.
- Si el modelo no es adecuado se presentan dificultades (burocracia innecesaria, problemas durante el desarrollo, etc.).

Existen varios modelos genéricos los cuales no son descripciones detalladas al 100%, sino abstracciones que se pueden usar para explicar diferentes enfoques de desarrollo. Proponen una guía útil.

Cuando se decide usar uno de estos modelos hay que detallar las tareas a realizar para alcanzar las metas de desarrollo y adaptar el modelo de proceso resultante y ajustarlo a la naturaleza específica del proyecto en cuestión.



**653  
811  
910**

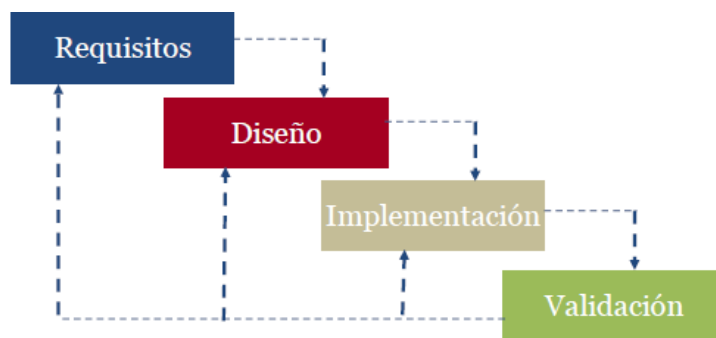
# Tipos de modelos software

Modelos clásicos	Modelos ágiles	Modelos especializados
<ul style="list-style-type: none"><li>▪ Modelo en cascada.</li><li>▪ Modelos de prototipado.</li><li>▪ Desarrollo en espiral.</li><li>▪ Desarrollo incremental.</li></ul>	<ul style="list-style-type: none"><li>▪ Programación extrema (XP)</li><li>▪ Scrum</li></ul>	<ul style="list-style-type: none"><li>▪ Basado en componentes</li><li>▪ Métodos formales</li></ul>

## Modelos clásicos

### Modelo en cascada

Modelo secuencial que se aplica partiendo de la especificación hasta su finalización. Es un ciclo de vida clásico y las actividades fundamentales se suceden a lo largo del desarrollo.



- **Ventajas**
  - Adecuado cuando los requisitos están totalmente claros al inicio.
- **Inconvenientes**
  - Los proyectos reales raramente siguen el flujo secuencial del modelo
    - El cliente a veces no conoce todos los requisitos al inicio
    - Miembros del equipo deben esperar a que terminen tareas precedentes
  - El cliente debe tener paciencia
  - Poco flexible

### Modelos de prototipado

**Prototipo:** Versión inicial del software que se utiliza para informarse más del problema y sus posibles soluciones.

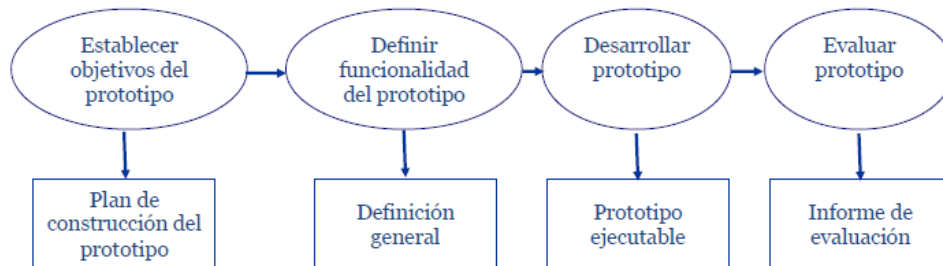
Es un diseño rápido centrado en los aspectos visibles del cliente.



Un prototipo se puede usar de varias maneras en un proceso de desarrollo de software:

- Un prototipo está disponible de forma rápida y se puede enseñar al cliente
- Obtención y validación de requisitos
- Explorar soluciones de diseño
- Ejecutar pruebas

Fases del desarrollo



### Inconvenientes

- El prototipo no tiene por qué usarse como el sistema final.
- Puede ser imposible hacer que el prototipo cumpla requisitos no funcionales (rendimiento, seguridad, fiabilidad, etc.).
- Los estándares de calidad se suelen relajar en un prototipo.
- Los prototipos se suelen hacer rápido
  - Poca documentación.
  - Puede dificultar el mantenimiento a largo plazo.
- Impaciencia de los clientes.

## Desarrollo en espiral

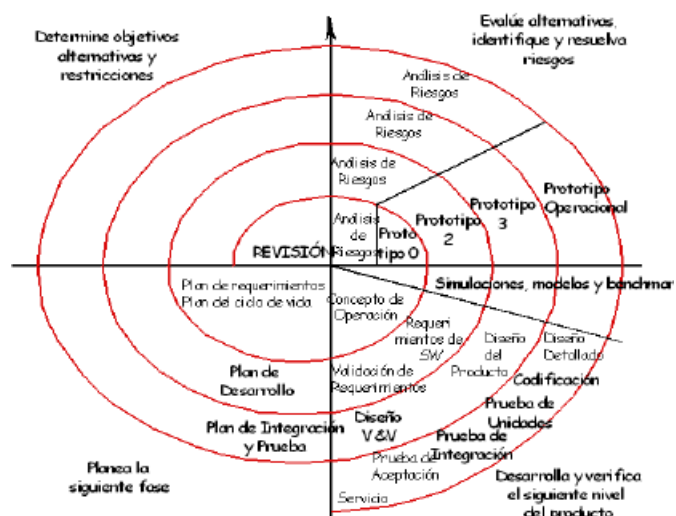
Es un proceso de tipo evolutivo. Se presenta como una espiral y no como una secuencia de actividades.

Cada ciclo de la espiral representa una fase del proceso.

Está orientado a la prevención de riesgos, los cuales se evalúan de forma explícita a lo largo del proceso.

Cada ciclo se divide en cuatro sectores:

- Determinación de objetivos
- Evaluación y reducción de riesgos
- Desarrollo y validación
- Planificación



### Ventajas

- Modelo realista para el desarrollo de software a gran escala
  - El software evoluciona a la vez que lo hace el modelo
- El uso de prototipos
  - Permite conocer mejor el riesgo
  - El cliente puede observar cómo va el producto

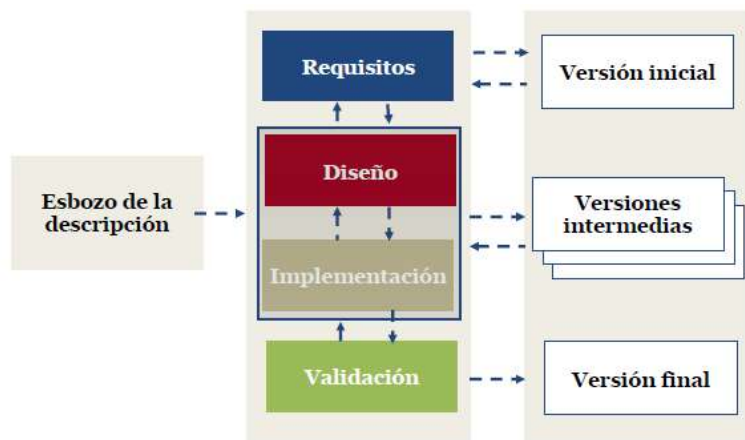
### Inconvenientes

- Requiere expertos en gestión de riesgos

## Desarrollo incremental

La idea es desarrollar una implementación inicial, presentársela al usuario e ir refinándola a través de diferentes versiones.

Las actividades del proceso se mezclan, en vez de separarse como se hace en el modelo en cascada. En ocasiones, la especificación se desarrolla junto con el software.



**Objetivo:** desarrollar software de alta calidad de una manera iterativa o incremental.

Los procesos ágiles y el desarrollo en espiral se ajustan a esta categoría.

### Ventajas:

- La especificación se puede desarrollar de forma creciente, ya que el cliente puede evaluar sobre la marcha cómo va el proyecto
- Puede satisfacer necesidades inmediatas de los clientes

### Inconvenientes:

- Es difícil establecer a priori el número de ciclos que serán necesarios para construir el producto.
- Es difícil establecer una arquitectura estable del sistema
  - Inapropiado para sistemas grandes y complejos, con un periodo de vida largo en el que varios equipos desarrollan distintas partes del sistema.
- Si los sistemas se desarrollan rápidamente, no es rentable producir documentos que reflejen cada cambio del sistema
- Los cambios continuos tienden a corromper la estructura del software: refactorizar



## Gana dinerito extra.

Recomienda a tus negocios favoritos que se anuncien en Wuolah y llévate **50€**.

Te daremos un código promocional para que puedan anunciarse desde 99€.

- 1 Ve a tu negocio favorito
- 2 Dale tu código de promo
- 3 Diles que nos llamen o nos escriban.



### Modelos especializados

Son enfoques concretos de ingeniería del software que se aplican en algunos contextos.

#### Desarrollo basado en componentes

- Enfoque basado en la reutilización
- Un programa se construye integrando componentes independientes débilmente acoplados
- Símil: electrónica (componentes estándar)

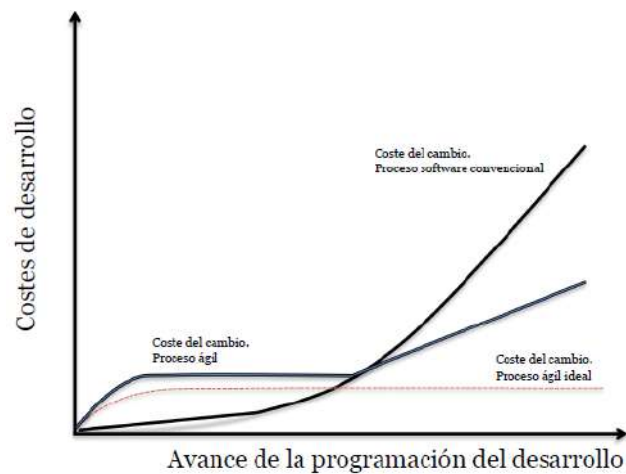
#### Métodos formales

- Especificación matemática del software
  - Álgebra, Cálculo, Lógica, Autómatas, etc.
- A través del análisis matemático se pueden detectar errores e inconsistencias
- Usado generalmente para sistemas críticos

### Modelos ágiles

Los métodos ágiles combinan:

- Filosofía
  - Fomento de la satisfacción del cliente
  - Desarrollo incremental
  - Equipos de pequeño tamaño y muy motivados
  - Métodos informales, menor documentación
  - Simplicidad y flexibilidad
- Guías de desarrollo
  - Comunicación continua y activa con el cliente (el cliente es parte del equipo)
  - Fomento de la entrega de partes del producto frente a análisis y diseño



- Ideas sobre las que sustentan los métodos ágiles:



**653  
811  
910**

- Es difícil predecir inicialmente qué requisitos software se mantendrán y cuáles cambiarán.
- Para muchos tipos de software, diseño e implementación son tareas que se entremezclan.
- Análisis, diseño, implementación y pruebas son tareas con un grado de impredecibilidad desde un punto de vista de la planificación.
- Solución para abordar la impredecibilidad
  - Proceso del software debe ser adaptable.

## Programación extrema (XP)

Es el proceso ágil más conocido. Está indicado para proyectos con requisitos cambiantes, de alto riesgo y/o equipos de desarrollo pequeños (2-12 personas).

### Las cuatro claves de XP

- **Comunicación**
  - Entre miembros del equipo, gestores de proyecto y clientes
- **Simplicidad**
  - “Do the simplest thing that could possibly work”
  - “Never implement a feature you don’t need now”
- **Retroalimentación (feedback)**
  - Por el sistema (tests unitarios), por el cliente, por el equipo
- **Coraje**
  - Perder el miedo a refactorizar, quitar código obsoleto, persistencia para resolver los problemas que se presentan, etc.

### Características principales

- Planificación incremental (stories/story cards).
- Pequeñas versiones (releases) con valor de negocio.
- Diseño simple: cumplir con los requisitos actuales y no más.
- Test-first development.
- Refactorización.
- Pair programming.
- Integración continua.

### Críticas al modelo

- Modelo poco realista: muy centrado en el programador.
- No se escriben las especificaciones con detalle.
- Refactorizaciones constantes (consume tiempo).
- Las actividades que caracterizan al proceso son demasiado interdependientes.

# Scrum

Es una metodología ágil de gestión de proyectos que apareció en los 90. Se basa en un proceso iterativo que define un conjunto de prácticas y roles:

- **Roles:** scrum máster, team, product owner
- **Prácticas:** backlogs, sprints, meetings, demos

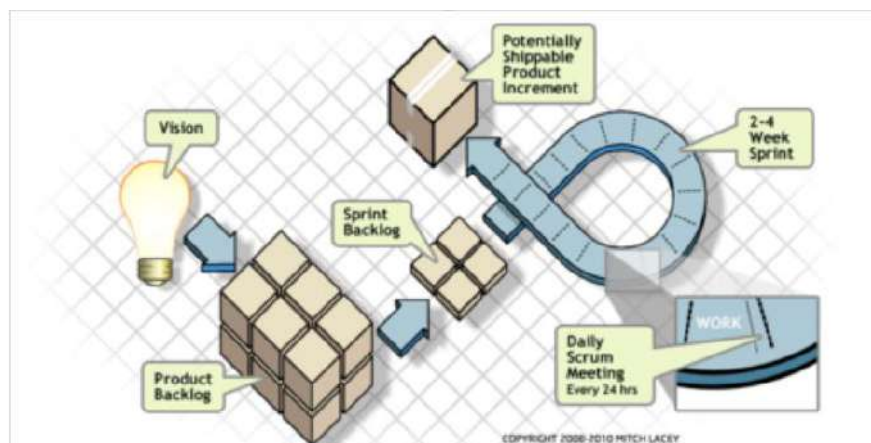
Ha mostrado ser muy útil en proyectos de tamaño pequeño y mediano.

## Características

- Se fomenta la comunicación entre los miembros del grupo.
- Conocimiento de las potencialidades.
- Optimización del tiempo.
- Interacciones continuas.
- Adaptabilidad a cambios y priorizaciones.
- Ciclos cortos: disminución de riesgo.
- Consenso con el cliente.
- Se asume que habrá cambios en el proyecto
- Entregas tras cada ciclo corto
- Se involucra al cliente / profesor en todo el proceso
- Todo el equipo participa en todas las fases
- Menos jerarquía y roles
- Objetivos comunes para todos
- Proceso de reflexión al final de cada sprint.

Scrum es adecuado para proyectos con Incertidumbre. Generará

- Autoorganización
- Autonomía: cada grupo elige la estrategia que considere adecuada
- Autosuperación: proyecto mejorándose progresivamente.
- Enriquecimiento de todos los miembros del grupo
- Transmisión del conocimiento.





## Tema 3 – Gestión de proyectos

Objetivos .....	2
Gestión del Proyecto Software .....	2
Importancia de la gestión .....	2
Características de la gestión .....	2
Actividades de la gestión .....	3
Actividades.....	3
Personal del proyecto .....	3
Planificación del proyecto.....	3
Proceso de planificación .....	4
Estructura del plan del proyecto.....	4
Organización de las actividades .....	5
Programación temporal o Calendario del Proyecto .....	5
Problemas en la planificación temporal. ....	6
¿Cómo estimar el tiempo de las tareas?.....	6
Gráficos de barras y redes de actividades .....	6
Otras herramientas de planificación.....	7
Asignación personal .....	7
Gestión del riesgo .....	8
Ejemplos de gestión del riesgo .....	8
El proceso de gestión del riesgo .....	9

---

# GESTIÓN DE PROYECTOS

---

## Objetivos

- Definir la gestión de proyectos de software y sus características
- Discutir la planificación de proyectos y el proceso de planificación
- Mostrar como la presentación gráfica de la planificación se puede utilizar en la Gestión de proyectos
- Discutir la noción de riesgo y el proceso de gestión de riesgos

## Gestión del Proyecto Software

El éxito de un proyecto software dependerá de si se lleva a cabo en el tiempo acordado (**planificación**), dentro del presupuesto acordado, cumple los requerimientos y hay buen ambiente en el equipo de trabajo.

La gestión del proyecto son las actividades que tienen por objetivo que el proyecto se lleve a cabo satisfactoriamente. El encargado de realizarlo es el **gestor del proyecto**.

### Importancia de la gestión

La Ingeniería de software es una actividad económica importante que está sujeta a restricciones económicas y a restricciones no técnicas. Los proyectos bien gestionados a veces fallan, pero los proyectos mal gestionados siempre fallan.

Solo se puede aprender a ser gestor del proyecto si desempeñas dicha función. La gestión de proyectos no es exclusiva de la ingeniería del software.

### Características de la gestión

- El producto a desarrollar es intangible.
- El producto tiene una flexibilidad excepcional
- La ingeniería de software no está reconocida como una disciplina de la Ingeniería con el mismo estatus de la mecánica, eléctrica, matemáticas, etc.
- El proceso de desarrollo de software no está estandarizado.
- La mayoría de los proyectos de software son "únicos".

# Actividades de la gestión

## Actividades

- Redactar propuestas
- Estimación del coste del proyecto
- Planificación y programación temporal del proyecto
- Seguimiento y revisión del proyecto
- Selección y evaluación del personal
- Redacción de informes y presentaciones

## Personal del proyecto

Son el elemento más importante en los proyectos. Puede ser imposible reclutar a la gente ideal para trabajar en el proyecto debido a cosas como que el presupuesto del proyecto no permite pagar altos salarios de personal experimentado, el personal con la experiencia necesaria puede que no esté disponible o que la organización puede preferir formar a sus empleados en las capacidades necesarias del desarrollo de proyectos de software.

Los gestores tienen que trabajar con estas limitaciones, especialmente cuando hay escasez de personal capacitado.

# Planificación del proyecto

**Plan del proyecto:** conjunto de actividades necesarias para desarrollar el proyecto.

*“El objetivo de la planificación del proyecto es proporcionar un marco conceptual que permita al gerente hacer estimaciones razonables de recursos, costo y calendario.”*

Probablemente es la actividad que más tiempo consume, ya que es una actividad continua desde el concepto inicial del proyecto hasta que se entrega y, además, los planes deben ser revisados regularmente a medida que está disponible nueva información.

## Proceso de planificación

Establecer las **restricciones del proyecto**

Realizar una **valoración inicial** de los parámetros del proyecto

**while** (el proyecto no finaliza o no se cancela) **loop**

Redactar la planificación del proyecto

Iniciar las **actividades** de acuerdo a la planificación

Esperar (durante algún tiempo)

Revisar el **avance** del proyecto

Revisar las **estimaciones** de los **parámetros** del proyecto

Actualizar la **planificación** del proyecto

Renegociar las **restricciones** y las **entregas** del proyecto

**if** (aparecen problemas) **then**

Iniciar una **revisión técnica** y sus posibles soluciones

**end if**

**end loop**

## El plan del proyecto establece

- Ámbito del proyecto
- Estudio de viabilidad
- **Análisis de riesgos**
- Los **recursos** disponibles para el proyecto
- Estimar costo y esfuerzo
- El desglose en **tareas** del trabajo
- Una **planificación** y un calendario para las tareas

## Estructura del plan del proyecto

- Introducción
- Organización del proyecto
- Análisis de riesgos
- Requisitos de software y hardware
- Desglose del trabajo (en tareas)
- Planificación y calendario del trabajo
- Seguimiento y mecanismos de información

## Organización de las actividades

Las actividades en un proyecto se deberían organizar para producir resultados tangibles que permitan a los gestores juzgar el progreso del proyecto.

Los **hitos** (milestones) marcan puntos importantes en el desarrollo del proyecto.

- Normalmente coinciden con el final de alguna actividad.
- Se usan generalmente para monitorizar el avance dentro del proyecto y la gestión de riesgos
- P.ej. Personal para el proyecto reclutado o hardware adquirido.

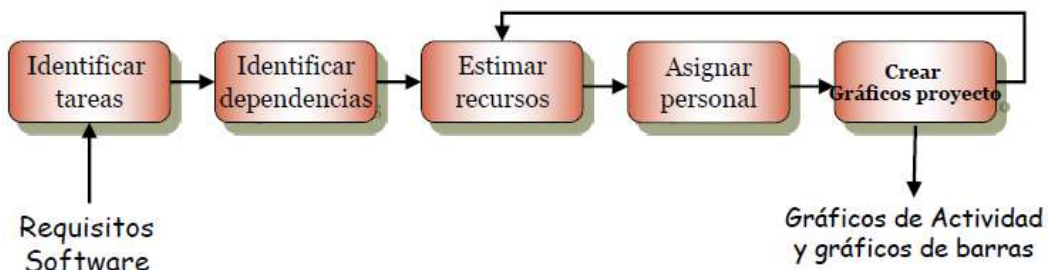
Los **entregables** (deliverables) son resultados (tangibles o intangibles) del proyecto

- P.ej. Documento o implementación del software



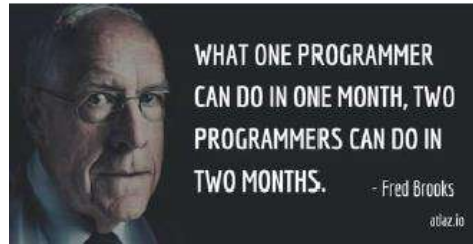
## Programación temporal o Calendario del Proyecto

- Distribuye el proyecto en **tareas** y estima el **tiempo** y los **recursos** requeridos para completar cada tarea
- **Organiza las tareas de forma concurrente** para mejorar el uso de la fuerza laboral
- **Minimiza dependencias entre tareas** para evitar los retrasos debidos a que una tarea espere a la terminación de otra
- Depende de la intuición y experiencia de los gestores



## Problemas en la planificación temporal.

- Es difícil estimar la longitud y dificultad de las tareas y esto dificulta la estimación del coste
- La productividad no es proporcional al número de personas trabajando en una tarea
- Incluir más personal en un proyecto avanzado retrasa el proyecto por los esfuerzos adicionales de comunicación
- Lo inesperado siempre sucede. Es necesario considerar siempre las contingencias



## ¿Cómo estimar el tiempo de las tareas?

Usando registros de proyectos previos y/o con experiencia del gestor.

Se suelen manejar **3 tiempos diferentes**:

**Tiempo optimista ( $t_o$ ):** el menor tiempo para realizar la tarea (asumiendo que todo va mejor que lo esperado).

**Tiempo pesimista ( $t_p$ ):** tiempo más largo para realizar la tarea (asumiendo que todo va mal, salvo catástrofes).

**Tiempo normal ( $t_n$ ):** estimación del tiempo de la tarea suponiendo que todo sucede según lo previsto.

**Tiempo esperado ( $t_e$ ):**

$$t_e = \frac{t_o + 4t_n + t_p}{6}$$

## Gráficos de barras y redes de actividades

Se utilizan notaciones gráficas para ilustrar la planificación del proyecto. Muestra la partición del proyecto en tareas.

Se suelen usar dos tipos de gráficas:

**Diagrama de Pert:** gráfica de actividades con las dependencias entre tareas y la ruta crítica

**Diagrama de Gantt:** gráficas de barras con la planificación en el tiempo del calendario de actividades

## Otras herramientas de planificación

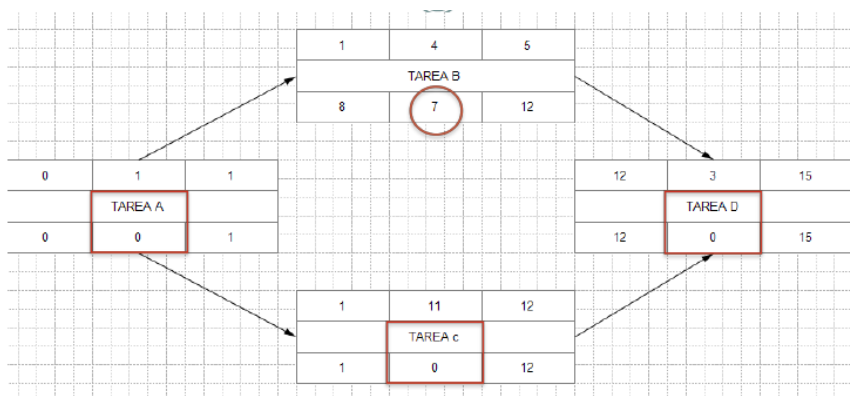
- Program Evaluation Review Technique (PERT) (también conocido como Critical Path Method (CPM))
- Técnica alternativa para organizar tareas, establecer intervalos de tiempo y mostrar dependencias
- Dos tipos de diagramas:
  - Activity on arrow
  - **Activity on node**

Early Start	Duration	Early Finish
Task Name		
Late Start	Slack	Late Finish

- **Duration** (duración) de la tarea.
- **Slack (holgura)**: relacionado con camino crítico. Cálculo:

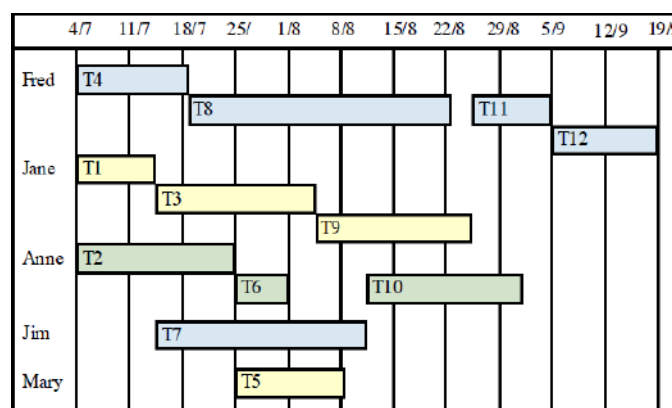
$$\text{SLACK} = \text{LATESTART} - \text{EARLY START}$$

$$\text{SLACK} = \text{LATEFINISH} - \text{EARLYFINISH}$$



**Camino crítico:** camino con holgura 0.

## Asignación personal





## Gestión del riesgo

Un riesgo es la probabilidad de que ocurra alguna circunstancia adversa.

- Los **riesgos del proyecto** afectan a la planificación y a los recursos.
- Los **riesgos del producto** afectan a la calidad y al rendimiento del software bajo desarrollo.
- Los **riesgos del negocio** afectan a la organización que desarrolla el software.

**Gestión del riesgo:** identificar los riesgos y preparar un plan que minimice sus efectos sobre el proyecto (contingencias).

### Ejemplos de gestión del riesgo

Riesgo	Tipo	Descripción
Movilidad del Personal	Proyecto	Personal con experiencia abandonará el proyecto antes de que finalice
Cambios de Gestión	Proyecto	Habrà un cambio de la gestión de la organización con prioridades diferentes.
No disponibilidad del Hardware	Proyecto	El hardware que es esencial para el proyecto no estará disponible en la fecha prevista.
Cambios de Requisitos	Proyecto y producto	Habrà un mayor número de cambios en los requisitos que los previstos.
Retraso en las Especificaciones	Proyecto y producto.	Las especificaciones de interfaces esenciales no estarán disponibles según la fecha prevista.
Subestimación del Tamaño	Proyecto y producto.	El tamaño del sistema se ha subestimado.
Herramientas CASE con menor rendimiento	Producto	Las herramientas CASE que soportan el proyecto no ofrecen el rendimiento o las prestaciones esperadas.
Cambios Tecnológicos	Negocio	La tecnología en la que se basa el proyecto es superada por una nueva tecnología.
Competencia del producto	Negocio	Sale al mercado un producto de la competencia antes que se termine el sistema.



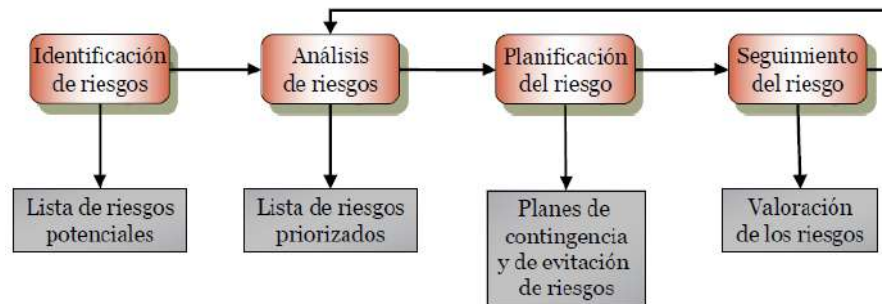
## El proceso de gestión del riesgo

**Identificación de Riesgos:** Identificar los riesgos del proyecto, producto y negocio

**Análisis del Riesgo:** Evaluar la posibilidad y las consecuencias de estos riesgos

**Planificación del Riesgo:** Redactar un plan de riesgos para evitar o minimizar sus efectos

**Seguimiento del Riesgo:** Supervisar los riesgos durante el proyecto



### Identificación de Riesgos

- Riesgos tecnológicos
- Riesgos del personal
- Riesgos de la organización
- Riesgos de los requisitos
- Riesgos de estimación

Tipo de riesgo	Posibles riesgos
<b>Tecnológicos</b>	La base de datos usada en el sistema no es capaz de procesar tantas transiciones por segundo como se esperaba. Los componentes software que se iban a reutilizar tienen defectos que limitan su funcionalidad.
<b>Personal</b>	Es imposible reclutar personal con las habilidades necesarias. Personal clave está enfermo o no está disponible en momentos críticos. Formación necesaria para el personal no está disponible...
<b>Organización</b>	Una reestructuración de la organización provoca un cambio de los gestores responsables del proyecto. Problemas financieros de la organización obligan a reducciones en el presupuesto del proyecto.
<b>Herramientas</b>	El código generado por las herramientas CASE es ineficiente. No se pueden integrar las herramientas CASE.
<b>Requisitos</b>	Se proponen unos cambios de requisitos que necesitan un importante rediseño Los clientes no aprecian la magnitud de los cambios de requisitos que proponen.
<b>Estimación</b>	Se subestima el tiempo necesario para el desarrollo del software. El índice de reparación de errores está subestimado. El tamaño del software (también) está subestimado.

## Análisis del riesgo

Evaluar la probabilidad y seriedad de cada riesgo.

**Probabilidad:** muy baja, baja, moderada, alta o muy alta.

**Efectos del riesgo:** catastrófico, serio, tolerable o insignificante.

Riesgo	Probabilidad	Efectos
Problemas financieros de la organización obligan a reducciones en el presupuesto del proyecto.	Bajo	Catastrófico
Es imposible reclutar personal con las habilidades necesarias.	Alto	Catastrófico
Personal clave está enfermo o no está disponible en momentos críticos	Moderado	Serio
Los componentes software que se iban a reutilizar tienen defectos que limitan su funcionalidad.	Moderado	Serio
Se proponen unos cambios de requisitos que necesitan un importante rediseño	Moderado	Serio
Una reestructuración de la organización provoca un cambio de los gestores responsables del proyecto.	Alto	Serio

Riesgo	Probabilidad	Efectos
La base de datos usada en el sistema no es capaz de procesar tantas transiciones por segundo como se esperaba.	Moderado	Serio
Se subestima el tiempo necesario para el desarrollo del software.	Alto	Serio
No se pueden integrar las herramientas CASE.	Alto	Tolerable
Los clientes no aprecian la magnitud de los cambios de requisitos que proponen	Moderado	Tolerable
Formación necesaria para el personal no está disponible..	Moderado	Tolerable
El índice de reparación de errores está subestimado.	Moderado	Tolerable
El tamaño del software está subestimado.	Alto	Tolerable
El código generado por las herramientas CASE es ineficiente.	Moderado	Insignificante

## Planificación del riesgo

Considerar cada riesgo y desarrollar una estrategia para manejarlo.

- **Estrategias para Evitarlos**
  - Se reduce la probabilidad de que el riesgo se produzca
  - *P.ej. Promover actividades de prevención de accidentes laborales.*
- **Estrategias de Minimización**
  - Se reduce el impacto del riesgo sobre el proyecto o el producto
  - *P.ej. Tener planes de sustitución de personal*
- **Planes de Contingencia**
  - Si se origina el riesgo, los planes de contingencia son planes para manejarlos
  - *P.ej. Recuperar pérdidas de datos con backups más reciente.*

Riesgos	Estrategias
Problemas financieros de la organización	Preparar un documento informativo para la alta dirección que muestre la forma en la cuál el proyecto realiza una importante contribución a los objetivos de la empresa
Problemas de reclutamiento	Alertar al cliente de las posibles dificultades y la posibilidad de retrasos Investigar la compra de componentes software.
Enfermedad del personal	Reorganizar el equipo de desarrollo para que haya más superposición de trabajo y, por lo tanto, el personal conozca el trabajo de otros compañeros.
Componentes defectuosos	Sustituir los componentes potencialmente defectuosos con componentes comprados de conocida fiabilidad.
Cambios en los Requisitos	Obtener la información de trazabilidad para evaluar el impacto de los cambios en los requisitos, Maximizar la ocultación de la información en el diseño.
Reestructuración de la Organización	Preparar un documento informativo para la alta dirección que muestre la forma en la cuál el proyecto realiza una importante contribución a los objetivos de la empresa
Rendimiento de la Base de Datos	Investigar la posibilidad de comprar una base de datos de alto rendimiento.
Subestimar el tiempo de desarrollo	Investigar la compra de componentes software Investigar el uso de generadores de código

Formación  
Online  
Especializada

Clases Online  
Prácticas  
Becas

Escuela de  
LÍDERES

Jose María Girela  
Bim Manager.

## Seguimiento del riesgo

Evaluar regularmente cada riesgo identificado para decidir si se está volviendo más o menos probable. Además, evaluar si los efectos del riesgo han cambiado.

Cada riesgo clave debería discutirse en las reuniones de seguimiento del proyecto.

Tipo de Riesgo	Indicadores Potenciales
Tecnología	Entrega tardía del hardware o del soporte software, muchos informes de problemas tecnológicos
Personal	Moral baja, Relaciones personales pobre entre el equipo, Mercado de trabajo
Organización	Rumores en la organización, ausencia de acciones de los gestores seniors
Herramientas	Resistencia del equipo a usar la herramientas, quejas sobre la herramientas CASE, demandas de estaciones de trabajo potentes
Requisitos	Muchas peticiones de cambio de requisitos, quejas de los clientes
Estimación	Fracaso para llegar a un acuerdo en la planificación, fracaso en la solución de problemas reportados

## Tema 4 - Requisitos

Introducción.....	2
Requisitos del Software .....	2
Ingeniería de requisitos .....	2
Tipos de requisitos .....	3
Requisitos funcionales y no funcionales .....	3
Requisitos funcionales vs no funcionales .....	4
Requisitos funcionales .....	4
Requisitos no funcionales .....	5
Relación entre RNFs y RFs.....	8
Requisitos del Dominio .....	8
Requisitos y diseño .....	9
Especificación de requisitos .....	10
Documento de definición del sistema .....	10
Documento de requisitos del sistema .....	10
Especificación de Requisitos del Software (SRS).....	11
Proceso de Ingeniería de requisitos.....	13
Estudio de viabilidad .....	14
Obtención y análisis de requisitos .....	14
Obtención y análisis de los requisitos.....	15
Actividades del proceso .....	16
Descubrimiento de requisitos .....	16
Fuentes de información .....	16
Técnicas de obtención de requisitos.....	17
Entrevistas.....	18
Casos de uso .....	19
Prototipos .....	20
Reuniones de grupo .....	20
Análisis de requisitos .....	20
Validación de los requisitos .....	22
Métodos.....	22
Cuestionario de validación de requisitos .....	22

---

# INGENIERÍA DE REQUISITOS

---

## Introducción

### Requisitos del Software

#### ¿Qué son?

Descripciones de los servicios que un sistema debe proporcionar y las restricciones a su modo de operación. Reflejan las necesidades del cliente para el sistema.

#### ¿Qué pueden abarcar?

Desde una declaración abstracta de alto nivel de un servicio, hasta la especificación matemática detallada y formal de una función del sistema.

#### Los requisitos tienen una doble función:

Pueden ser la base de la propuesta de un contrato: deben estar abiertos a ser interpretados

Pueden ser la base de un contrato: deben estar definidos detalladamente.

### Ingeniería de requisitos

Es el proceso para determinar, analizar, documentar y comprobar los requisitos de un sistema.

Los requisitos son, en sí mismos, la descripción de los servicios y las restricciones de un sistema que se descubren durante el proceso de IR (Ingeniería de requisitos).



## Tipos de requisitos

- **Requisitos de Usuario** (especificación de alto nivel)
  - Declaraciones en lenguaje natural y en diagramas de los servicios que el sistema debe proporcionar y las restricciones bajo las que debe funcionar.
  - Escrito para los clientes.
- **Requisitos del Sistema** (especificación funcional)
  - Documento estructurado donde se establecen con detalle los servicios y restricciones del sistema.
  - Describe lo que se implementará y no se implementará.
  - Escrito como un contrato entre el comprador y el desarrollador software.
  - Deben ser una especificación completa y consistente del sistema.

## Ejemplo

### Requisitos de usuario

1. El software debe proporcionar un medio para representar y acceder a archivos externos creados por otras herramientas.

### Requisitos del sistema asociados

- 1.1. Se suministrará al usuario los recursos para definir el tipo de archivos externos.
- 1.2. Cada tipo de archivo externo tendrá una herramienta asociada que será aplicada al archivo.
- 1.3. Cada tipo de archivo externo se representará como un icono específico sobre la pantalla del usuario.
- 1.4. Se proporcionarán recursos para que el usuario defina el icono que representará un tipo de archivo externo.
- 1.5. Cuando un usuario selecciona un icono que representa un archivo externo, el efecto de esa selección es aplicar la herramienta asociada con este tipo de archivo al archivo representado por el icono seleccionado.

## Requisitos funcionales y no funcionales

### Requisitos funcionales (RF)

Describen la funcionalidad o los servicios que se espera que el sistema suministre.

- Ej.: ¿Cómo reaccionará a determinadas entradas?
- Ej.: ¿Cómo se comportará en determinadas situaciones?

### Requisitos no funcionales (RNF)

Restricciones sobre los servicios o funciones ofrecidas por el sistema.

- Ej.: tiempo de respuesta, estándares, fiabilidad, capacidad E/S, etc.

### Requisitos del dominio

Requisitos que vienen impuestos por el dominio de la aplicación del sistema y que recogen características de ese dominio.

- Ej.: requisitos legales

## Requisitos funcionales vs no funcionales

Los **requisitos funcionales** describen las capacidades del sistema (**¿QUÉ HACE?**).

Los **requisitos no funcionales** describen las cualidades del sistema (**¿CÓMO HACE LO QUE HACE?**)

Muchos requisitos tienen una mezcla de ambos tipos (Ej. Aspectos de seguridad). Para facilitar su tratamiento los consideraremos como RNF.

Los RNFs son los que suelen influir más en la complejidad del proyecto. Además, son más difíciles de verificar que se cumplen.

## Requisitos funcionales

Describen los servicios y la funcionalidad del sistema. Dependen del tipo de software, del tipo de sistema donde se usará y de los posibles usuarios.

Los RF del usuario pueden ser declaraciones de alto nivel sobre lo que realizará el sistema, pero los RF del sistema deben describir los servicios del sistema detalladamente.

- Ej. funcionalidad del sistema, entradas/salidas y excepciones en detalle.

Son usados por los desarrolladores.

### Ejemplos

1. El usuario tendrá la posibilidad de buscar en el conjunto de toda la base de datos o en cualquier subconjunto que seleccione de ella.
2. El sistema proporcionará visores adecuados que permitan leer los documentos almacenados en el repositorio de documentos.
3. A cada pedido se le deberá asignar un identificador único (ID\_PEDIDO) que el usuario podrá copiar al área de almacenamiento permanente de la cuenta.

### Imprecisión en los requisitos

Aparecen problemas cuando los requisitos no están expresados con precisión. Los requisitos ambiguos pueden entenderse de forma distinta por los usuarios y por los desarrolladores.

Los desarrolladores tienden a entenderlos de forma que se simplifique la implementación.

*Ejemplo: "Visores adecuados"*

- Usuario: existan visores especializados para cada tipo de documento.
- Desarrollador: proporcionar un visor de texto que muestre el contenido de los documentos.



## *Compleitud y consistencia de los requisitos*

Los requisitos deben ser completos y consistentes.

**Compleitud:** Todos los servicios y funciones solicitadas por el cliente deben estar definidos.

**Consistencia:** No debe existir conflictos o contradicciones en la descripción de los servicios y funciones del sistema.

En la práctica, para sistemas grandes y complejos, es imposible obtener un documento completo y consistente.

## Requisitos no funcionales

Definen propiedades y restricciones del sistema.

- Ej. Fiabilidad, tiempo de respuesta, capacidad de almacenamiento, ...

Algunos requisitos del proceso pueden obligar a utilizar estándares de calidad, determinadas herramientas CASE o lenguajes de programación.

Pueden ser más críticos que los RFs.

- Si no se cumplen, invalidan el sistema.
- Ej. No se satisfacen requisitos de seguridad sistemas críticos (bancarios, control de avión): no se pueden certificar para uso.

Su implementación suele englobar a todo el sistema

- Suelen afectar la arquitectura global del sistema.
- Un requisito no funcional puede generar varios requisitos funcionales.
- Ej. Sistema seguro a transmisiones cifradas, login con password, etc.

## *Clasificación de los requisitos no funcionales*

**Requisitos del producto:** Especifican el comportamiento del producto.

- Ej.: Velocidad de ejecución, tasa aceptable de fallos, seguridad, usabilidad, ...

**Requisitos de la organización y del proceso:** Son consecuencia de las políticas y procedimientos de la propia organización (del cliente o desarrollador).

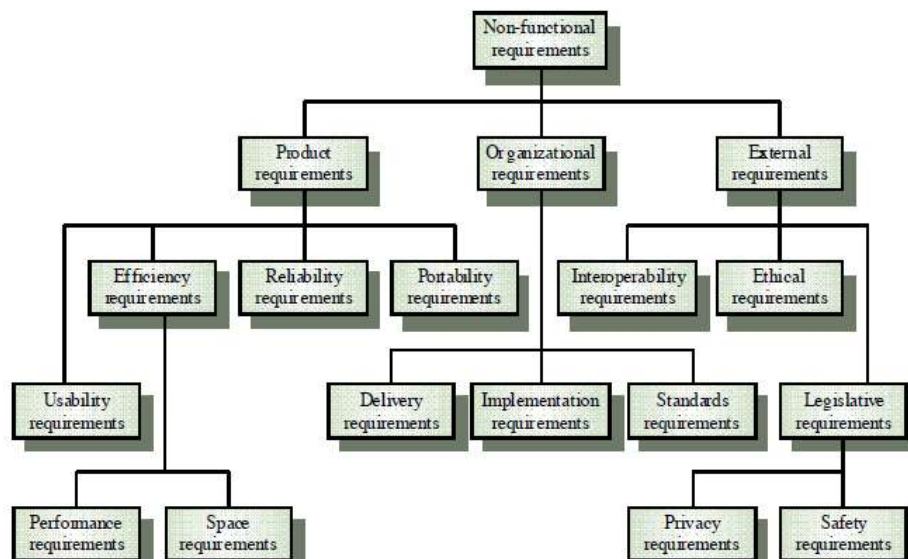
- Ej.: estándares de proceso, métodos de diseño, lenguajes de programación, tipo de documentación a entregar.

**Requisitos externos:** Se derivan de factores externos al producto y a su proceso de desarrollo.

- Ej.: Interoperabilidad del sistema con otros, requisitos legales (LOPD), éticos, ...

## Ejemplos

- **Requisito del producto**
  - 4.C.8. Se utilizará en todas las comunicaciones el conjunto de caracteres ASCII estándar.
- **Requisito de la organización**
  - 9.3.2 El proceso de desarrollo y los documentos entregados serán conformes con el estándar ISO -XXXX.
- **Requisito externo**
  - 7.6.5 El sistema no revelará datos de carácter personal de los clientes a los usuarios con nivel 3 o inferior excepto el nombre y número de referencia.



## Metas y requisitos

Los RNFs pueden resultar difíciles de definir con precisión, y los requisitos imprecisos son difíciles de verificar.

**Meta:** Un propósito general para el sistema (p.ej. facilidad de uso).

**Requisito no funcional verificable:** Aquellos que se pueden asociar a un objetivo medible (medibles cuantitativamente).

Las metas son útiles porque transmiten las prioridades de los clientes y usuarios.

## Ejemplos

- **Metas del sistema**
  - El sistema debe ser fácil de usar para personas experimentadas.
  - El sistema deberá minimizar los errores del usuario.
- **Requisitos verificables (detallados)**
  - Los usuarios experimentados deberán poder utilizar todas las funciones del sistema después de dos horas de entrenamiento.
  - Después de un entrenamiento, el número medio de errores cometidos por los usuarios experimentados no excederá de 2 errores al día.

## Medidas para requisitos no funcionales

Atributo	Medida
Rapidez	Transacciones procesadas por segundo Tiempo de respuesta al usuario/eventos Tiempo de actualización de pantalla
Tamaño	K Bytes Número de chips de RAM
Facilidad de uso	Tiempo de aprendizaje Número de ventanas de ayudas
Fiabilidad	Tiempo medio entre fallos Probabilidad de no disponibilidad Tasa de ocurrencias de fallos
Robustez	Tiempo de reinicio después de un fallo % de eventos que provocan un fallo Probabilidad de corrupción de datos por fallos
Portabilidad	% de sentencias dependientes del objetivo Número de sistemas objetivo

## Conflictos y compromisos

Debido a su naturaleza global, la mayoría de los RNFs interfieren con otros RNFs.

A veces pueden ser incompatibles: **Conflicto**.

- En estas situaciones debemos especificar en qué circunstancias qué RNF prevalece sobre los otros.
- Ej. Seguridad vs. anonimato.

A veces se oponen, pero no son incompatibles: **Compromiso**.

- Debemos especificar cuál es la relación óptima entre ellos.

Esto conlleva que la especificación de RNFs relacionados debe hacerse de manera conjunta o al menos referenciada.

## Relación entre RNFs y RFs

**Los RNFs suelen conllevar la introducción de RFs adicionales en el sistema**

- Ej. Confidencialidad de los datos à Funciones de cifrado.

**Los RNFs también pueden limitar la manera de implementar los RFs** o establecer restricciones de diseño.

- Ej. Transmitir datos confidenciales à Uso de determinados protocolos.

**La mayoría de los RNFs influyen en la arquitectura del sistema y su despliegue.**

- Ej. Almacenamiento de datos personales à La BD debe estar físicamente situada en Europa.
- Ej. Robustez ante fallos à uso de BD distribuida

**Los RFs pueden ser incompatibles con los RNFs**

- Ej. Monitorización de cámaras de vigilancia en una escuela vs. privacidad.

**El cumplimiento de los RNFs puede implicar cambios en la manera en la que los RFs se llevan a cabo.**

- Ej. El sistema debe permitir la generación de informes médicos de los empleados (RF), pero también debe mantener su privacidad (RNF): Anonimizar los informes.

Casi cualquier otra relación (directa e indirecta) entre RFs y RNFs es posible.

## Requisitos del Dominio

Describen las características del dominio en el que se encuadra la organización.

Pueden ser:

- Requisitos funcionales nuevos
- Restricciones sobre los existentes
- Definir cómo se deben realizar cálculos específicos.

Si no se cumplen, el sistema puede no trabajar de forma satisfactoria.



## Ejemplo

*En un sistema de protección de trenes, la desaceleración del tren se calculará como:*

$$D_{\text{tren}} = D_{\text{control}} + D_{\text{gradiente}}$$

*Donde*

*Dgradiente es  $9.81 \text{ ms}^2$  \* gradiente compensado/alfa*

*Los valores de  $9.81 \text{ ms}^2$ /alfa son conocidos para los diferentes tipos de trenes*

Es difícil de entender para alguien que no es un especialista en el tema (dominio). Sobre todo las implicaciones que puede tener con otros requisitos.

## Problemas con los requisitos del dominio

**Comprensibilidad:** Se expresan en la “jerga” de la aplicación del dominio (esta suele ser desconocida para los ingenieros del software).

**Requisitos implícitos:** Los especialistas entienden su área de tal forma que no creen necesario declarar de forma explícita los requisitos del dominio.

## Requisitos y diseño

En principio, los requisitos deberían establecer qué hará el sistema y el diseño cómo se implementará.

En la práctica, requisitos y diseño son inseparables

- Se puede definir una arquitectura inicial del sistema para estructurar los requisitos.
- Los sistemas pueden interoperar con otros ya existentes que obliguen a ciertos requisitos del diseño.
- El uso de un determinado diseño puede ser un requisito del dominio.

# Especificación de requisitos

Se refiere a la elaboración de un documento que puede ser sistemáticamente revisado, evaluado y aprobado.

Para sistemas complejos se producen hasta tres tipos de documentos:

- Definición del sistema
- Requisitos del sistema
- Requisitos de software

Para productos de software simple, sólo se suele utilizar el tercero (requisitos de software)

## Documento de definición del sistema

También conocido como documento de Requisitos del Usuario, establece los requisitos de alto nivel del sistema desde la perspectiva de dominio del problema.

Especifica el comportamiento externo del sistema.

- No debe incluir detalles de la arquitectura del sistema o su diseño.
- Dirigido a: usuarios y clientes

Se usa términos del dominio: Lenguaje natural.

## Documento de requisitos del sistema

Para sistemas con componentes software y hardware se suele separar:

- Descripción de los requisitos del sistema.
- Descripción de los requisitos de software.

Los requisitos del sistema se especifican y los requisitos de software se derivan de estos.

IEEE 1233 proporciona una guía para el desarrollo de los requisitos del sistema.

## Especificación de Requisitos del Software (SRS)

Establece las bases para un acuerdo entre clientes y contratistas o proveedores sobre:

- Lo que el producto software tiene que hacer.
- Lo que no se hará.

En los proyectos dirigidos al mercado general estas funciones las realizan los departamentos de marketing y desarrollo.

La SRS **permite hacer una evaluación rigurosa de los requisitos antes de comenzar el diseño** y reduce posteriores rediseños.

Debe proporcionar una base realista para la estimación de costes, riesgos y tiempos.

Los requisitos de usuario se suelen escribir en lenguaje natural, pero **la SRS debe complementarse con descripciones formales o semiformales**.

### *Alternativas para escribir la especificación de requisitos del sistema (SRS)*

Notación	Descripción
Lenguaje Natural	<ul style="list-style-type: none"><li>• Los requisitos se escriben con frases numeradas en lenguaje natural.</li><li>• Cada frase debe expresar un requisito.</li></ul>
Lenguaje Natural Estructurado	<ul style="list-style-type: none"><li>• Los requisitos se escriben en lenguaje natural en un formulario estándar o una plantilla.</li><li>• Cada campo proporciona información sobre un aspecto de los requisitos.</li></ul>
Lenguajes de Descripción de Diseño	<ul style="list-style-type: none"><li>• Se utiliza un lenguaje similar a un lenguaje de programación, pero con características más abstractas.</li><li>• Este enfoque se utiliza raramente aunque puede ser útil para las especificaciones de interfaz.</li></ul>
Notaciones gráficas	<ul style="list-style-type: none"><li>• Los modelos gráficos complementados con anotaciones de texto, se utilizan para definir los requisitos funcionales del sistema.</li><li>• Ej. Diagramas de UML.</li></ul>
Especificaciones matemáticas	<ul style="list-style-type: none"><li>• Basadas en conceptos matemáticos, tales como máquinas de estado finito o conjuntos.</li><li>• Eliminan la ambigüedad.</li><li>• Problema: la mayoría de los clientes no entienden una especificación formal.</li></ul>

## Especificación en lenguaje natural

Los requisitos se escriben como frases en lenguaje natural complementadas con diagramas y tablas.

Utilizado para definir los requisitos porque es expresivo, intuitivo y universal. Esto ayuda a que usuarios y clientes puedan comprender los requisitos.

### Pautas para escribir los requisitos

- Definir un formato estándar y usarlo de forma consistente para todos los requisitos.
- Utilizar el futuro simple para los obligatorios (deberá tener) y el condicional para los deseables (debería tener).
- Resaltar el texto con las partes claves del requisito.
- Evitar el uso de lenguaje técnico.

### Ejemplo

#### **Recurso de Cuadrícula**

*Para ayudar a la colocación de entidades en un diagrama, el usuario activará una cuadrícula en centímetros o en pulgadas, mediante una opción del panel de control.*

*Inicialmente la cuadrícula estará desactivada. Se podrá activar o desactivar en cualquier momento durante una sesión de edición y puesta en pulgadas o centímetros.*

*La opción de cuadrícula se mostrará en la vista de reducción de ajuste, pero el número de líneas de cuadrícula a mostrar se reducirá para evitar a la saturación del diagrama más pequeño con líneas de cuadrícula.*

### Problemas

Mezcla tres clases de requisitos

- Requisito funcional conceptual
  - La necesidad de una cuadrícula.
- Requisito no funcional
  - Unidades.
- Requisito no funcional de interfaz de usuario
  - Activación / Desactivación.



## Especificaciones en lenguaje estructurado

Forma restringida del lenguaje natural, la cual:

- Mantiene la expresividad y comprensión del lenguaje natural.
- Limita la terminología utilizada y emplea plantillas.
- Se reduce la ambigüedad y flexibilidad del lenguaje natural, incrementando de uniformidad.

Incorporan construcciones de control derivadas de los lenguajes de programación

- Ej. Para, si, entonces, ...

## Especificación basada en plantillas

- Definición de la función o entidad.
- Descripción de sus entradas y de donde provienen.
- Descripción de sus salidas y hacia donde van.
- Indicación de otras entidades utilizadas (si existen).
- Los efectos laterales (si hay).

## Proceso de Ingeniería de requisitos

Los procesos utilizados por la I.R. varían ampliamente dependiendo de

- Dominio de la aplicación.
- Personal involucrado.
- Organización que está elaborando los requisitos.

Sin embargo, hay un número de actividades genéricas comunes a todos los procesos



## Estudio de viabilidad

Permite decidir si el sistema propuesto es conveniente.

Es un estudio rápido y orientado a conocer:

- Si el sistema contribuye a los objetivos de la organización.
- Si el sistema se puede realizar con la tecnología actual y con el tiempo y el coste previsto.
- Si el sistema puede integrarse con otros existentes.

## Obtención y análisis de requisitos

### Obtención de requisitos

- El proceso mediante el cual **se captura el propósito y funcionalidades del sistema** desde la perspectiva del usuario.
- **Técnicas:** observación, entrevistas, herramientas CASE (UML).

### Análisis de requisitos

- El proceso de razonamiento sobre los requisitos obtenidos en la etapa anterior para **obtener una definición detallada** de los requisitos.
- **Técnicas:** representaciones gráficas (UML) y técnicas de revisión.

### Especificación de requisitos

- Proceso por el que se **documenta** el comportamiento requerido de un sistema software.
- **Técnicas:** notación de modelado y lenguajes de especificación.

### Validación de requisitos

- Proceso de **confirmación, por parte de los usuarios**, de que los requisitos especificados son válidos, consistentes y completos.
- Deben **definir el sistema que el cliente y los usuarios desean**.
- **Técnicas:** listas de comprobación y técnicas de revisión.

### Gestión de requisitos

- Proceso de manejar los requisitos cambiantes durante el desarrollo del sistema.
- **Técnicas:** herramientas CASE.

# Obtención y análisis de los requisitos

Es el proceso de descubrimiento de los requisitos.

El personal técnico trabaja con los clientes y usuarios para descubrir el dominio de la aplicación, los servicios que el sistema debe proporcionar y las restricciones operativas del sistema.

Puede involucrar a usuarios finales, gestores, ingenieros de mantenimiento, expertos del dominio, sindicatos, etc...

Se utiliza el término stakeholders (participantes/interesados) para referirse a todos ellos.



## Problemas

- Los participantes no conocen realmente lo que desean.
- Expresan los requisitos con sus propios términos.
- Pueden aparecer requisitos de distintos grupos que entren en conflicto.
- Influencia de factores políticos y de la organización.
- Cambio de requisitos durante el proceso de análisis.
- Pueden aparecer nuevos participantes o el entorno del negocio puede cambiar.

## Actividades del proceso



## Descubrimiento de requisitos

El proceso de recopilación de información sobre los sistemas propuestos o existentes y la destilación de esa información en los requisitos del usuario y del sistema.

Las fuentes de información incluyen la documentación, los participantes interesados en el sistema, y las especificaciones de sistemas similares.

## Fuentes de información

### Los objetivos generales o de alto nivel del software

Constituyen el motivo fundamental por el que se lleva a cabo el desarrollo del software. Un estudio de viabilidad es una forma relativamente barata de hacer esto.

### El domino del problema

El ingeniero de software debe adquirir, o tener a su disposición, el conocimiento sobre el dominio de aplicación. Otros actores lo suelen conocer en profundidad.

### Los propios participantes o actores del proceso y sus diferentes puntos de vista

Sobre la organización y sobre el software bajo desarrollo.



### El entorno operativo

Requisitos derivados del entorno en el que se ejecutará el software.

Ej.: Restricciones de tiempos para software de tiempo real o limitaciones de interoperabilidad para sistemas ofimáticos.

### El entorno de la organización

Al que debe adaptarse el software. A menudo, el software es necesario para dar soporte a un proceso de negocio.

Suele estar condicionado por la estructura, la cultura y la política interna de la organización.

### Técnicas de obtención de requisitos



## Entrevistas

En las entrevistas (formales o informales) el equipo interroga a los interesados sobre el sistema que utilizan y el sistema a desarrollar.

Hay dos tipos de entrevista:

- **Entrevistas cerradas** donde un conjunto predefinido de preguntas debe contestarse.
- **Entrevistas abiertas**, donde no hay un programa predefinido y donde se analizan una serie de temas con los interesados.

### *Entrevistas en la práctica*

Normalmente, una mezcla entrevistas cerradas y abiertas.

**Las entrevistas son buenas para conseguir una comprensión global** de lo que los interesados hacen y cómo podrían interactuar con el sistema.

**Las entrevistas no son buenas para entender los requisitos de dominio**

- Los ingenieros software pueden no entender la terminología específica de un dominio.
- Algunos de los conocimientos del dominio son tan familiares que los actores encuentran difícil articularlos o piensan que no vale la pena expresarlos.

### *Algunos tipos de preguntas*

- Sobre detalles específicos.
- Sobre la visión de futuro que el entrevistado tiene sobre algo del sistema.
- Sobre ideas alternativas.
- Sobre una solución mínimamente aceptable.
- Acerca de otras fuentes de información.

### *Escenarios*

Los escenarios son ejemplos reales de cómo se utilizará el sistema en la práctica.

Proporcionan un contexto para la obtención de los requisitos del usuario.

Permiten un marco de trabajo para realizar preguntas acerca de las tareas del usuario

¿qué pasaría si...? ¿cómo se hace?

**Deben incluir:**

- Una descripción de la situación inicial
- Una descripción del flujo normal de los acontecimientos
- Una descripción de lo que puede salir mal
- Información acerca de otras actividades concurrentes
- Una descripción del estado del sistema cuando finalice el escenario.

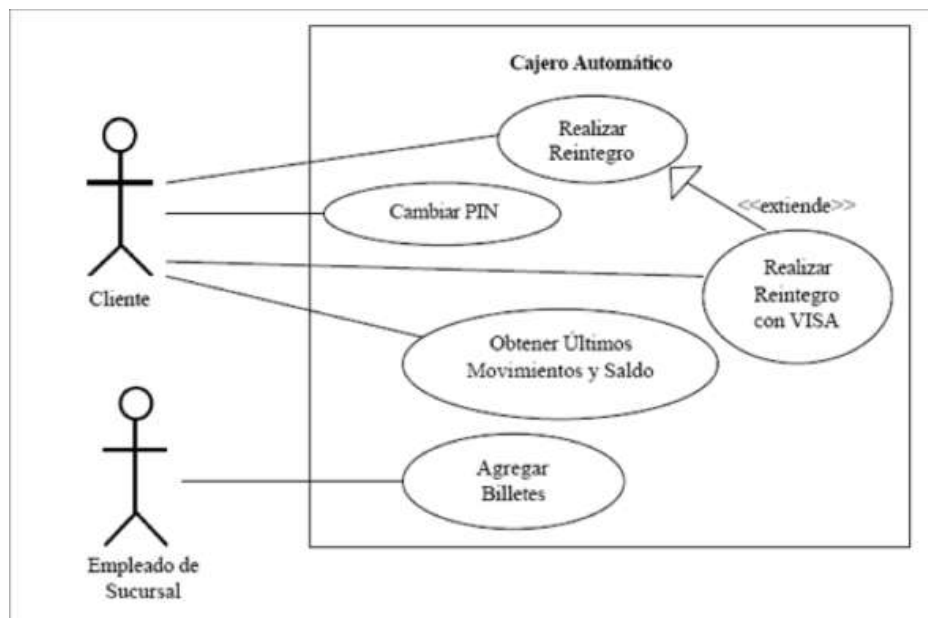
El tipo más común de escenario son los casos de uso.

## Casos de uso

Son una técnica basada en escenarios de UML que identifica a los actores involucrados en una interacción con el sistema y la describe.

Un conjunto de casos de usos describiría todas las posibles interacciones con el sistema.

Los diagramas de secuencia se pueden usar para añadir detalle a los casos de uso mostrando la secuencia de eventos procesados por el sistema.





## Prototipos

Herramienta valiosa para clarificar requisitos ambiguos.

Pueden actuar de manera similar a los escenarios, proporcionando a los usuarios un contexto dentro del cual pueden comprender mejor cuál es la información que necesitan proporcionar.

Existe una amplia gama de técnicas de creación de prototipos

Desde maquetas en papel de los diseños de pantalla hasta versiones beta de los productos software.

## Reuniones de grupo

**Propósito:** unificar requisitos y obtener más información sobre las necesidades software de un grupo de participantes que trabajando individualmente.

Pueden intercambiar y afinar ideas que pueden ser difíciles de sacar a la superficie a través de entrevistas. Aparecen los conflictos entre requisitos al principio.

Cuando funciona bien, esta técnica puede dar lugar a un conjunto más rico y más coherente de requisitos

Pero se deben manejar con cuidado:

- Necesidad de un facilitador.
- Evitar que la capacidad crítica del equipo se resienta por la lealtad de grupo.
- Evitar que se favorezcan las necesidades de los que participan abiertamente (tal vez los superiores) en detrimento de otras personas.

## Análisis de requisitos

- Detectar y resolver conflictos entre requisitos.
- Descubrir los límites del software y cómo debe interactuar con su entorno.
- Elaborar los requisitos del sistema para obtener, a partir de ellos, los requisitos del software a desarrollar.





## *Clasificación de requisitos*

Los requisitos se pueden clasificar según un número de dimensiones:

- Si el requisito es funcional o no funcional
- Si el requisito se deriva de uno o más requisitos de alto nivel o de una propiedad emergente o lo impone directamente un participante o alguna otra fuente.
- Si el requisito es del producto o del proceso.
- Ámbito del requisito
  - Medida en la cual un requisito afecta al software y a sus componentes.
  - Global (algún requisito no funcional) o particular
- Volatilidad / estabilidad
  - Algunos requisitos cambian durante el ciclo de vida del software, e incluso durante el desarrollo
  - Es útil tener alguna estimación de la probabilidad de cambio de los requisitos

## *Modelado conceptual*

**Objetivo:** ayudar a comprender el problema, en lugar de iniciar el diseño de la solución.

El desarrollo de modelos de un problema del mundo real es **clave para el análisis de requisitos del software**.

Se pueden desarrollar varios tipos de modelos:

- Modelos de datos
- Modelos de flujo de datos y de control
- Modelos de comportamiento (estados)
- Modelos de interacción
- Etc...

## *Negociación de los requisitos*

**Objetivo:** resolver problemas con los requisitos cuando dos partes requieren características incompatibles entre sí.

No es aconsejable que la decisión la tome el ingeniero software:

Debe conseguir un consenso entre las partes.

Es importante, sobre todo por motivos contractuales, que estas decisiones se puedan “trazar” hasta el cliente.

## Validación de los requisitos

**Objetivo:** demostrar que los requisitos definen el sistema que el cliente realmente quiere.

El coste de los errores en los requisitos es alto por tanto la validación es muy importante

- Arreglar un error de los requisitos después de la entrega puede costar hasta 100 veces el coste de arreglar un error de implementación.

### Métodos

Revisión de los requisitos	<ul style="list-style-type: none"><li>• Un grupo de personas examina los requisitos buscando inconsistencias, malentendidos, puntos poco claros, conflictos y otros problemas similares.</li></ul>
Prototipado	<ul style="list-style-type: none"><li>• Es una buena forma de probar cualquier producto.</li><li>• Gran ayuda para detectar problemas y clarificar requisitos.</li></ul>
Validación del Modelo	<ul style="list-style-type: none"><li>• Si se ha utilizado notaciones de modelado es posible realizar ciertas comprobaciones automáticas.</li></ul>
Pruebas de aceptación	<ul style="list-style-type: none"><li>• Elaborar un plan de aceptación mediante la verificación de requisitos.</li></ul>

### Cuestionario de validación de requisitos

<b>Conformidad con estándares</b>	La especificación en su conjunto, ¿es conforme a los estándares definidos? ¿es conforme a los estándares definidos cada uno de los requisitos?
<b>Seguimiento (Trazabilidad)</b>	¿Pueden identificarse unívocamente los requisitos? ¿Incluyen referencias o enlaces a otros requisitos relacionados así como las razones para incluirlos?
<b>Estructuración</b>	¿Está estructurado el documento de requisitos? ¿Están agrupados los requisitos relacionados entre sí? ¿Con otra estructura serían más fáciles de entender?
<b>Ambigüedad</b>	¿Son ambiguos algunos requisitos? ¿Pueden darse distintas interpretaciones de algunos de los mismos?
<b>Comprensión</b>	¿Son comprensibles los requisitos? ¿Puede un lector entender lo que significa cada uno de ellos? ¿Están libres de detalles de diseño e implementación?
<b>Consistencia</b>	¿Son consistentes los requisitos? ¿Hay alguna contradicción entre algunos requisitos?
<b>Compleitud</b>	¿Es completo el conjunto de requisitos? ¿Falta algún requisito? ¿Cada requisito de forma individual es completo? ¿Falta alguna información?
<b>Relevancia</b>	¿Individualmente es cada requisito pertinente para el problema y su solución?
<b>Gestionable</b>	¿Están expresados de forma que un cambio no afecte demasiado al resto? ¿Se han identificado las dependencias entre requisitos?
<b>Viabilidad</b>	¿Es posible implementar todos los requisitos con los recursos disponibles? ¿Son viables dadas las restricciones de coste y plazos?

## Revisiones

	Completo	Consistente	No ambiguo	Estructurado	Comprensible	Trazable
Req 1	X		X	X		X
Req 2		X		X	X	X
Req 3	X		X		X	X
...						
Req N	X	X	X			X

*Tabla de validación de requisitos*

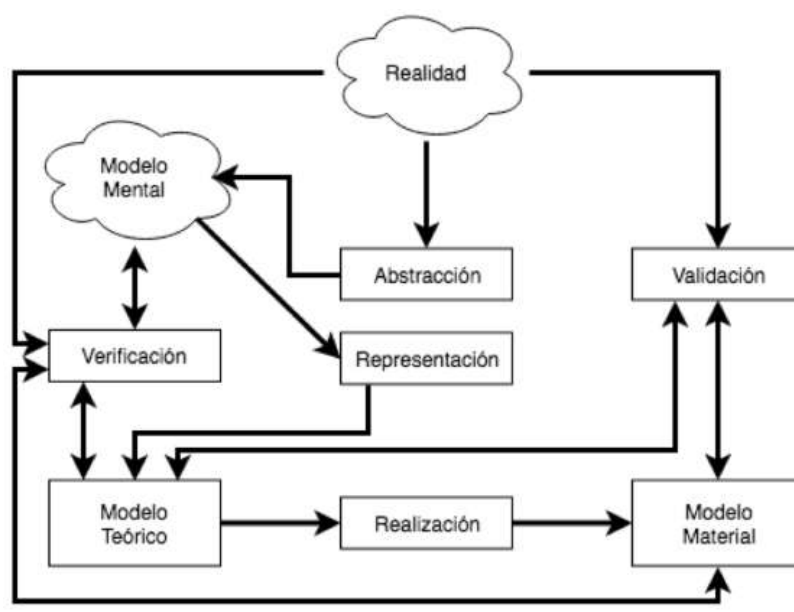
# Tema 5 – Modelado de sistemas con UML

Introducción.....	2
El proceso de modelado .....	2
¿Qué es un modelo? .....	2
Vistas de un modelo.....	3
Modelado de sistemas software.....	3
Perspectivas del sistema .....	3
Uso de los modelos gráficos .....	4
Lenguaje UML .....	4
Modelos de contexto .....	4
Perspectiva de procesos .....	5
Modelos de Interacción .....	5
Modelos de casos de uso .....	6
Diagramas de secuencia.....	6
Modelos Estructurales .....	6
Diagrama de clases .....	7
Modelos de Comportamiento.....	7
Modelos dirigidos por datos .....	8
Modelos dirigidos por eventos .....	8
Modelos de máquinas de estado .....	9

# MODELADO DE SISTEMAS CON UML

## Introducción

El proceso de modelado



## ¿Qué es un modelo?

Un modelo es una simplificación de nuestra percepción de una realidad que no siempre existe (p.ej. invención).

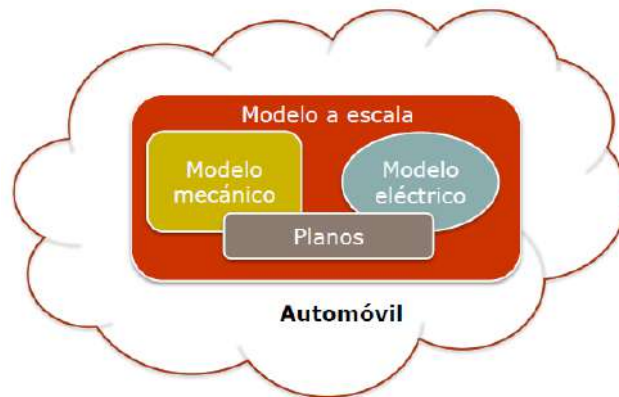
El modelo es una abstracción de la realidad que nos permite quedarnos con los aspectos más relevantes de la misma.

- El modelo es más simple de manejar y examinar.
- Nos permite hacer predicciones (p.ej. meteorología).

Validez de los modelos:

- **Validación:** concordancia con la realidad.
- **Verificación:** concordancia entre modelos.

## Vistas de un modelo



## Modelado de sistemas software

### ¿Para qué se usan?

El modelado del sistema permite al analista **entender la funcionalidad** del sistema.

Los modelos son utilizados para **comunicarse** con los clientes y el equipo de desarrollo.

Los diferentes modelos presentan diferentes **perspectivas (vistas) del sistema**.

Representar un sistema usando algún tipo de notación gráfica, que actualmente está casi siempre basada en **UML**.

## Perspectivas del sistema

Perspectiva externa	• Muestra el contexto o entorno donde debe funcionar el sistema.
Perspectiva de interacción	• Muestra las interacciones entre el sistema y su entorno o entre los componentes internos del sistema.
Perspectiva de comportamiento	• Modela el comportamiento dinámico del sistema y la forma en la que responde a los distintos eventos.
Perspectiva estructural	• Modela la organización del sistema o la estructura de los datos que se procesan en el sistema.

## Uso de los modelos gráficos

Son un medio para facilitar la discusión sobre el **sistema propuesto** (o que ya existe)

- Modelos incorrectos o incompletos son válidos si su papel es ayudar en la discusión

Pueden servir para documentar un **sistema existente**

- Los modelos deberían ser una representación precisa del sistema, pero no necesitan ser completos

Pueden servir **para describir detalladamente** el sistema

- Se puede utilizar para generar una implementación del sistema
- Los modelos tienen que ser correctos y completos

## Lenguaje UML

El lenguaje Unificado de Modelado (UML) es un lenguaje de modelado gráfico para:

- Describir y diseñar sistemas software.
- Especialmente orientado a diseño OO.

Desarrollado por Grady Booch, Ivar Jacobson y James Rumbaugh a mediados de los 90.

UML está organizado en un conjunto de elementos de modelado:

- Vistas.
- Diagramas.
- Bloques de modelado.
- Mecanismos

## Modelos de contexto

Se usan para mostrar el **contexto de funcionamiento del sistema** y para ilustrar sus **límites**.

- **Los límites del sistema** definen qué está dentro y qué está fuera del sistema.
- Muestran otros sistemas que se utilizan o dependen del sistema en desarrollo.

La posición de los límites del sistema tiene un efecto profundo sobre los requisitos del sistema.

Definir un límite del sistema es una decisión *política*.

- Es posible que haya presiones para definir los límites del sistema que aumentarán o reducirán la influencia o la carga de trabajo de las diferentes áreas de una organización.

Los **modelos arquitectónicos** muestran el sistema y sus relaciones con otros sistemas.

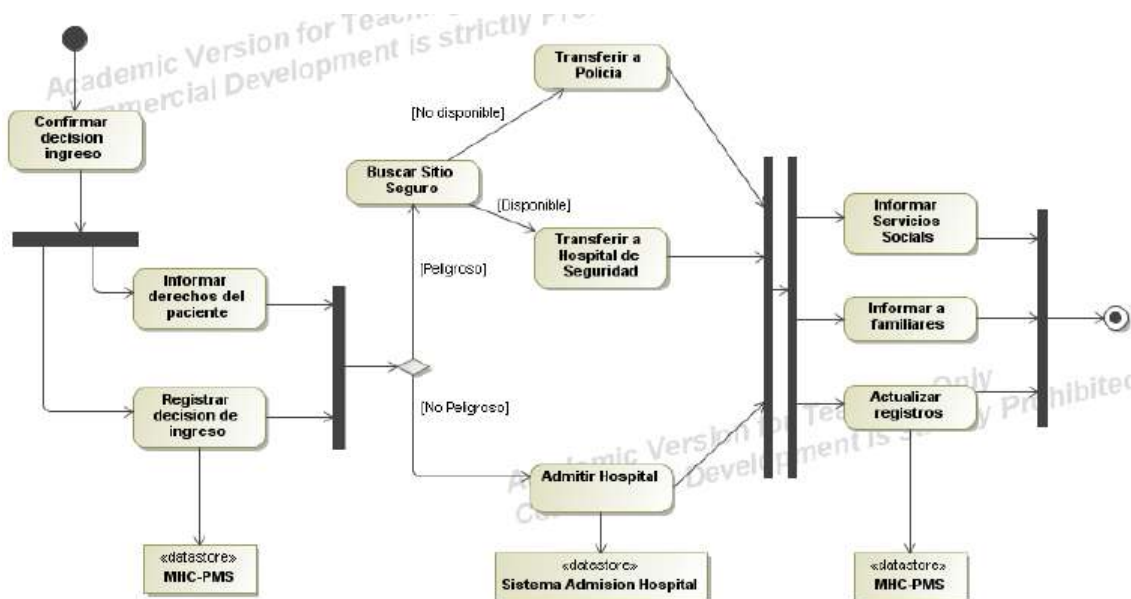


## Perspectiva de procesos

Los modelos de contexto sólo muestran a otros sistemas que se relacionan con el nuestro, pero no muestran cómo nuestro sistema se utiliza en dicho contexto (para esto se utilizan modelos de proceso).

Los **diagramas de actividad de UML** se pueden utilizar para definir Modelos de Procesos de Negocio (BPM - Business Process Models).

### *Modelo de procesos para ingreso involuntario (diagrama de actividad)*



## Modelos de Interacción

Modelar las **interacciones del usuario** es importante ya que ayuda a identificar los requisitos de estos.

Modelar las **interacciones de sistema a sistema** pone de relieve los problemas de comunicación que puedan surgir.

Modelar la **interacción entre componentes del sistema** ayuda a entender si la estructura del sistema propuesto es probable que proporcione el rendimiento y la fiabilidad solicitada.

Se pueden utilizar **los diagramas de casos de uso y de secuencia**.

## Modelos de casos de uso

Los casos de uso fueron desarrollados originalmente para **ayudar en la obtención de requisitos**.

Cada caso de uso representa una tarea discreta que implica la interacción con un sistema externo.

Los **actores** de un caso de uso pueden ser personas u otros sistemas.

Se representan de forma gráfica mediante un **diagrama de casos de uso** (UML) y se detallan de forma textual mediante algún documento o plantilla.

## Diagramas de secuencia

Los diagramas de secuencia se utilizan para modelar las **interacciones entre los actores y los objetos dentro de un sistema**.

Un diagrama de secuencia muestra la secuencia de interacciones que tienen lugar durante un caso de uso particular (**escenario**).

Los objetos y los actores involucrados se muestran en la parte superior del diagrama, con una línea de puntos trazada verticalmente a partir de éstos.

Las interacciones entre los objetos se indica por las flechas con etiquetas.

## Modelos Estructurales

Los modelos estructurales muestran la organización de un sistema en términos de los componentes que conforman ese sistema y sus relaciones.

Los modelos estructurales pueden ser

- **Modelos estáticos**, que muestran la estructura del diseño del sistema
- **Modelos dinámicos**, que muestran la organización del sistema cuando se está ejecutando.

Se pueden crear modelos estructurales de un sistema para discutir el diseño de su arquitectura.

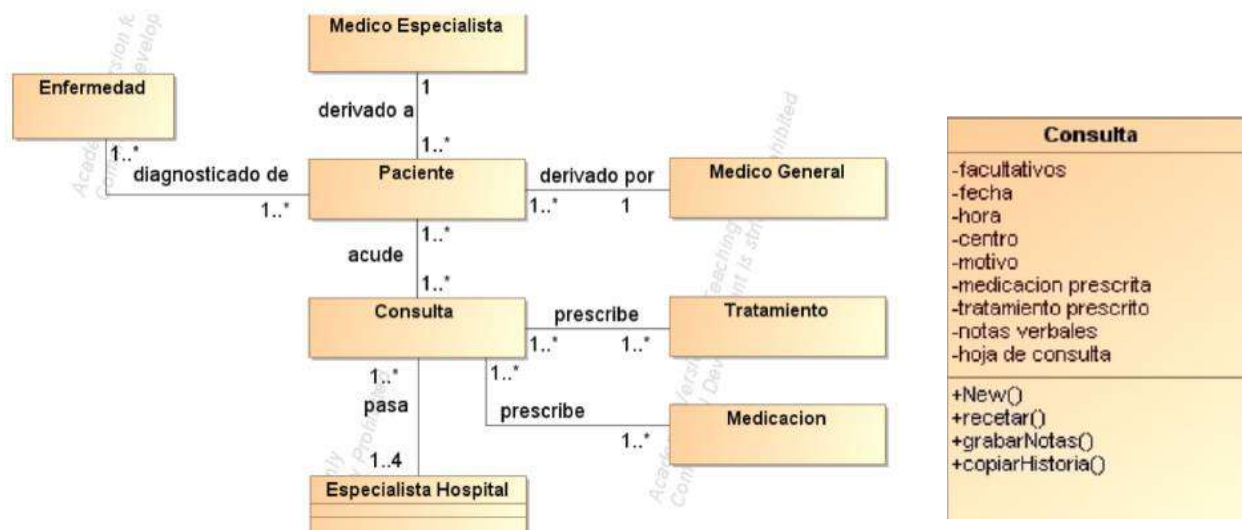
## Diagrama de clases

Los diagramas de clases se utilizan para mostrar las clases del sistema bajo desarrollo y las asociaciones entre estas clases.

Una **clase** se puede considerar como una definición general de un tipo de objeto del sistema.

Una **asociación** es un enlace entre clases que indica que existe alguna relación entre esas clases.

Durante las primeras etapas del proceso de ingeniería de software los objetos representan algo en el mundo real (Ej. Un paciente, una receta médica, médico, etc. Modelar el problema).



## Modelos de Comportamiento

Son modelos del **comportamiento dinámico de un sistema** que se está ejecutando.

Muestran **lo que sucede cuando el sistema responde a un estímulo** de su entorno.

Se puede pensar en estos estímulos como de dos tipos:

- **Datos.** Algunos datos que llegan al sistema tienen que ser procesados.
- **Eventos.** Ocurre algún evento que desencadena el procesamiento en el sistema. Los eventos pueden tener datos asociados.

## Modelos dirigidos por datos

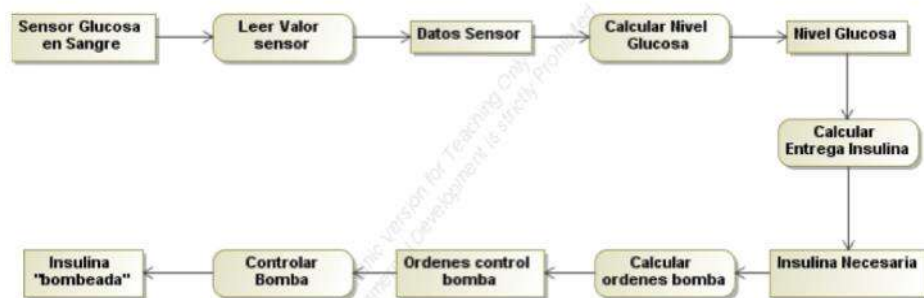
Muchos sistemas empresariales son sistemas de procesamiento de datos que están fundamentalmente basados en esos datos.

Están controlados por la entrada de datos al sistema, con relativamente poco procesamiento de eventos externos.

Los modelos dirigidos por datos muestran la secuencia de acciones implicadas en el procesamiento de los datos de entrada y en la generación de la salida asociada a ellos.

Son particularmente útiles durante el análisis de requisitos, ya que se pueden utilizar para mostrar el resultado extremo a extremo del procesamiento de un sistema.

### Diagrama de actividad



## Modelos dirigidos por eventos

Los sistemas de tiempo real están frecuentemente basados en eventos, con un mínimo procesamiento de datos.

Ej.: un sistema de conmutación de telefonía fija responde a eventos como "receptor descolgado", generando un tono de marcación.

Los Modelos dirigidos por eventos muestran cómo responde el sistema a los eventos externos o internos.

Se basa en la suposición de que un sistema tiene un número finito de estados y que los eventos (estímulos) pueden causar una transición entre estos estados.

## Modelos de máquinas de estado

Modelan el comportamiento del sistema en respuesta a eventos externos e internos.

Usados a menudo para sistemas de tiempo real, muestran las respuestas frente a estímulos.

Muestran los estados del sistema como nodos y los eventos como arcos entre nodos

- Cuando ocurre un evento, el sistema se mueve de un estado a otro.

Los diagramas de estado son parte integral de UML.

## Tema 5.1 – Casos de uso

Introducción.....	2
Elementos del diagrama .....	2
Actores .....	3
Caso de uso .....	3
Escenarios .....	4
Descripción gráfica de casos de uso .....	4
Especificación de un caso de uso .....	5
Especificación textual.....	5
Relaciones entre casos de uso .....	6
Generalización de casos de uso .....	7
Generalización de actores.....	7
Relación <<include>> .....	7
Relación <<extends>> .....	8
Diagrama de casos de uso.....	8

---

# CASOS DE USO

---

## Introducción

### ¿Qué son?

Los diagramas de casos de uso muestran la funcionalidad del sistema desde el punto de vista de un observador externo (qué hace el sistema sin importar cómo lo hace).

Capturan los requisitos funcionales del sistema y definen el límite entre el sistema y los elementos externos.

## Elementos del diagrama

### Actor

- Representa a cualquier elemento que intercambia información con el sistema.
- Son entidades externas al sistema.

### Caso de uso

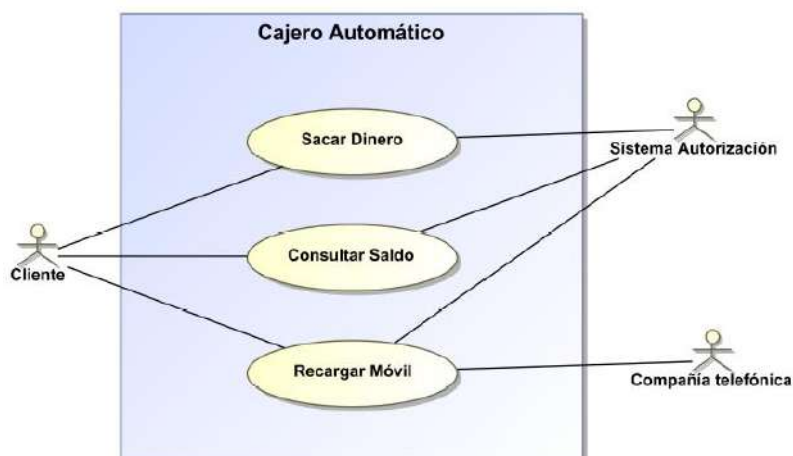
- Conjunto de secuencias de acciones e interacciones entre los actores y el sistema objeto de estudio para obtener una función o capacidad (requisito funcional).

### Asociación

- Representa una relación entre los elementos del diagrama.

### Escenario

- Describe una secuencia concreta del caso de uso durante una ejecución del sistema.
- Es decir, es una instancia del caso de uso.



## Actores

Son quién/qué inicia los eventos involucrados en una tarea (están fuera del sistema).

Pueden ser:

- ❖ **Usuarios con un rol en el sistema.**
  - *Ej. En una agencia de viajes*
    - Usuarios consultan posibles viajes.
    - Agentes añaden posibles viajes.
    - Administradores mantienen el sistema.
- ❖ **Dispositivos de E/S**, como sensores y/o actuadores, siempre que sean independientes de la acción de un usuario.
  - *Ej. En un regulador de temperatura, el sistema actúa en función de:*
    - Usuario establece la temperatura ideal.
    - El sensor de temperatura se encarga de activar/desactivar el calefactor.
- ❖ **Sistemas externos con los que el sistema se tiene que comunicar.**
  - *Ej. En un cajero automático de un banco:*
    - El cajero se comunica con un servicio de autorización para validar cliente, obtener dinero...
- ❖ **Temporizador o reloj en sistemas de tiempo real.**
  - *Ej. El sistema hace algo periódico como respuesta a un evento iniciado por un reloj (comprobar temperatura).*

## Caso de uso

Describen una interacción entre uno/varios actores y el sistema

1. Captura alguna función visible para el usuario.
2. Logra un objetivo discreto para el usuario (de principio a fin, tiene que acabar).
3. Especifica una funcionalidad completa (no se busca descomposición funcional del sistema).
4. Puede ser pequeño o grande.
5. Comparable a la definición clásica de “transacción”.

Ejemplos:

- *Introducir un pedido.*
- *Inicializar un cliente nuevo.*
- *Generar una factura.*



## Escenarios

Es una **secuencia de pasos que describe una interacción entre un usuario y el sistema.**

Proporciona un ejemplo de lo que ocurre cuando alguien interactúa con el sistema.

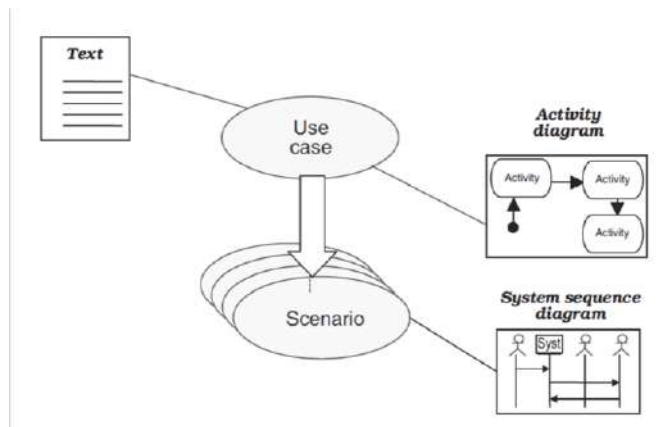
**Relación entre casos de uso y escenarios:** Un escenario es una concreción de un caso de uso. Dicho de otra forma, un caso de uso es un conjunto de escenarios que comparten un objetivo de usuario común.

## Ejemplo

*Un paciente llama a la clínica para concertar una cita para un chequeo anual. La recepcionista encuentra la cita libre más cercana en el registro de citas y programa la visita para esa cita.*

**Flujo alternativo:** ¿y si no hay citas disponibles en la clínica?

## Descripción gráfica de casos de uso



# Especificación de un caso de uso

Cada caso de uso debe tener una especificación verificable.

Especificación se puede hacer usando:

- Uno o varios diagramas de actividad o de interacción.
- Un documento o plantilla con texto.

## Especificación textual

La descripción se centra en lo que debe hacerse, no en la manera de hacerlo.

Definir varios escenarios posibles (**escenario principal y alternativos**)

- Cada uno identifica diferentes secuencias de interacciones o eventos entre los actores y el sistema.
- Detallar cada escenario narrativamente.

Si el escenario alternativo es complejo, separar con una nueva descripción. La secuencia de eventos de los escenarios se representar mediante un diagrama de secuencia.



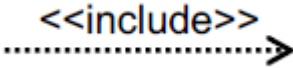
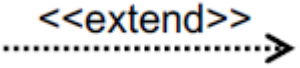
### Documento de especificación de un caso de uso

- **Identificador único**
- **Declaración de objetivos** (propósito del caso de uso)
- Autor
- Prioridad
- Riesgos
- Supuestos
- **Precondiciones y activación**
- **Escenario principal**
- **Escenarios alternativos**
- Finalización
- **Postcondiciones**
- Problemas o temas para destacar
- Requisitos funcionales relacionados
- Requisitos no funcionales relacionados
- Maquetas GUI

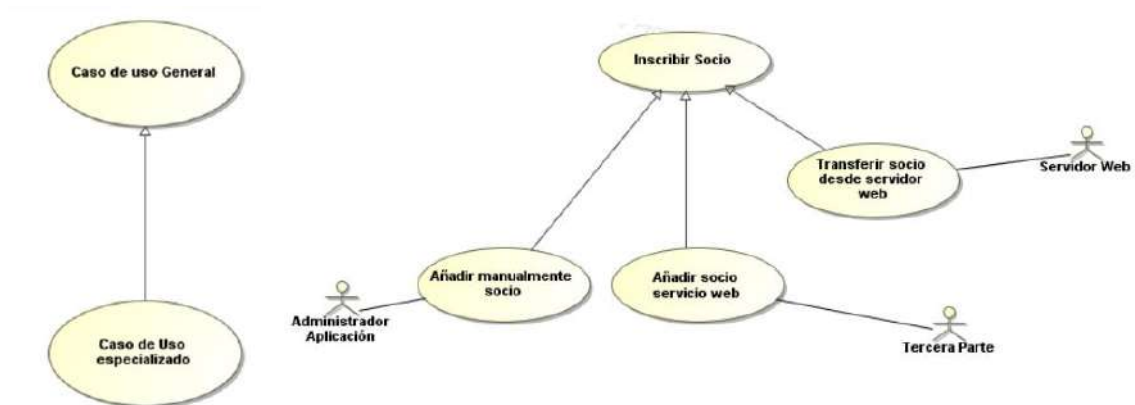
## Plantillas – Ejemplos

Use Case Name	Register Loan	Flow of Events	<ol style="list-style-type: none"> <li>1. The reader asks to borrow a book</li> <li>2. Librarian checks the reader's status to verify that the reader may borrow a book (alt 1)</li> <li>3. Librarian checks verifies that the reading item is available (alt 2)</li> <li>4. Verify that the book does not exceed the loan limit (alt 3)</li> <li>5. Librarian marks the reading item as loaned to the reader</li> <li>6. Librarian gives reading item to the reader</li> </ol>
Current Status	Designed, but not implemented	Alternative Flows	<ol style="list-style-type: none"> <li>1) The reader cannot borrow reading items if he/she is penalized for not returning reading items – error message</li> <li>2) The book is not available – error message</li> <li>3) The reader cannot take more than the limit of items for the loan – error message</li> </ol>
Goal Statement	Register a book loan	Non-Functional Requirements (Security, Performance, etc.)	The reader must not have to wait more than 30 seconds for the book loan to be registered
Author	Edita	Notes	
Priority	High		
Pre-conditions	The librarian (user) must already be authenticated and authorized		
Post-conditions	Reading item loan is registered to the customer		

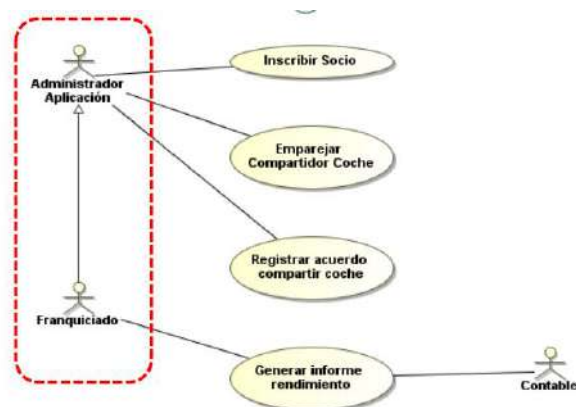
## Relaciones entre casos de uso

Relación	Función	Notación
<b>Asociación</b>	Interacción entre un actor y un caso de uso en el que participa.	
<b>Generalización</b>	Relación entre un caso de uso general y otro más específico que hereda características y añade otras.	
<b>Incluye</b>	Inserción de un fragmento de comportamiento dentro del caso de uso que lo incluye. Se usa para "factorizar".	
<b>Extiende</b>	Inserción de comportamiento adicional a un caso de uso bajo determinadas condiciones.	

## Generalización de casos de uso



## Generalización de actores



## Relación <<include>>

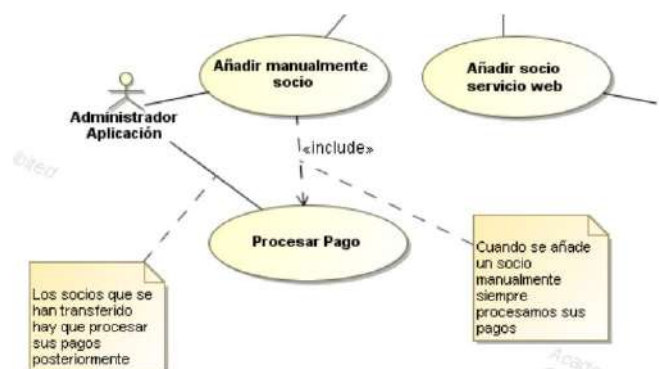
Una relación de inclusión denota que un caso de uso está incluido en otro.

Se da cuando

- Un caso de uso se utiliza por sí mismo,
- Y además otros casos de uso incluyen siempre esa funcionalidad.

También aparecen cuando dos o más casos de uso comparten una funcionalidad (factorizar).

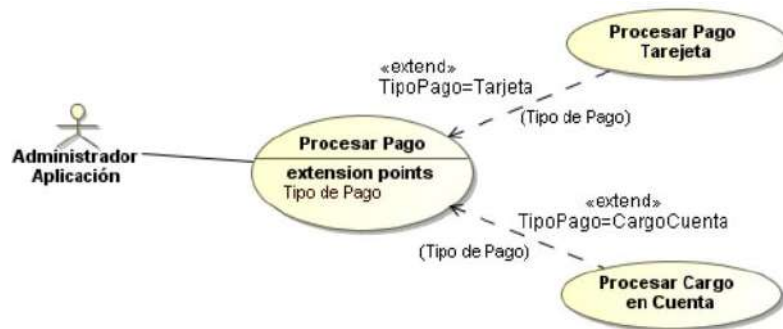
Denota que el caso de uso **SIEMPRE** está incluido en el otro caso de uso.



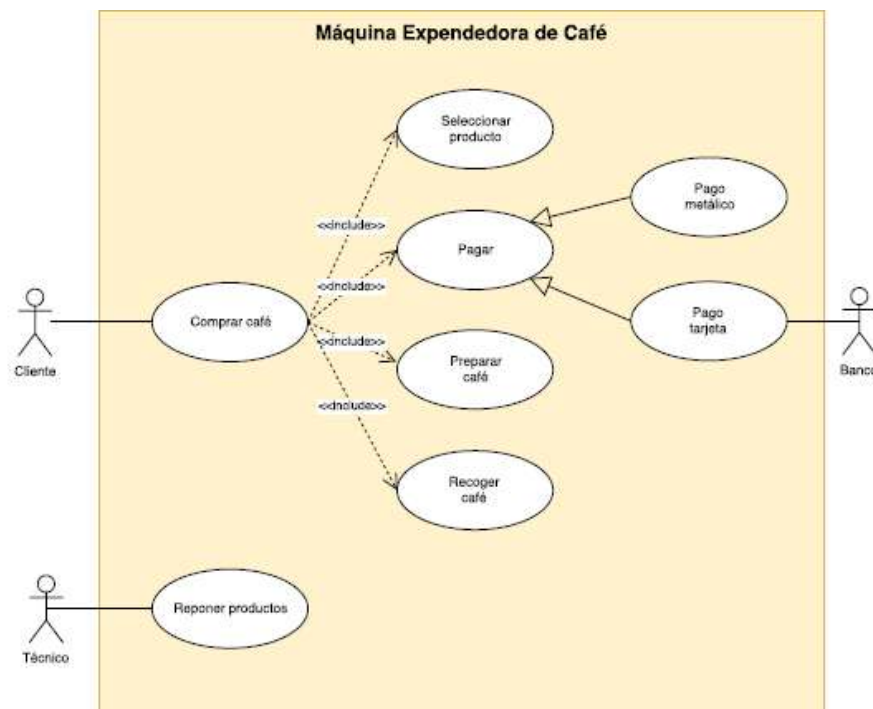
## Relación <<extends>>

Indica que **OPCIONALMENTE** un caso de uso es ampliado (extendido) por otro caso de uso

El caso de uso base declara uno o más puntos de extensión (condiciones) cuando se amplía el caso de uso.



## Diagrama de casos de uso



Nombre	Comprar café
Objetivo	El cliente adquiere un café de la máquina expendedora.
Dependencias	<ul style="list-style-type: none"> <li>- Seleccionar producto</li> <li>- Pagar producto</li> <li>- Preparar café</li> <li>- Recoger café</li> </ul>
Actores	<ul style="list-style-type: none"> <li>- Cliente</li> <li>- Banco (para pago con tarjeta)</li> </ul>
Pre-condiciones	<ul style="list-style-type: none"> <li>- La máquina debe estar enchufada.</li> <li>- El cliente se encuentra frente a la máquina.</li> </ul>
Post-condiciones	<ul style="list-style-type: none"> <li>- El café se ha servido al cliente o a éste se le ha mostrado un mensaje de error mostrando.</li> </ul>
Escenario principal	<ol style="list-style-type: none"> <li>1. El cliente selecciona el producto que desea consumir (caso de uso "Seleccionar producto").</li> <li>2. La máquina le muestra al cliente el importe del producto seleccionado.</li> <li>3. Se inicia el caso de uso "Pagar producto".</li> <li>4. Se inicia el caso de uso "Preparar café"</li> <li>5. El cliente recoge el producto elegido (caso de uso "Recoger café")</li> </ol>
Escenarios alternativos	<ol style="list-style-type: none"> <li>2. No existen existencia del producto seleccionado o de vasos → se le muestra un mensaje de error al usuario.</li> </ol>
Requisitos no funcionales	<ul style="list-style-type: none"> <li>- El tiempo de servir un café no superará 1 minutos.</li> </ul>

## Tema 5.2 – Diagrama de clases

Clases y objetos.....	2
Objetos.....	2
Clases .....	2
Objetos y clases.....	3
Clase.....	3
Atributos .....	3
Operación.....	4
Diagrama de clases Conceptual y de Implementación .....	5
Abstracción en UML.....	5
Relaciones .....	6
Asociaciones.....	6
Multiplicidad .....	6
Navegabilidad .....	7
Roles.....	7
Clases asociación.....	8
Asociación reflexiva .....	8
Agregación y composición .....	8
Agregación .....	8
Composición.....	9
Agregación/Composición vs. Asociación .....	10
Generalización .....	10
Clase abstracta.....	10
Modelado orientado a objetos .....	11
Modelado OO: Actividades .....	11
Buscando clases .....	11
Identificación de clases candidatas: Técnica de frases nominales .....	12

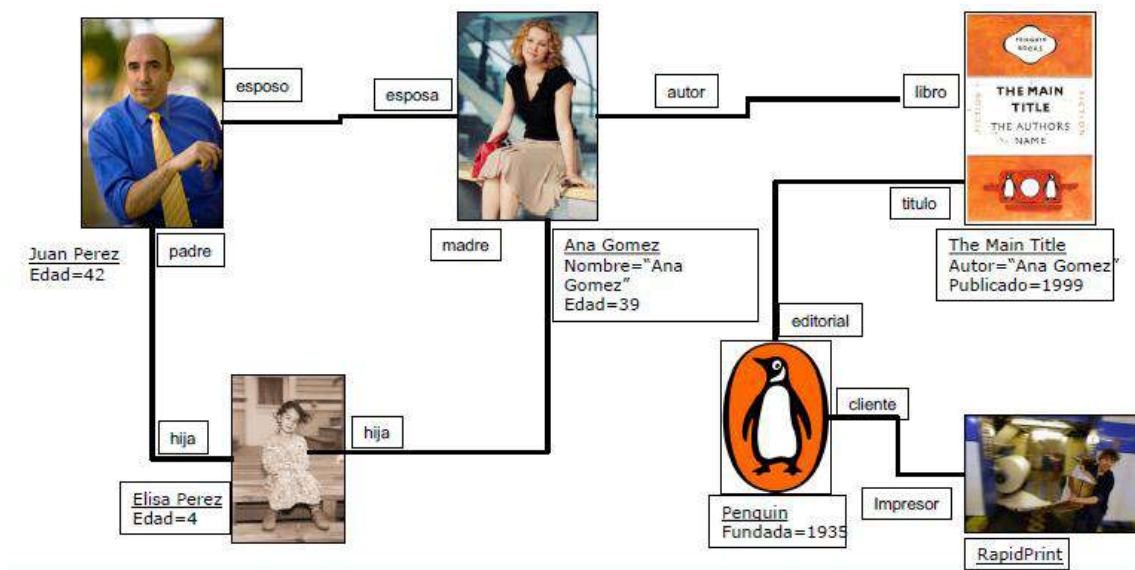


# DIAGRAMAS DE CLASES

## Clases y objetos

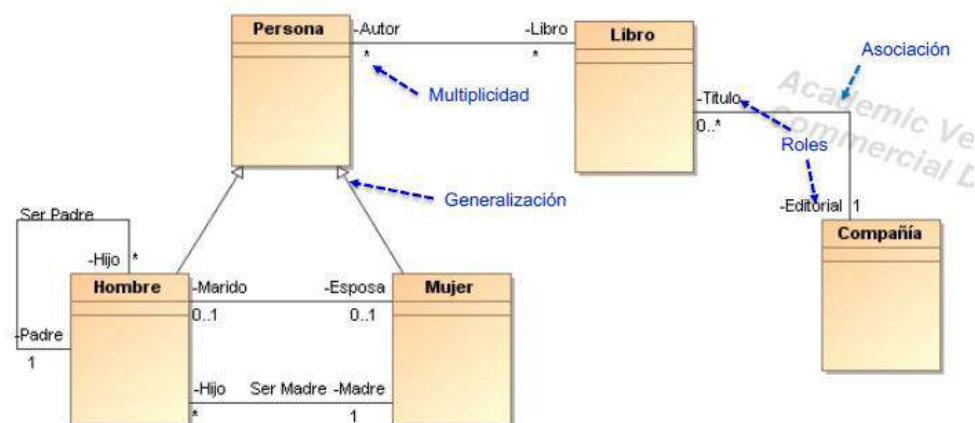
### Objetos

Un modelo de objetos, los valores de atributos reales y las relaciones entre ellos.



### Clases

Un modelo de tipos de objetos y los atributos y relaciones permitidas.





## Objetos y clases

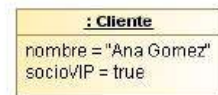
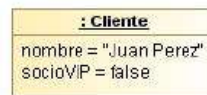
Un **objeto** es un concepto o entidad única, inequívocamente identificable (a menudo denominado una *instancia* de una clase).

*Objeto = Hechos*

### Definición: Objeto: Clase

Una **clase** es una plantilla que describe las propiedades de un concepto (a menudo referido como un *Tipo*)

*Clase = Reglas*

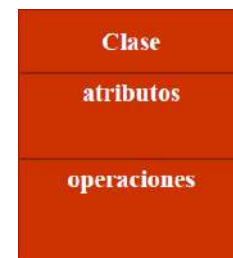


## Clase

Describe la estructura y el comportamiento de objetos que tienen las mismas características y semántica.

La estructura se describe mediante sus **atributos**.

El comportamiento mediante sus **operaciones**.



## Atributos

Representan la información almacenada en una clase (propiedad estructural de la clase). Representados como atributos integrados o por relación.

### Especificación

*Visibilidad nombre: tipo multiplicidad = valor\_defecto {propiedad}*

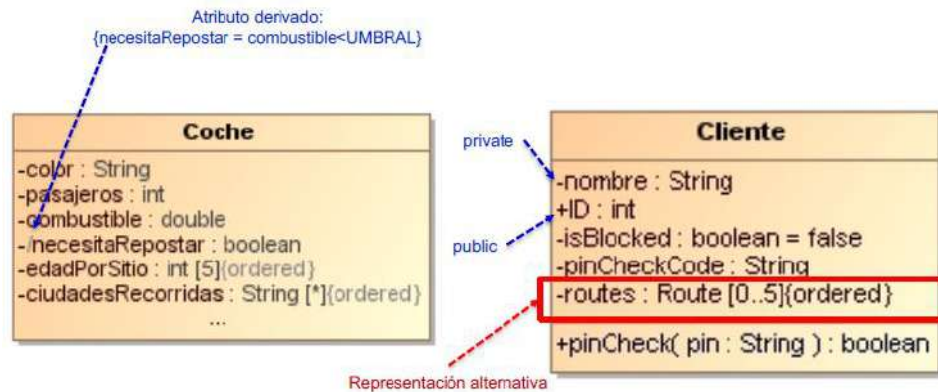
### Visibilidad

- Pública (+)
- Privada (-)
- Protected (#)

Formación  
Online  
Especializada

Clases Online  
Prácticas  
Becas

## Ejemplos



## Operación

Una **operación** define una propiedad de comportamiento de una clase. Cada operación tiene un objeto destino implícito.

Se puede aplicar la misma operación a distintas clases.

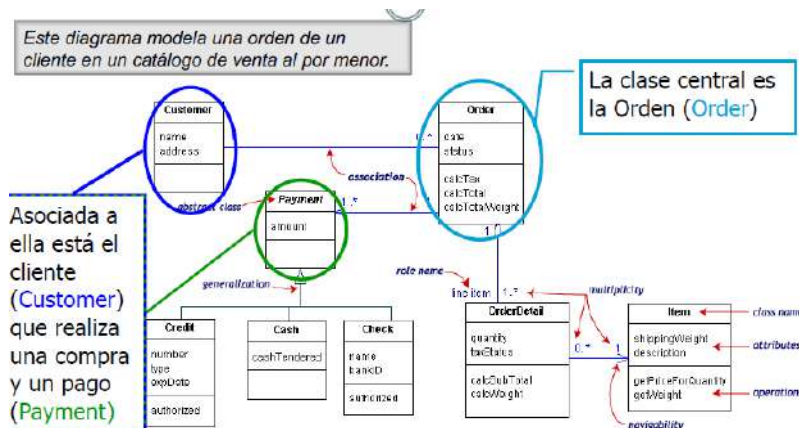
## Especificación

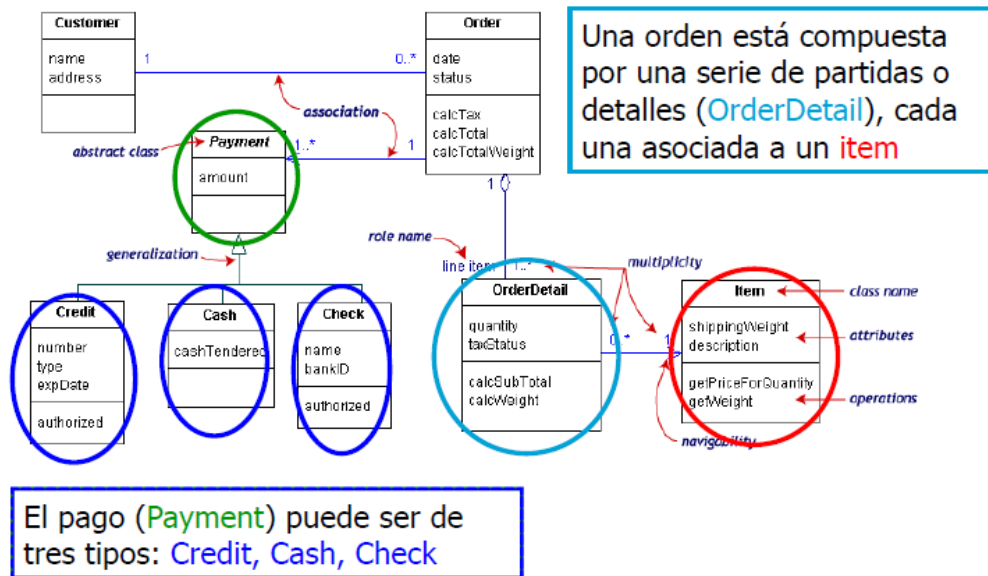
Visibilidad nombre (lista\_parámetros): tipo\_devuelto {propiedad}

## Ejemplos

- `generarFactura`
- `+concatenar(str1:String, str2:String): String`

## Ejemplo





## Diagrama de clases Conceptual y de Implementación

Un diagrama de clases puede ilustrar varios niveles de detalles y diferentes objetivos, por ejemplo...

### Diagrama de Clases Conceptual

- Desarrollado por los analistas de sistema (negocio) para modelar los recursos del sistema (negocio).
- Contiene tipos de datos y operaciones del negocio.
- Contiene solo clases del “negocio” – no clases de la implementación.

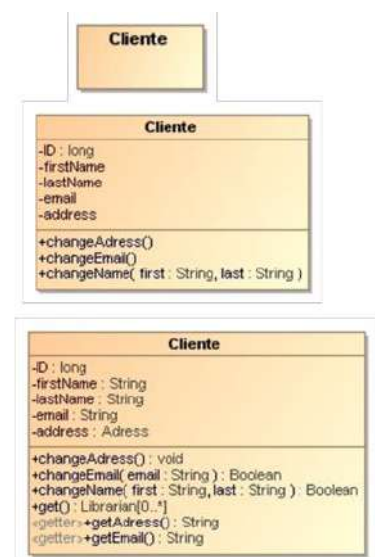
### Diagrama de Clases de Implementación

- Desarrollado por los diseñadores para modelar los requisitos del código.
- Contiene detalles de programación tales como tipos de datos del lenguaje o de la base de datos y modificadores de acceso.
- Se puede usar para generar el código.

## Abstracción en UML

1. Conceptos de análisis inicial.
2. Detalles del **análisis** totalmente especificado.
3. Detalles del nivel de **implementación**.

**Idea:** ocultar detalles para no complicar la visión de conjunto del diagrama (dejar sólo lo que necesitamos)



# Relaciones

## Asociaciones

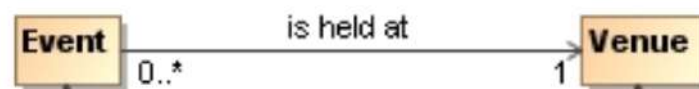
**Asociación:** Relación estructural entre dos clases.

Por defecto bidireccionales, aunque pueden ser unidireccionales.

Las asociaciones están caracterizadas por:

- Nombre de la asociación.
- Roles.
- Multiplicidad.
- Navegabilidad.

Especifica que objetos de un tipo tienen una referencia a objetos de otro tipo.



Permite a los objetos de una clase contactar los objetos de otra clase para acceder a sus datos y su comportamiento.

## Multiplicidad

Indica cuántos objetos de un extremo de la asociación pueden conectarse con un objeto del otro extremo.



### Multiplicidad

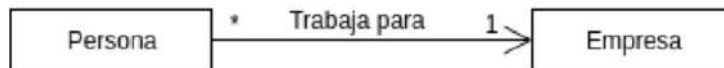
0..1	0 ó 1
*	Muchos
n	Exactamente n
m,n	m o n
m-n	Entre m y n
n+	Más de n
	Exactamente 1

# Navegabilidad

Aparece en **asociaciones unidireccionales**.

- Una flecha muestra la dirección en la cual la asociación puede ser atravesada (o preguntada).
- La flecha nos permite conocer quien “**posee**” la implementación de la asociación.

Las asociaciones sin flecha son bidireccionales.



Especifica qué objetos de un tipo tienen una **referencia** a objetos de otro tipo.

Permite a los objetos de una clase contactar con los objetos de otra clase para acceder a sus datos y su comportamiento.

**Ejemplo anterior:**

- Podríamos preguntarle a la persona sobre la empresa donde trabaja.
- Pero la empresa no puede decirnos qué personas trabajan en ella.

## Roles

El **rol** indica el papel o la cara que la clase de un extremo de la asociación presenta a la clase del otro extremo. Una asociación binaria tiene dos roles.

Se pueden nombrar explícitamente:

- Para recorrer asociaciones
- Para especificar direccionalidad.

Los nombres de los roles:

- Deben ser únicos en asociaciones que parten de una misma clase.
- Son necesarios para asociaciones entre objetos de la misma clase.



## Clases asociación

**Clase asociación:** es una clase que se conecta a una asociación para definir propiedades de la asociación.

La clase asociación está ligada a la conexión formada entre las clases.

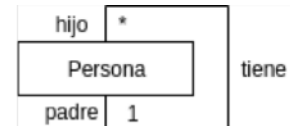
- Cada enlace conlleva una instancia de la clase asociación.

Común en asociaciones muchos a muchos y uno a muchos.



## Asociación reflexiva

La **misma clase** puede ser principio y final, pero eso no implica un único objeto.



## Agregación y composición

### Agregación

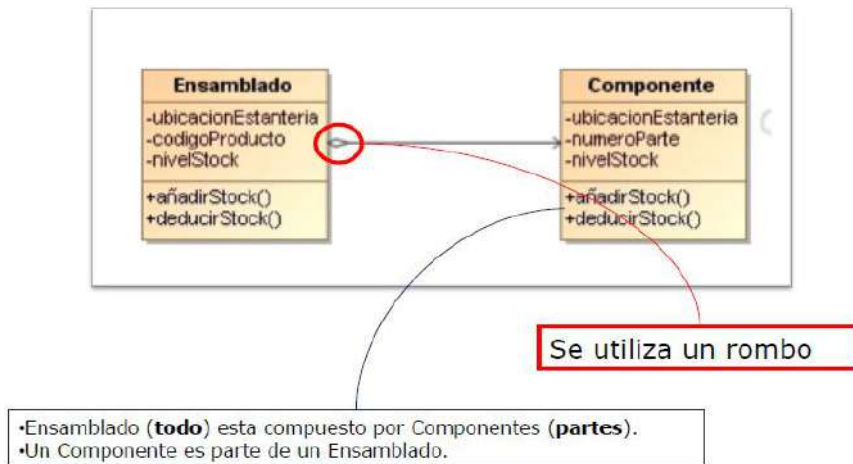
Una **agregación** (relación “*es parte de*”) es una asociación en la que una de las partes representa el todo y la(s) otra(s) la(s) parte(s).

Indica que una instancia de una clase:

- Además de tener sus propios atributos
- Puede estar compuesta por (o incluir) instancias de otras clases.

Es usual la **propagación** de operaciones entre un objeto y sus componentes.



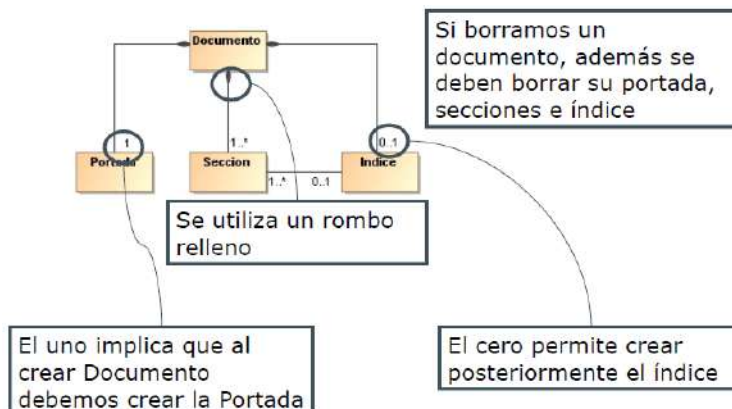
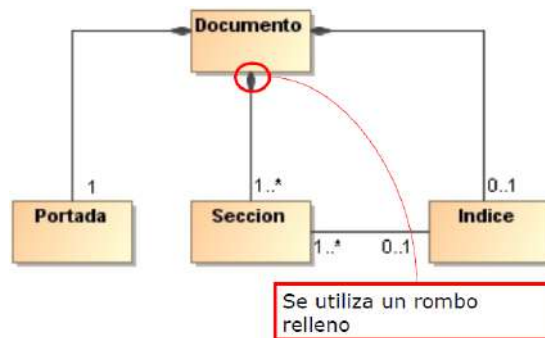


## Composición

También modela una relación "ser-parte-de. Pero además **implica que el ciclo de vida de la parte está ligado al del todo**.

En una composición una "parte" no puede existir sin ser parte de un "todo".

- En la agregación esto no es así (un componente podría existir sin estar ensamblado).
- Un objeto que representa a la "parte" sólo puede pertenecer a un "todo".



## Agregación/Composición vs. Asociación

Si los objetos instanciados de las clases están ligados por una relación todo/parte:

- La relación es de agregación/composición.

Si los objetos se consideran independientes aun cuando haya enlaces frecuentes entre ellos:

- La relación es una asociación.



## Generalización

Una **generalización** (relación “*es un tipo de*”) es una relación desde una clase especializada hasta una clase general.

- Relación entre un elemento general (superclase, padre o clase base) y uno más específico (subclase, hijo o clase derivada).

Cada subclase **hereda** todos los atributos y operaciones de la superclase y después los **especializa** de diferentes formas.

Las subclases **extienden** a las superclases con nuevos atributos y operaciones o **redefinen** la implementación de las operaciones heredadas.

## Clase abstracta

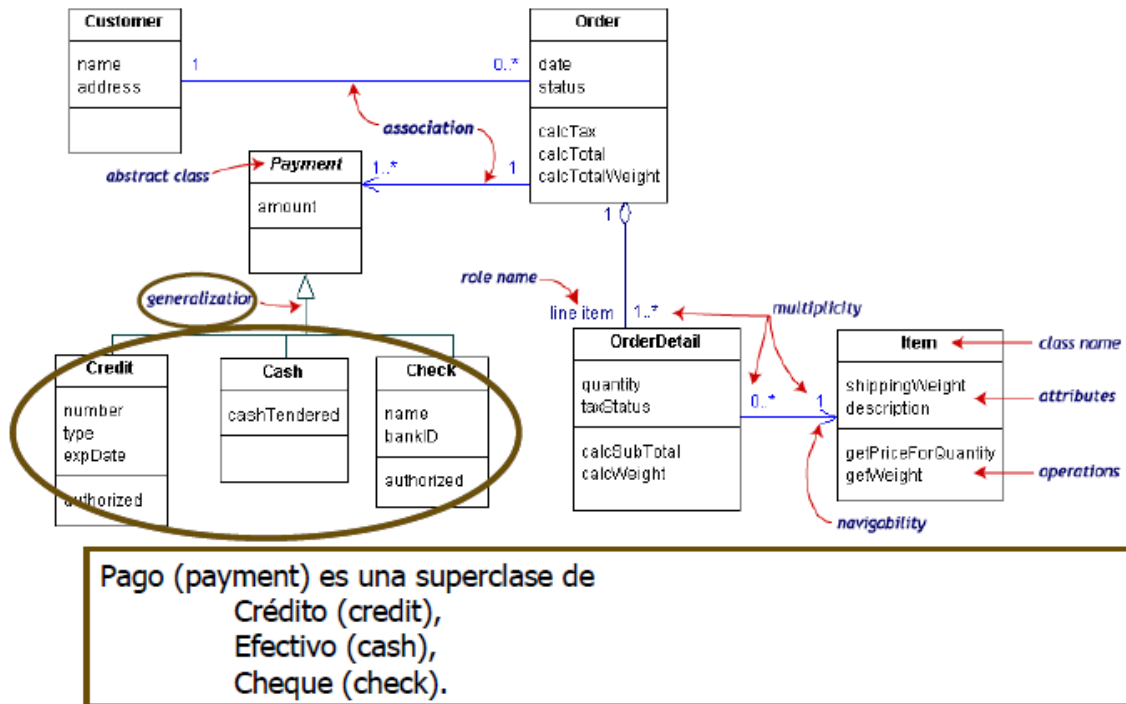
Una **clase abstracta** es una clase no instanciable.

Suele contener una o más **operaciones abstractas** (sin implementación) que las clases derivadas definen.

Se especifica con el estereotipo <<abstract>> y el nombre en cursiva.



## Ejemplo



## Modelado orientado a objetos

### Modelado OO: Actividades

1. Identificar los objetos y seleccionar las clases candidatas.
2. Obtener los atributos de las clases necesarios para satisfacer requisitos de información.
3. Buscar las relaciones entre clases.
4. Asociar los atributos con clases o relaciones entre clases.

### Buscando clases

#### Clases del Modelo del Dominio

- Información del negocio que se debe almacenar o analizar.
- Sustantivos en la descripción del problema, políticas y procedimientos del negocio, material de formación y productos en funcionamiento.
- Roles desempeñados por los actores.

Formación  
Online  
Especializada

Clases Online  
Prácticas  
Becas

Escuela de  
LÍDERES

Jose María Girela  
Bim Manager.

## Clases de Implementación

- Servicios del negocio.
- Elementos del modelo de datos.
- Vistas gráficas.
- Controladores (manejador eventos, flujo, y lógica de control).
- Bibliotecas de clases, componentes.

## Identificación de clases candidatas: Técnica de frases nominales

Técnica para identificar componentes del modelo en las descripciones textuales del dominio (p.ej. casos de uso).

Partes del habla	Componente del modelo	Ejemplos
Sustantivo propio	Objeto	Alicia
Sustantivo común	Clase	Persona, Registro
Verbo de acción	Operación	Crea, Envía, Selecciona
Verbo de ser	Generalización	Es un tipo de, es alguno de
Verbo de tener	Agregación	Tiene, consiste en, incluye
Verbo modal	Restricciones	Debe ser
Adjetivo	Atributo	Descripción de incidente, color de coche

-- Ejercicio de transparencias --

## TEMA 5.3 – Diagrama de secuencia

Elementos de los Diagramas .....	3
Participantes .....	3
Líneas de vida y Activación .....	3
Mensajes .....	4
Creación y destrucción.....	4
Comunicación síncrona.....	5
Comunicación asíncrona .....	5
Mensajes encontrados.....	6
Mensajes perdidos.....	6
Auto-mensajes .....	6
Fragmentos combinados.....	7
Alternativas.....	7
Bucles.....	8
Opción.....	8
Break .....	9
Paralelo .....	9

# DIAGRAMAS DE SECUENCIA

Muestran la **interacción** de un conjunto de **objetos** enfatizando el **orden en el tiempo** de los mensajes

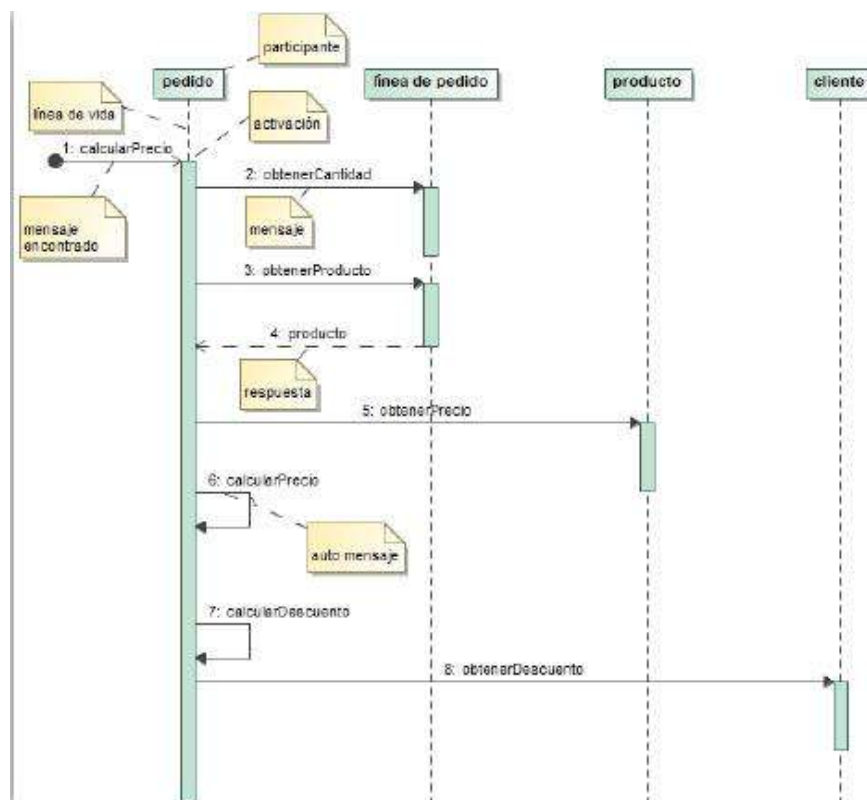
- las interacciones entre objetos en el orden secuencial en el cual las interacciones ocurren.

En la **fase de requisitos**, los **casos de usos** se refinan en uno o más diagramas de secuencia

- Contiene detalles de implementación del escenario
  - Objetos y clases usados para la implementación
  - Mensajes intercambiados entre los objetos

No están pensados para mostrar lógicas de procedimientos complejos.

*Ejemplo: Calcular precio de un pedido*



Tiempo progresa hacia abajo.

Los objetos involucrados en la operación aparecen de izquierda a derecha en función de cuándo toman parte en la secuencia de mensajes.

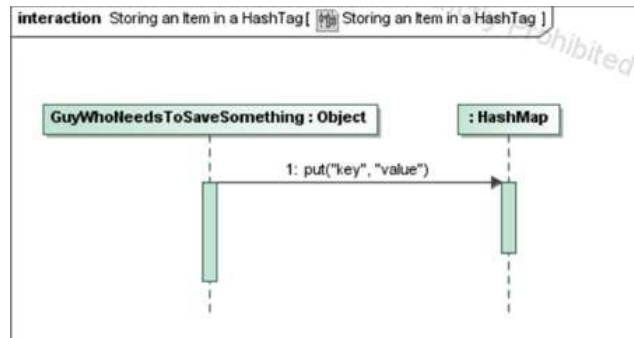
## Elementos de los Diagramas

### Participantes

Los **participantes** son instancias de clases (objetos)

Pueden tener nombre o ser anónimos

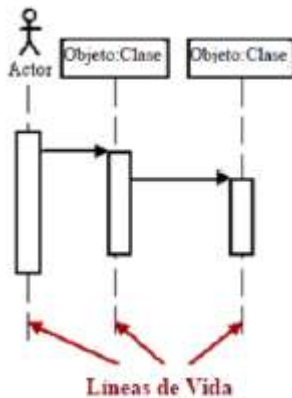
Ejemplo con dos participantes: uno llamado GuyWhoNeedsToSaveSomething y el otro sin nombre.



### Líneas de vida y Activación

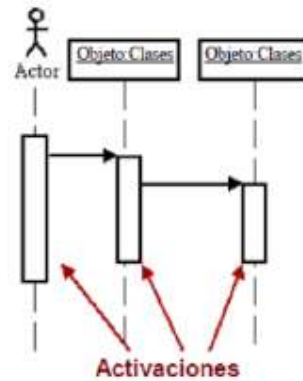
#### Líneas de vida

Indican la presencia del objeto en el tiempo.  
Los objetos pueden crearse/destruirse.



#### Activación

Representa cuándo el participante está activo en la interacción.

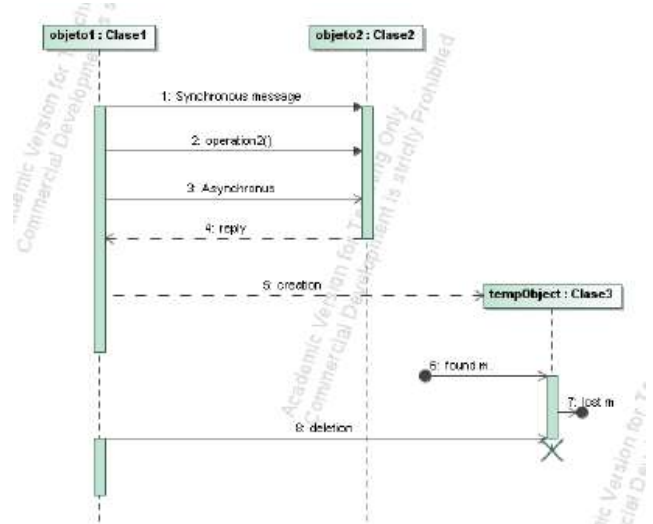


## Mensajes

Muestran una serie **interacciones** entre los participantes para llevar a cabo el proceso modelado.

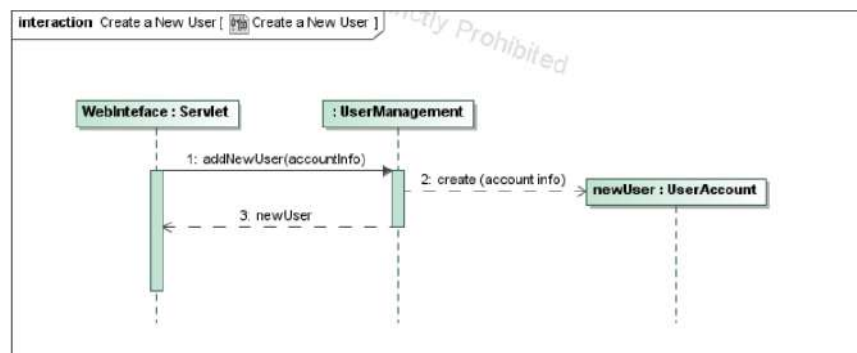
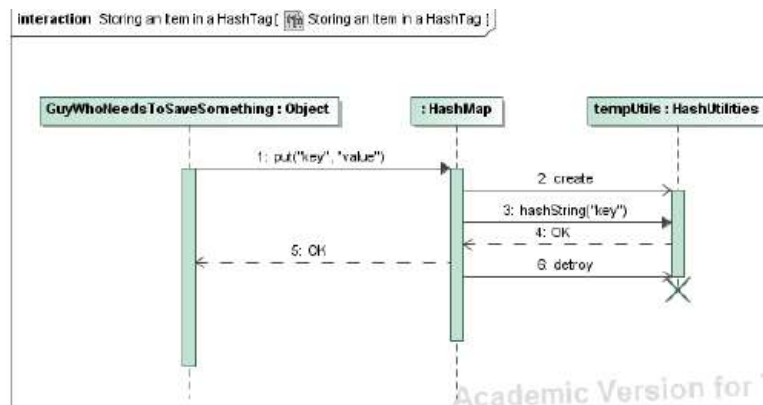
El primer mensaje de un diagrama de secuencia siempre se inicia en la parte superior.

El mensaje que se envía al objeto receptor representa una **operación** que la clase del objeto implementa.



## Creación y destrucción

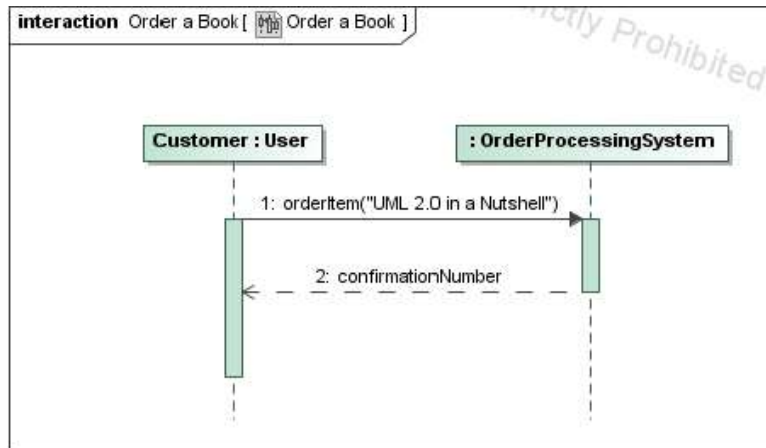
Los participantes (instancias) se pueden **crear y destruir de forma dinámica** durante la ejecución.



## Comunicación síncrona

El objeto que envía el mensaje queda bloqueado hasta que recibe la respuesta.

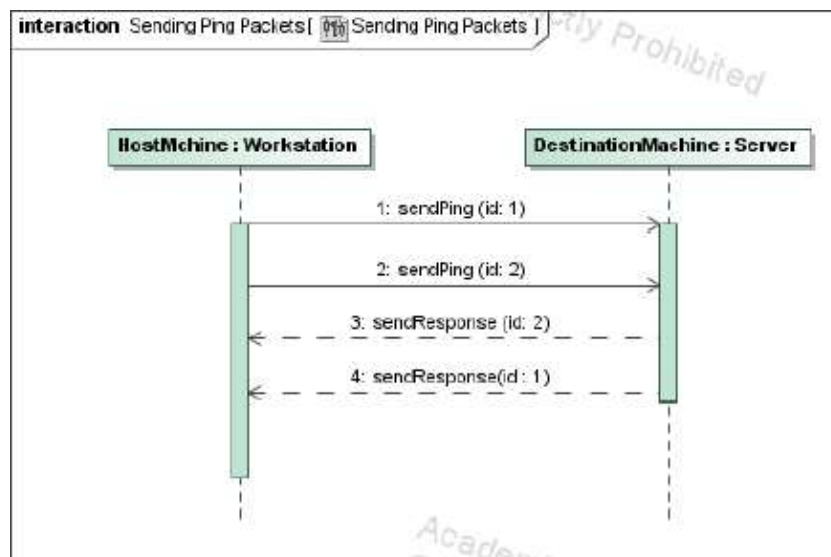
- Los mensajes se representan con flechas con la cabeza llena
- Los mensajes de respuesta se dibujan con línea discontinua



## Comunicación asíncrona

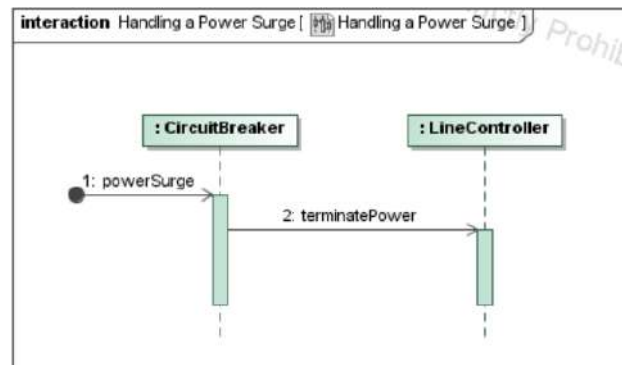
Los **mensajes asíncronos** terminan inmediatamente

- Crean un nuevo hilo de ejecución dentro de la secuencia
- Se representan con flechas con la cabeza abierta



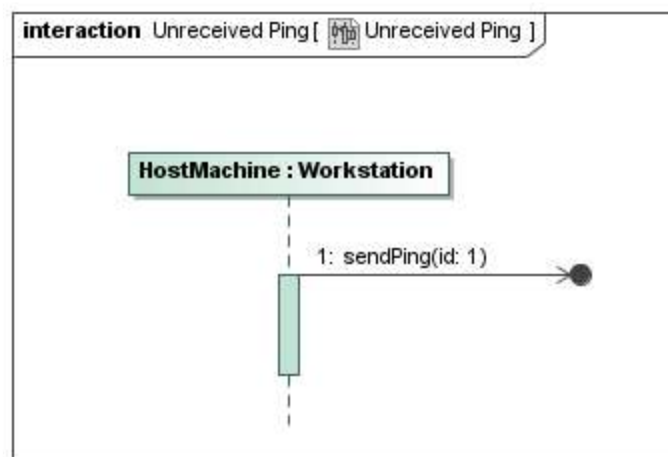
## Mensajes encontrados

Son aquellos cuyo origen no importa.



## Mensajes perdidos

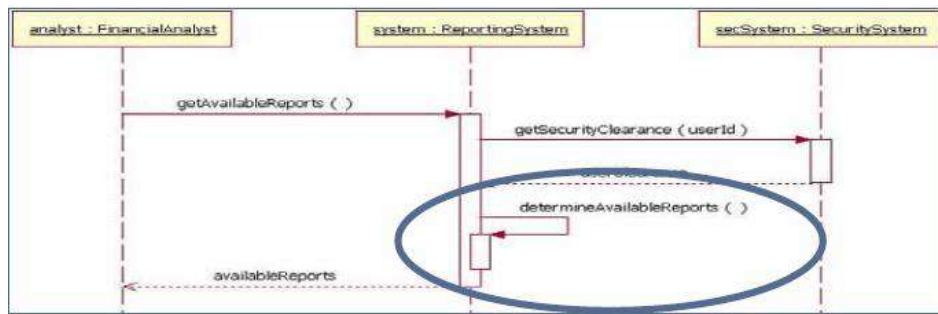
Son aquellos que no llegan a ningún participante.



## Auto-mensajes

Un objeto se manda mensaje a sí mismo. La flecha parte y termina en el mismo objeto.





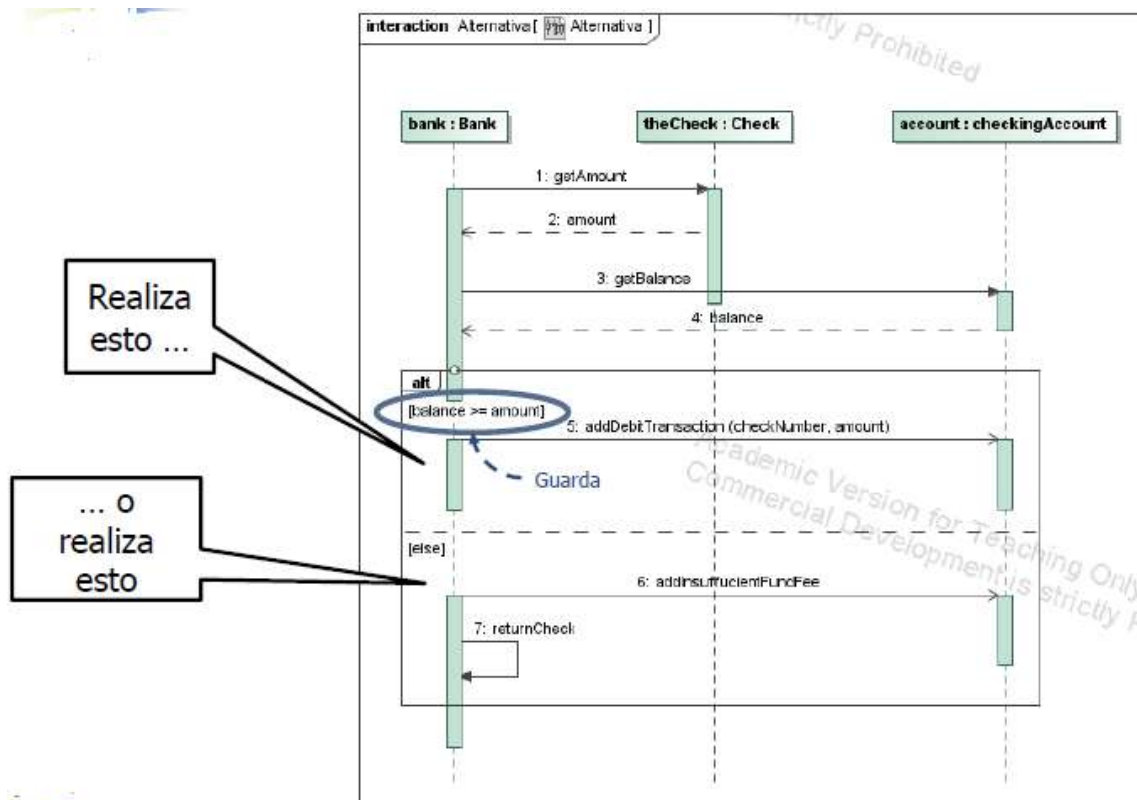
## Fragmentos combinados

Permiten definir una interacción compleja mediante componentes más simples.

- Alternativa
- Opción
- Paralelas
- Break
- Bucle
- Etc.

Usar con mesura: tienden a complicar el diagrama.

## Alternativas



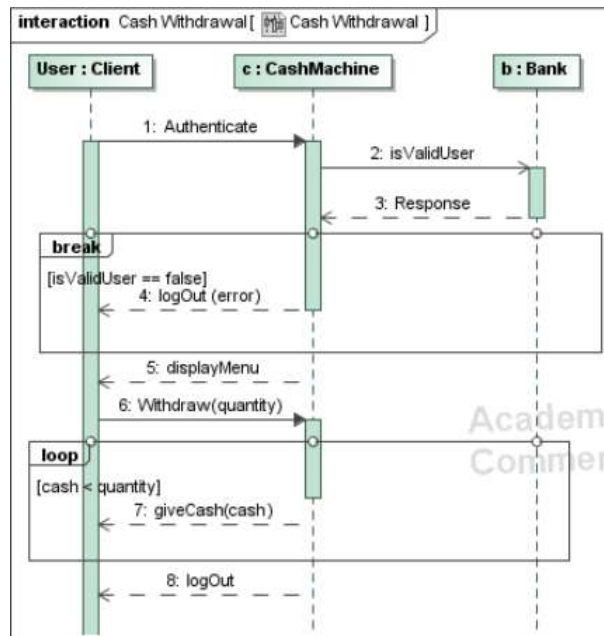
Formación  
Online  
Especializada

Clases Online  
Prácticas  
Becas

Escuela de  
LÍDERES

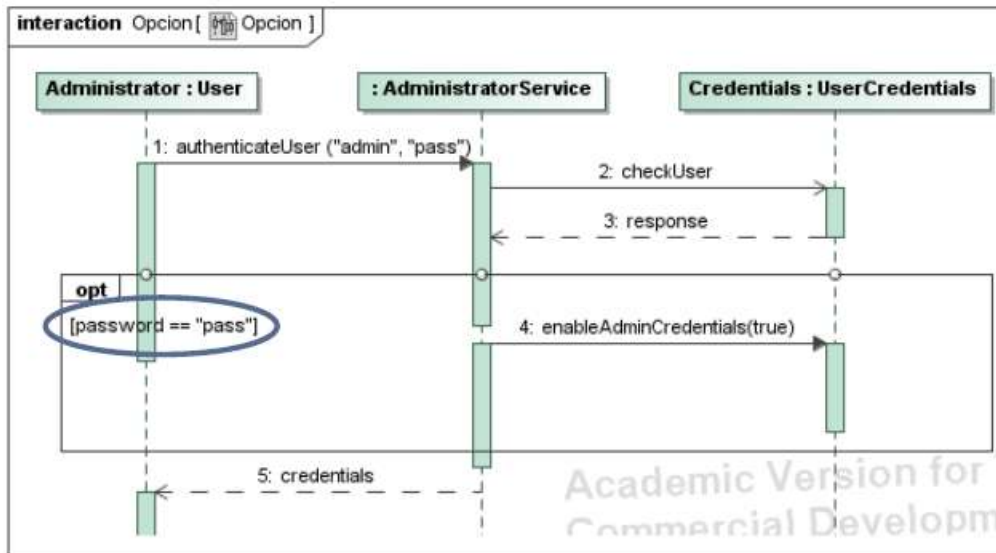
Jose María Girela  
Bim Manager.

## Bucles



## Opción

Ciertas acciones se ejecutan sólo si se cumple la condición



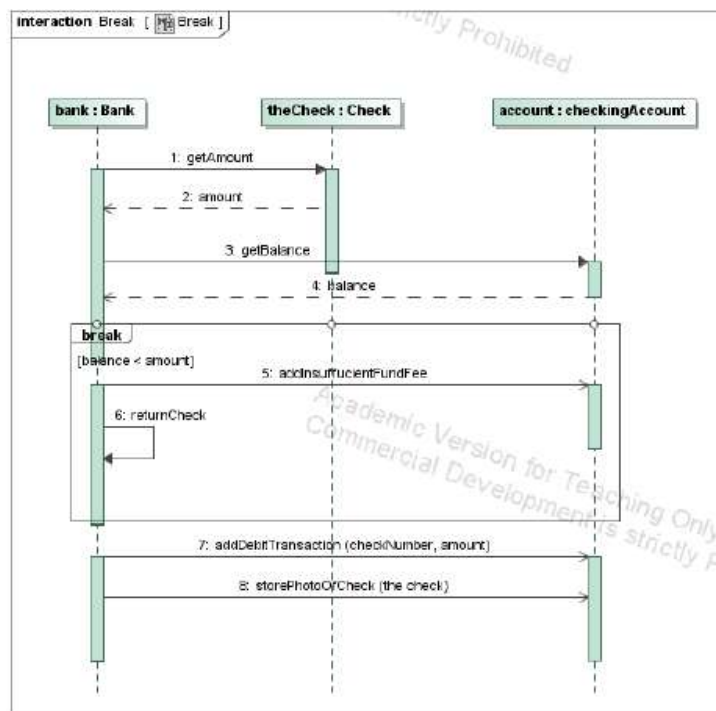
## Break

Similar la opción, pero lo que sigue no se ejecuta si entra en el break.

Ej.: Si [balance < amount]

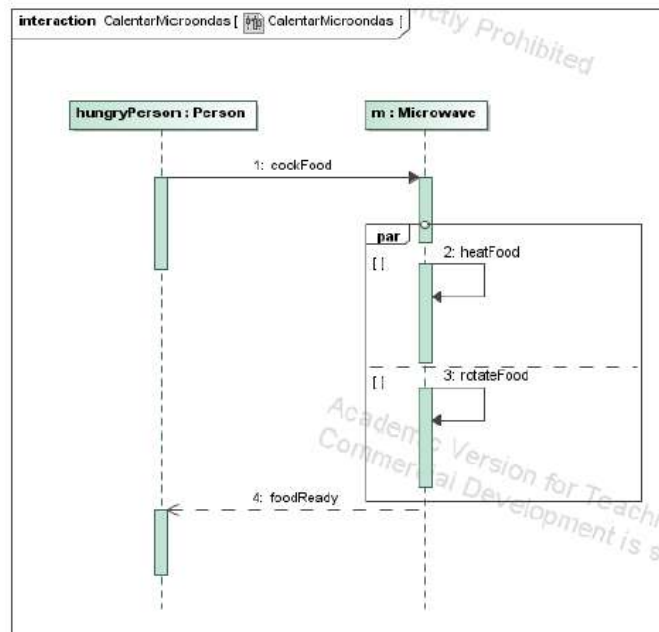
Se ejecutan mensajes del fragmento break.

Se detiene la ejecución del resto de interacciones.



## Paralelo

Especifica que varias acciones se realizan a la vez, cada una en un hilo de ejecución.



*Más ejemplos en transparencias.*

## Tema 6.1 – Arquitectura Software

Concepto de Arquitectura Software .....	2
Necesidad de la arquitectura software .....	2
Concepto .....	2
Importancia .....	2
Elementos de una arquitectura software .....	3
Estilo o patrón arquitectónico .....	3
Estilos de Arquitectura Software .....	4
Ejemplos de estilos arquitectónicos .....	4
Tuberías y filtros .....	4
MapReduce .....	6
Llamada y retorno funcional .....	8
Llamada y retorno orientada a objetos .....	10
Arquitectura de capas .....	11
Cliente servidor .....	12
Arquitectura Modelo-Vista-Controlador .....	15

---

# ARQUITECTURAS SOFTWARE

---

## Concepto de Arquitectura Software

### Necesidad de la arquitectura software

*"A medida que el tamaño y la complejidad de los sistemas de software aumenta, el problema del diseño va más allá de los algoritmos y las estructuras de datos de la implementación: el diseño y la especificación de la estructura general del sistema surge como un nuevo tipo de problema ... Este es la arquitectura de software a nivel de diseño."*

### Concepto

**Arquitectura software:** descripción de los subsistemas de un sistema software, sus propiedades y las relaciones entre ellos.

#### Definiciones alternativas:

*"Estructura de los componentes de un programa o sistema, sus interrelaciones, y los principios y reglas que gobiernan su diseño y evolución en el tiempo."* Garlan y Perry, 1995.

*"Estructura o estructuras de un sistema, lo que incluye sus componentes de software, las propiedades observables de dichos componentes y las relaciones entre ellos."* Bass, Clements y Kazman, 1998.

*"Para mí el término arquitectura conlleva la noción de los elementos que forman el núcleo del sistema, las piezas que son difíciles de cambiar. Los cimientos sobre los que se edifica el resto."* Martin Fowler, 2001.

### Importancia

¿Por qué es necesario desarrollar un modelo arquitectónico?

- Facilita la comprensión a cualquier miembro del equipo.
- Permite que cada miembro pueda trabajar en los subsistemas de forma individual.
- Prepara al sistema para su extensión.
- Facilita la reutilización del sistema.

# Elementos de una arquitectura software

## Componentes

- Elementos computacionales donde se realiza el “trabajo” (subsistemas software)
- Pueden ser de grano grueso (e.g. un servidor web) o fino (e.g. un módulo)

## Conectores

- Comunican componentes
- Pueden ser explícitos (e.g. invocación) o implícitos (e.g. evento)

## Configuración

- Disposición concreta de los componentes y conectores que forman una arquitectura software

# Estilo o patrón arquitectónico

Expresa la **organización estructural** para sistemas software.

Cada **estilo arquitectónico define la organización general** en base a:

- El tipo de componentes y sus responsabilidades
  - capas, servidores, bases de datos, componentes software...
- El tipo de conectores
  - tuberías, invocaciones, difusión de eventos...
- La configuración adecuada
  - Cómo se conectan los componentes entre sí mediante conectores

Decidir el estilo arquitectónico apropiado para un sistema no es trivial.

Ya puedes sacarte tu B1/B2/C1 de inglés desde casa

#LinguaskillEnCasa



## Estilos de Arquitectura Software

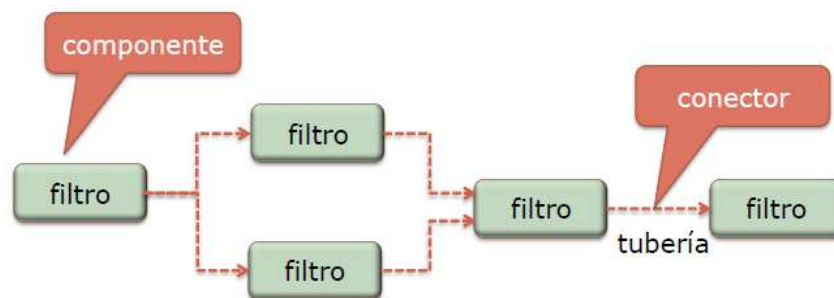
### Ejemplos de estilos arquitectónicos

- Basados en flujos de datos
  - Tuberías y filtros
  - MapReduce
- Basados en llamada y retorno
  - Funcional
  - Orientado a objetos
- Arquitectura de capas
- Cliente-Servidor
- Modelo-Vista-Controlador

### Tuberías y filtros

Dirigido por el flujo de datos (dataflow)

- Los datos se procesan incrementalmente conforme llegan.
- Se puede generar la salida antes de consumir toda la entrada.



#### Componentes: filtros

- Leen flujos de entrada
- Transforman localmente los datos
- Escriben flujos de salida
- Son independientes
  - no tienen estado compartido
  - se desconocen entre sí

#### Conectores: tuberías

- Flujos de datos (e.g. buffer FIFO, socket, etc.)



```
$ ls -l | grep "Aug"

-rw-rw-rw- 1 john doc 11008 Aug 6 14:10 ch02
-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07
-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro
-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros
```

El comando "ls" muestra en formato largo los archivos contenidos en una dirección determinada y a continuación, con esa lista el comando "grep" filtra los que tienen coincidencia con "Aug"

- \$ ps -ef | grep httpd | wc -l

727

El comando "ps" lista los procesos activos del sistema operativo, "grep" selecciona aquellos con la coincidencia "httpd" y cuenta las líneas de estos ficheros, devolviendo ese valor.

- Composición en lenguajes funcionales (p.ej. Haskell)
- Cauce de un procesador, fases de un compilador, etc.
- Lenguajes de programación multimedia (p.ej. PureData, Max, etc.)

## *Ventajas de tuberías y filtros*

**Semántica composicional:** el comportamiento global es la composición de los comportamientos de los filtros.

**Reutilización:** dos filtros existentes se pueden conectar si soportan el tipo de datos adecuado.

**Mantenimiento:** sustitución de filtros por otros (más eficientes, ...).

**Ejecución paralela o distribuida.**

## *Inconvenientes de tuberías y filtros*

No es apropiado para **procesamiento interactivo**, es decir, con participación del usuario.

Coste de la transformación de los datos para leerlos y escribirlos en las tuberías.

**Gestión de errores complicada:** ¿qué pasa si falla un filtro intermedio?

Compartir un **estado global** es difícil.

# MapReduce

Introducido por Google para el **procesamiento distribuido masivo de datos (Big Data) sobre clusters**.

- Implementaciones libres alternativas (Hadoop)

Inspirado en programación funcional:

**map**: aplica una función sobre todos los componentes de una lista

**reduce**: extrae un resultado con los componentes de una lista

API disponible en varios lenguajes

- API oculta detalles complicados (balanceo de carga, paralelización automática, gestión de errores, etc.)

**Paralelismo masivo**: cluster de Yahoo! Para ejecutar Hadoop.

## *Idea clave de MapReduce*

**Problema**: contar las apariciones de una palabra en un conjunto de documentos

Solución con tuberías y filtros

```
$ cat *.docs | grep palabra | wc -l
```

Concatenar documentos, seleccionar la palabra y contarlas

Idea clave: grep se puede **paralelizar**

- aplica grep en paralelo:  
**counts = map (grep palabra) \*.docs**
- acumula resultados:  
**reduce (+) 0 counts**

## Esquema de MapReduce

MapReduce trabaja con pares **(key,value)**

1.  $\text{map: } (k, v) \mapsto [(k', v')]$

El par de entrada dará lugar a una lista de pares

2. Las salidas de los map se agrupan por claves ( $k'$ ,  $[v']$ ) y se pasan a reduce

3.  $\text{reduce: } (k', [v']) \mapsto [v'']$

El par (clave, lista) generará una nueva lista

El esquema es estable, para cada problema se decide qué hacen map y reduce.

## Diagrama de MapReduce

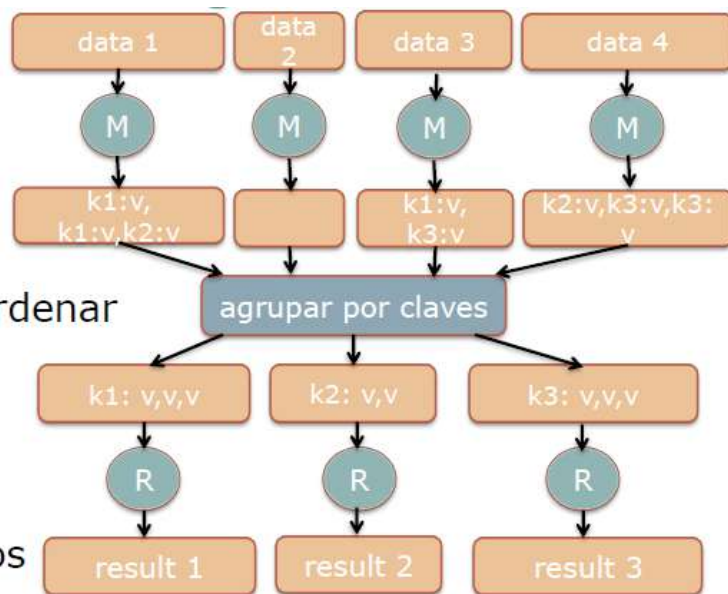
1. leer datos

2. **map**

3. agrupar y ordenar

4. **reduce**

5. escribir datos

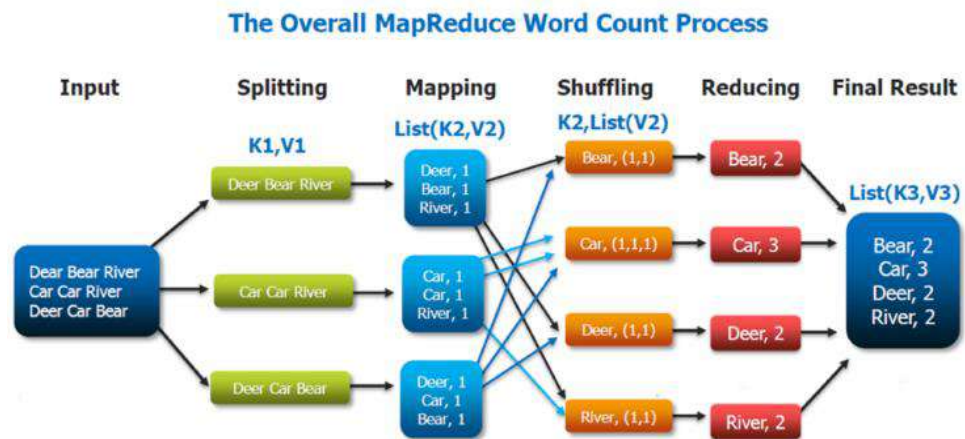


Ya puedes sacarte tu B1/B2/C1 de inglés desde casa



#LinguaskillEnCasa

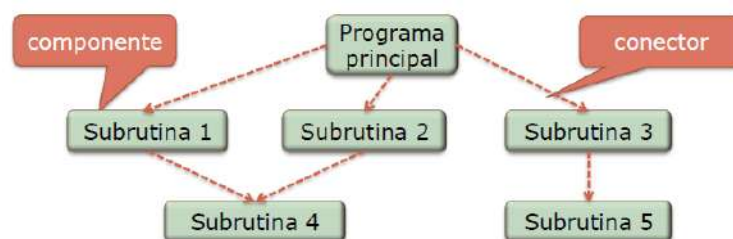
Ejemplo



Llamada y retorno funcional

Basado en la descomposición funcional del sistema

- Las subrutinas corresponden a las tareas a realizar
- Se combinan según el interfaz y el flujo de control



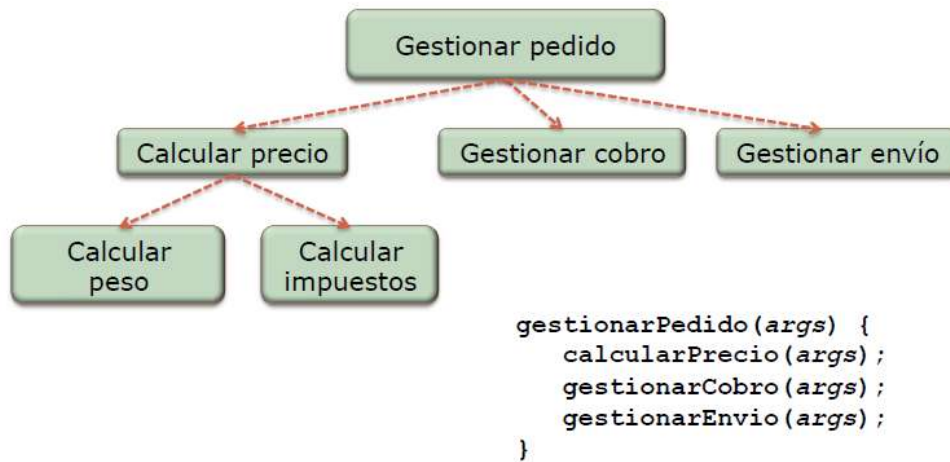
**Componentes: subrutinas**

- Implementan tareas o subtareas
- Ofrecen una interfaz al exterior (argumentos)
- Combinadas a través del flujo de control

**Conectores: llamadas**

- De acuerdo con la interfaz (argumentos)

## Ejemplo



## Ventajas e inconvenientes

### Ventajas:

- Se basa en partes bien identificadas de la tarea a realizar
- Se puede cambiar una subrutina sin que afecte a los clientes
- Se pueden reutilizar operaciones concretas

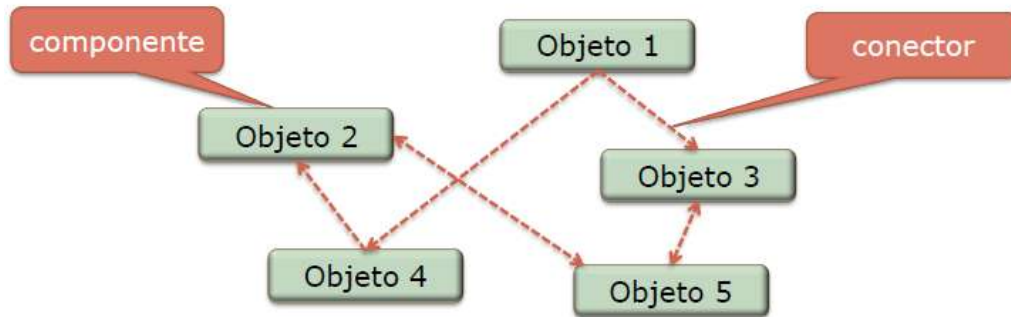
### Inconvenientes:

- Enfoque operacional que oculta el papel de los datos
- Dependencias entre subrutinas
- Es difícil de extender y adaptar a nuevas situaciones

## Llamada y retorno orientada a objetos

Basado en la descomposición en objetos

- Los objetos modelan entidades reales (diseño antropomórfico)
- Colaboran intercambiando mensajes



### Componentes: objetos

- Representan una entidad del dominio del problema (análisis) o de la solución (diseño)
- Encapsulan un estado privado y mantienen su integridad
- Exhiben un comportamiento público (interfaz)

### Conectores: mensajes

- Se pueden resolver en tiempo de ejecución (vinculación dinámica)

## Ventajas e inconvenientes

### Ventajas:

- Se reducen las dependencias.
- Se facilitan la reutilización, extensión y el mantenimiento.
- Se puede modificar la implementación sin afectar a los clientes.
- Se puede distribuir sobre varias máquinas o redes.

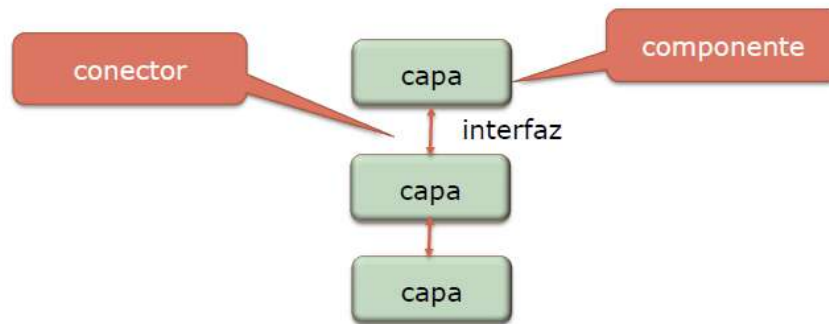
### Inconvenientes:

- Los objetos deben conocerse para cooperar.
- Efectos secundarios de la colaboración.
  - Si los objetos A y B colaboran con un objeto C es posible que los efectos de la colaboración entre A y C afecten a la colaboración con B o viceversa.

# Arquitectura de capas

Organizado en niveles de abstracción (capas)

- Cada capa se comunica exclusivamente con las adyacentes
- Las **interfaces** entre capas están claramente definidas



## Componentes: capas

- Grupo de tareas (servicios) que implementan un nivel de abstracción
- Presta servicios a la capa inmediatamente superior
- Demanda servicios de la capa inmediatamente inferior
- Son completamente independientes de las capas superiores
- Capas más bajas proporcionan servicios de bajo nivel (ej. protocolos de comunicación, acceso a datos, etc.).

## Conectores: interfaces

- Protocolos que definen la interacción entre capas (oferta de servicios)

## Ejemplos

- Protocolos de comunicación (modelo OSI)



- Sistemas operativos (Android)



Ya puedes sacarte tu B1/B2/C1 de inglés desde casa

#LinguaskillEnCasa



### *Ventajas e inconvenientes*

#### **Ventajas:**

- Nivel de abstracción creciente conforme ascendemos por las capas.
- **Reutilización:** preservando la interfaz, una capa se puede reemplazar por otra.
- **Mantenimiento:** el cambio de las interfaces de las capas sólo afecta a la capa adyacente (cambios localizados)

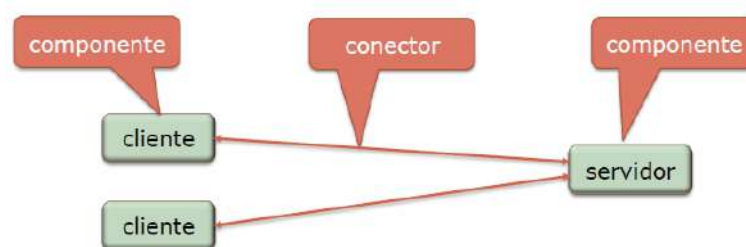
#### **Inconvenientes:**

- Puede ser difícil identificar las capas.
- **Rendimiento:** descenso y ascenso a través de las capas
  - Es tentador acceder a capas no adyacentes

### Cliente servidor

Modelo de sistema distribuido que muestra cómo datos y procesamiento se distribuyen a lo largo de varios procesadores

- Centraliza la gestión de la información
- Protocolo de petición-respuesta





### Componentes: servidor y clientes

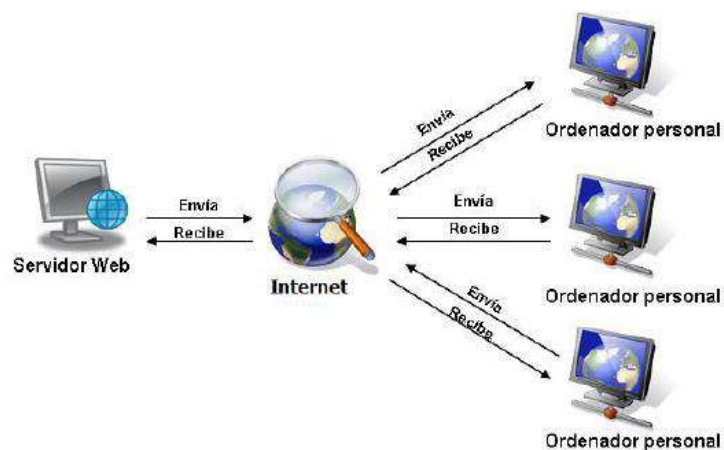
- El servidor oferta servicios
  - Gran capacidad de cómputo o almacenamiento.
  - Ej. Servidor de impresión, almacenamiento, Web, etc.
- Los clientes demandan servicios
  - **Ciente delgado:** poco procesamiento, se centra en la interfaz
  - **Ciente grueso:** tanto procesamiento como sea posible, el servidor almacena y comunica datos

### Conectores: flujos de datos

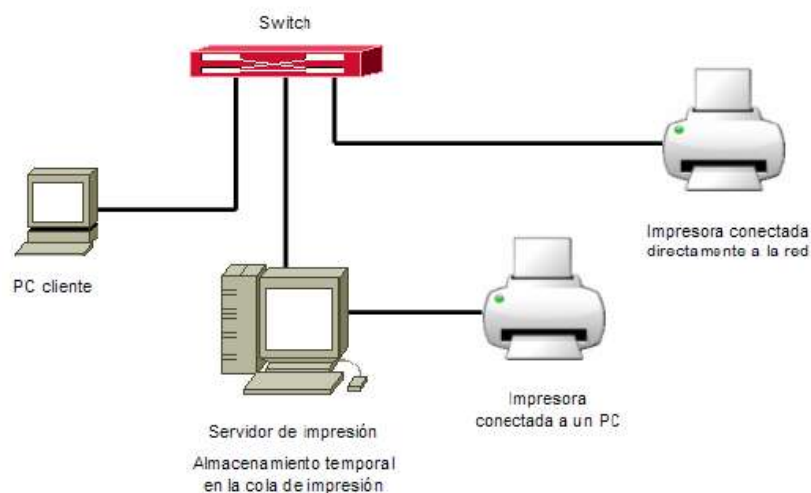
- Remotos: a través de red
- Locales: cliente y servidor residen en la misma máquina

### Ejemplo

Servicio web:



Servidor de impresión



## *Ventajas*

### **Control centralizado**

- Facilita el mantenimiento
- Refuerza seguridad

### **Escalabilidad**

- Es fácil aumentar la capacidad del servidor o los clientes

### **Uso racional del presupuesto hardware**

- Concentra el gasto en el servidor

## *Inconvenientes*

### **Congestión del tráfico**

- Cuello de botella, denegación de servicio, etc.

### **Robustez**

- ¿Qué pasa si falla el servidor?

### **Coste de la transformación de los datos**

- Cada cliente puede utilizar un formato distinto

# Arquitectura Modelo-Vista-Controlador

**Modelo-Vista-Controlador (MVC):** ayuda a separar la capa de interfaz de usuario de otras partes del sistema.

**Adecuada cuando:**

- Aplicaciones con interfaces de usuario interactivas.
- Hay múltiples formas de mostrar o interactuar con los datos.
- Los requisitos de la interacción y la presentación son desconocidos.

**Idea clave:** separar el modelo de negocio y la interfaz.

- **Modelo de negocio:** datos y lógica de negocio que implementa la funcionalidad básica de la aplicación.
- **Interfaz:** interacción con el usuario.
- Se intenta no contaminar el modelo con la interfaz.

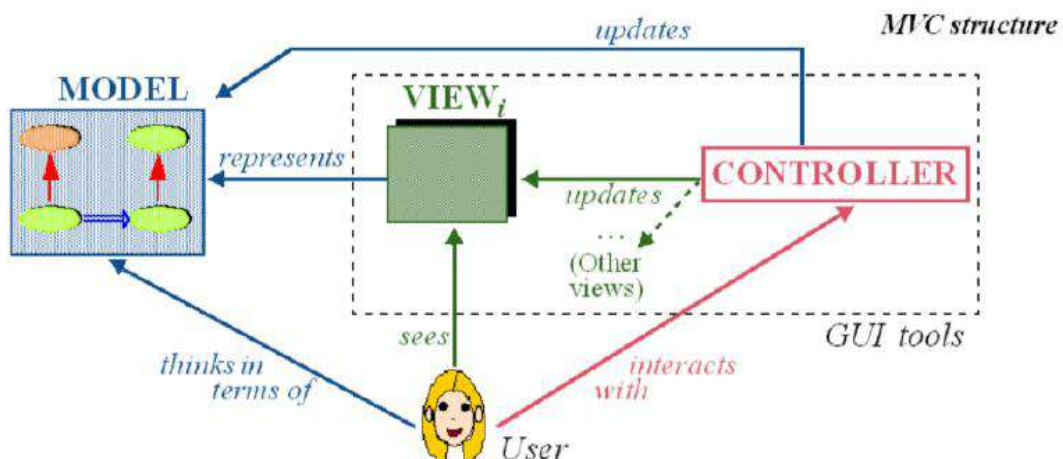
Tres tipos de componentes:

- **Modelo:** responsable de gestionar los datos del sistema y las operaciones asociadas a los datos.
- **Vista:** define y gestiona cómo los datos se muestran al usuario (representación visual del modelo).
- **Controlador:** responsable de la interacción con el usuario (p.ej. clicks de ratón, teclas pulsadas) y de pasar esas interacciones a la Vista y al Modelo.

A un modelo le pueden corresponder múltiples vistas y controladores

Interfaz = Vista + Controlador.

Controlador y vista están muy relacionados.



Ya puedes sacarte tu B1/B2/C1 de inglés desde casa



### *El modelo*

- Representa el problema a resolver.
- Encapsula el estado de la aplicación.
- Contiene los datos y la lógica del negocio.
- No contiene operaciones de entrada/salida.
- Es independiente de la vista y el controlador. No contiene referencias ni a la vista ni al controlador, pero facilita métodos para que estos lo usen.

### *La vista*

- Ofrece una visión total o parcial del estado del modelo.
- Contiene controles interactivos que lanzan eventos, pero no los maneja.
- No debe suponer el estado del modelo, debe reflejarlo fielmente. No utiliza estados almacenados.
- Es un observador pasivo (no afecta al modelo).

### *El controlador*

- Gestiona la interacción usuario-modelo.
- Responde a los eventos de la vista, indicando qué acciones debe ejecutar el modelo y cómo actualizar la vista.
  - Traduce los eventos de la vista a acciones sobre el modelo y actualizaciones de la vista.
- Contiene referencias al modelo y la vista.

### *Ejemplo*

APIs gráficas:

- Spring Boot
- Objective C

La mayoría de frameworks Web:

- Django
- Ruby on Rails

## *Ventajas e inconvenientes*

### **Ventajas:**

#### Robustez:

- Permite que los data cambien con independencia de su representación y viceversa.

#### Flexibilidad

- Los mismos datos se pueden presentar de múltiples formas
- Se pueden reutilizar componentes.

#### Mantenimiento

- Desarrollo, prueba y mantenimiento por separado

### **Inconvenientes:**

#### Sobrecarga extra:

- Puede requerir código adicional (o más complejidad) para modelos de datos o interacciones simples.

## Tema 6.2 – Diseño Software

Conceptos de Diseño Software .....	2
Concepto de diseño .....	2
Propiedades de diseño.....	2
Espacio y decisiones de diseño .....	3
Tomando decisiones de diseño.....	3
Principios de Diseño.....	4
Objetivos.....	4
Principios.....	4
Principios de diseño Orientado a Objetos .....	6
Principios <b>SOLID</b> .....	6
Principio de Responsabilidad Única (SRP).....	6
Principio abierto-cerrado (OCP).....	7
Principio de sustitución de Liskov .....	9
Principio de segregación de interfaces .....	11
Principio de inversión de dependencias .....	13
Patrones de Diseño .....	14
Motivación .....	14
Patrón .....	14
Refactorización .....	25
Concepto de refactorización.....	25
Ventajas de refactorización .....	25
¿Cuándo refactorizar?.....	26
Limitaciones de la refactorización .....	26
Candidatos a la refactorización.....	26
Un catálogo de refactorizaciones .....	27

---

# DISEÑO SOFTWARE

---

## Conceptos de Diseño Software

### Concepto de diseño

El **diseño** es un proceso de **resolución de problemas** cuyo objetivo es encontrar y describir una manera de:

- Implementar los requisitos funcionales (servicios que se esperan del sistema)
- Respetando las restricciones impuestas por los requisitos de calidad, plataforma y proceso software
- Ajustado al presupuesto y plazo de ejecución
- Siguiendo principios generales de calidad

**Diseño** es el proceso creativo de transformar un **problema** en una **solución**

- La descripción de una solución también se conoce como diseño

La fase de **análisis** plantea el problema (**qué**).

La fase de **diseño** especifica una solución particular para el problema (**cómo**).

### Propiedades de diseño

El diseño es un proceso de refinamiento que implica una **propuesta de solución**.

- Integra el paso del ¿Qué? al ¿cómo?

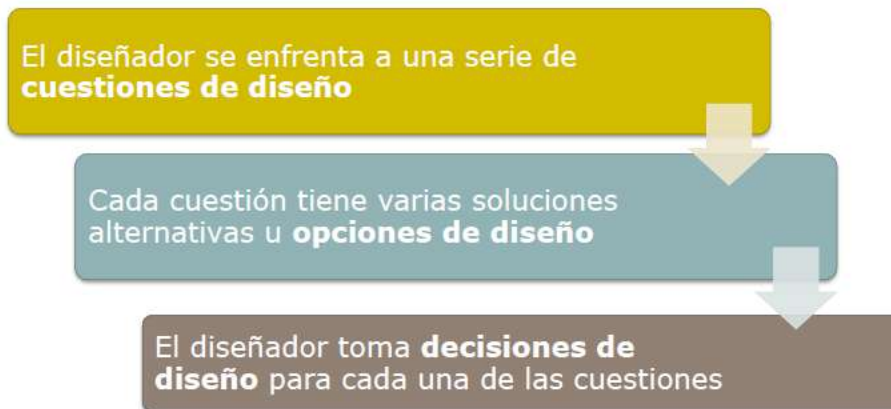
Es una **actividad creativa**, apoyada por principios, técnicas, herramientas...

Requiere adoptar **decisiones de diseño** explícitas.

- Éstas luego podrán ser corregidas o modificadas

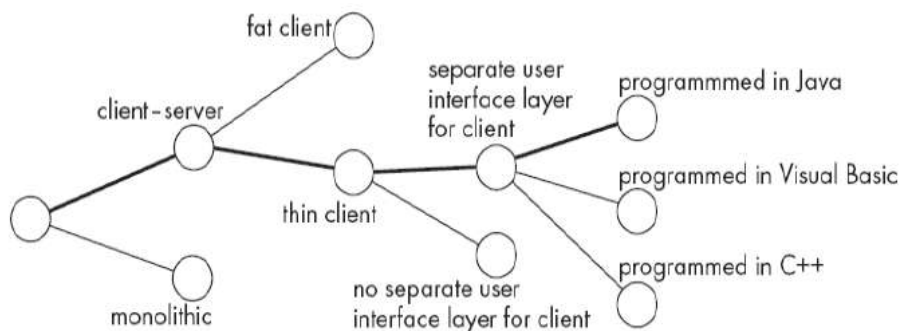
Es clave para la **calidad** posterior del software.

## Espacio y decisiones de diseño



**Espacio de diseño:** es el conjunto de diseños que se puede obtener según las diferentes opciones de diseño.

### *Ejemplo simple de espacio de diseño*



## Tomando decisiones de diseño

Para tomar una decisión de diseño nos guiamos por:

- Los requisitos
- La parte del diseño ya creada
- La tecnología disponible
- Los principios de diseño
- La experiencia



Ya puedes sacarte tu B1/B2/C1 de inglés desde casa

#LinguaskillEnCasa



## Principios de Diseño

### Objetivos

- **Satisfacer los requisitos** (funcionales y no funcionales)
- Asegurar la **calidad del software**:
  - Modelo de Calidad del Producto Software ISO/IEC 25010 establece el sistema para la evaluación de la calidad de un producto software
- Ajustarse al **presupuesto y plazo de entrega**.

### Principios

Principios genéricos que conducen a un buen diseño software:

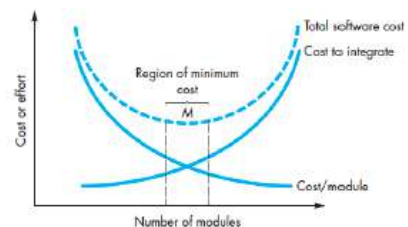
- **Modularidad**: divide y vencerás
- **Aumentar la cohesión**: agrupar las unidades software que están relacionadas.
  - Dicha unidad será mas sencilla de diseñar, implementar, probar y mantener.
- **Reducir el acoplamiento**: grado de relación de un módulo con los demás.
  - A menor acoplamiento el módulo será más sencillo diseñar, implementar, probar y mantener.

### *Modularidad: divide y vencerás*

El **sistema se divide en componentes** identificables y tratables por separado que se integran para satisfacer los requisitos del sistema.

#### Ventajas:

- Cada componente es más pequeño que el total, más fácil de entender y diseñar
- Los componentes se pueden cambiar o reemplazar
- Diferentes personas pueden trabajar en diferentes partes

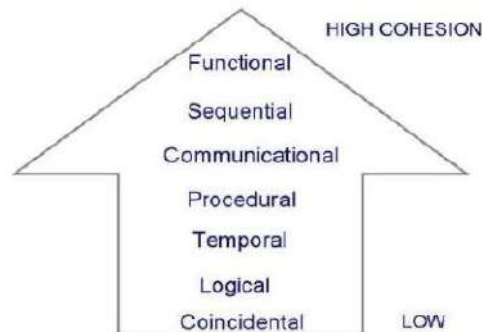


**Problema:** Llegar a un balance en el nº de módulos

## Aumentar la cohesión

**Cohesión:** medida del **grado de relación** de los **contenidos** de un componente entre sí

Existen varios **grados de cohesión**



## Tipos de cohesión

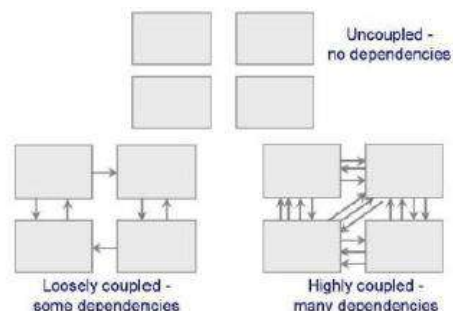
- **Cohesión funcional:** se agrupan elementos que desarrollan una única función sin efectos colaterales (p.ej. funciones matemáticas, ordenamiento de arrays).
- **Cohesión secuencial:** un módulo realiza distintas tareas en secuencia, de forma que las entradas de cada tarea son las salidas de la tarea anterior.
- **Cohesión comunicacional:** se agrupan los elementos que operan sobre los mismos tipos de datos. Un buen diseño de clases tiene este tipo de cohesión.
- **Cohesión procedimental:** similar a la secuencial, pero la salida de una tarea no tiene por qué ser la entrada a la siguiente. En este caso se agrupan tareas diferentes y posiblemente no relacionadas (p.ej. 1er año universidad: matricularse en universidad, buscar piso, salir de fiesta, etc.).
- **Cohesión temporal:** las operaciones que se realizan durante la misma fase de la ejecución del programa se mantienen juntas (p.ej. agrupación de operaciones de inicialización o terminación).
- **Cohesión lógica:** distintas operaciones se agrupan porque están relacionadas de forma lógica, esto es, realizan acciones parecidas (p.ej. agrupar rutinas para gestión del teclado y/o ratón).
- **Cohesión casual o coincidente:** las operaciones se agrupan de forma arbitraria (p.ej. clase de utilidades o dividir programa en N subrutinas de igual longitud).

## Reducir acoplamiento

**Acoplamiento:** medida del **grado de dependencia** entre los **componentes** de un sistema

Sistemas muy acoplados son difíciles de probar, implementar y mantener.

Hay varios **grados de acoplamiento**



## Tipos de acoplamiento

- **Acoplamiento normal:** una unidad de software llama a otra de un nivel inferior y tan solo intercambian datos (p.ej. parámetros de entrada / salida).
- **Acoplamiento externo:** las unidades de software están ligadas a componentes externos (p.ej. dispositivos de entrada / salida, librerías comunes, protocolos de comunicaciones, etc.)
- **Acoplamiento común:** dos unidades de software acceden a un mismo recurso común, generalmente memoria compartida, una variable global o un fichero.
- **Acoplamiento de contenido:** ocurre cuando una unidad de software necesita acceder a una *parte interna* de otra unidad de software.

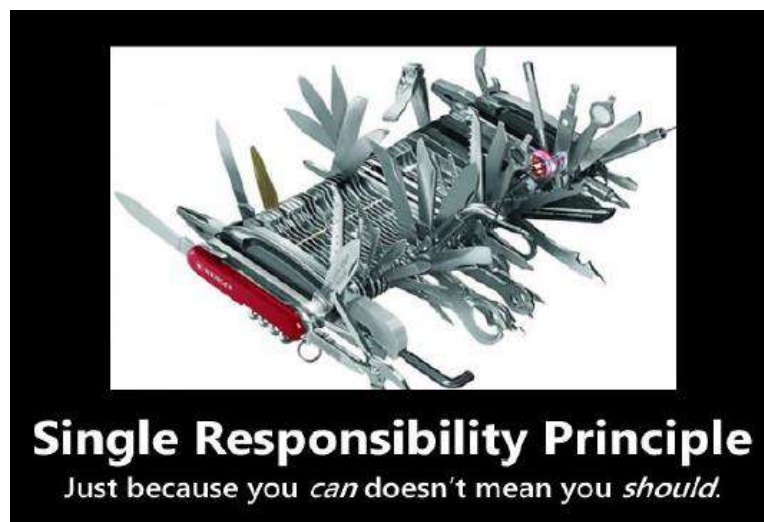
## Principios de diseño Orientado a Objetos

### Principios SOLID

Acrónimo acuñado por Robert C. Martin para resumir los 5 principios básicos del Diseño Orientado a Objetos:

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OPC)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

### Principio de Responsabilidad Única (SRP)



*“Una clase debería tener un solo motivo para cambiar”*

Robert C. Martin

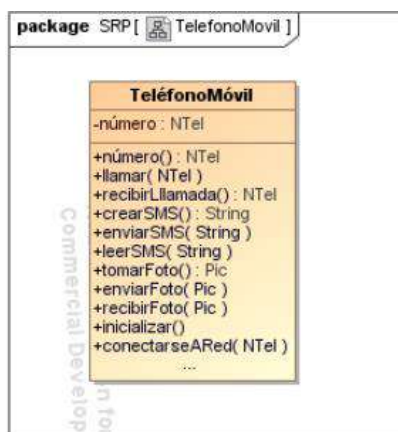
*“Una clase debe hacer una cosa y hacerla bien”*

Responsabilidad es la razón para cambiar.

Reinterpreta la **cohesión** desde el punto de vista de los motivos que fuerzan los cambios en una clase:

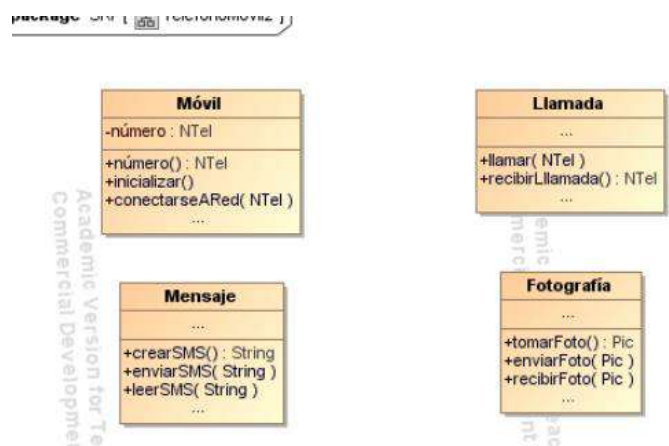
- Los servicios de la clase se deben alinear para proveer esa responsabilidad.
- Si una clase asume más de una responsabilidad: más sensible al cambio y las responsabilidades se acoplan.

### Violación de SRP



Clase que asume múltiples responsabilidades

### Aplicación de SRP



Clases donde cada una asume una responsabilidad

### Principio abierto-cerrado (OCP)



Ya puedes sacarte tu B1/B2/C1 de inglés desde casa

#LinguaskillEnCasa

*“Las clases deben estar abiertas a la extensión y cerradas a la modificación”*

Bertrand Meyer

**Adaptarse a nuevas situaciones:**

- sin modificar el código existente que funciona bien (cerrado para modificación)
- añadiendo nuevo código (abierto para extensión)

**Se puede conseguir mediante polimorfismo:**

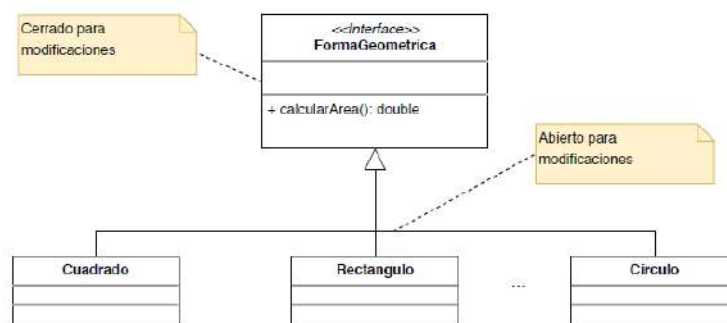
- **Dinámico:** sin especificación del tipo de datos (herencia, vinculación dinámica)
- **Estático:** tipos de datos explícitos (genericidad, plantillas)

*Violación de OCP*

```
switch (p.tipo) {  
    case "cuadrado":  
        a = area_cuadrado(p); break;  
    case "circulo":  
        a = area_circulo(p); break;  
    ...  
    default :  
        throw new Exception("No soportado");  
}
```

**Problema:** es imposible añadir un nuevo tipo de cuenta sin modificar el código.

*Aplicación de OCP*



Cambiar no es modificar lo que ya funciona, cambiar es extender

## Principio de sustitución de Liskov



*“Las subclases deben poder sustituir a las clases base sin que el código cliente lo note”*

Barbara Liskov

- *es-un = puede-sustituir-a*
- Usar bien la herencia
- Sustituir sin comprometer propiedades deseables
- Asegurar la interoperabilidad semántica de los subtipos
- Clases derivadas deben respetar los contratos (**precondiciones y postcondiciones**) definidas en las clases base.

### Violación de LSP

Un cuadrado es un rectángulo



La clase Cuadrado hereda de Rectángulo, pero no puede sustituir a esta

Un cuadrado es un rectángulo si altura y anchura coinciden:

```
class Cuadrado extends Rectangulo {  
    public int setAnchura(int a) {  
        anchura = altura = a;  
    }  
    public int setAltura(int a) {  
        anchura = altura = a;  
    }  
}
```

Un cuadrado no puede sustituir a un rectángulo: el cliente nota la diferencia

```
int pruebaArea(Rectangulo r) {  
    r.setAnchura(2);  
    r.setAltura(5);  
    assert(r.area() == 2*5);  
}
```

### *Relación entre OCP y LSP*

“Arreglo” mediante violación de OCP:

```
int pruebaArea(Rectangulo r) {  
    if (r instanceof Rectangulo) {  
        r.setAnchura(2);  
        r.setAltura(5);  
        assert(r.área() == 2*5);  
    }  
}
```

Violación de LSP = Violación latente de OCP.

El no cumplimiento de LSP conlleva el no cumplimiento de OCP.

## *Síntomas de la violación de LSP*

- **Ocultar el comportamiento heredado:**
  - Redefiniendo métodos con implementaciones vacías o lanzando excepciones indicando “no soportado”
- **Modificar el comportamiento heredado:**
  - Redefiniendo métodos con semánticas incompatibles
  - Lanzando excepciones nuevas, inesperadas por cliente
- **Modificar la clase base:**
  - Para adaptarla a las necesidades de la heredera
- Si redefines más que extiendes, deberías usar delegación, agregación o composición

## Principio de segregación de interfaces



*“Los clientes no deben depender de métodos que no utilizan”*

*Robert C. Martin*

- Algunas clases tienen **interfaces amplias**
- Los clientes sólo requieren algunos servicios
- Algunos clientes hacen crecer la interfaz
- Agrupar los clientes según necesidades y segregar la interfaz en función de los clientes

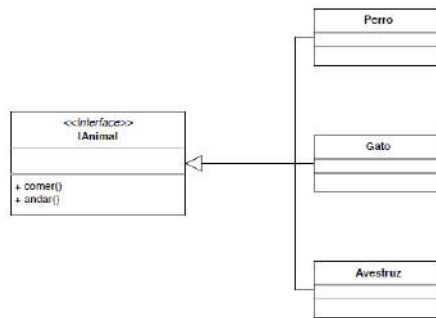


Ya puedes sacarte tu B1/B2/C1 de inglés desde casa

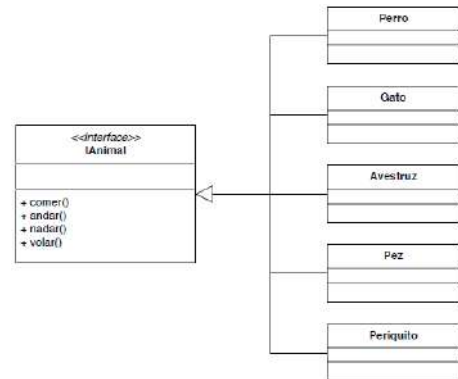
#LinguaskillEnCasa

### Violación de ISP

Todos los clientes usan la misma interfaz

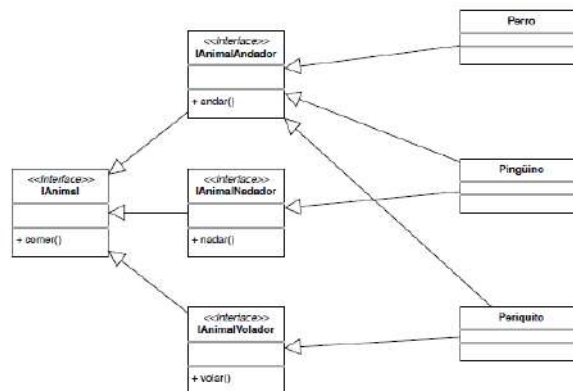


Añadir un cliente puede modificar la interfaz

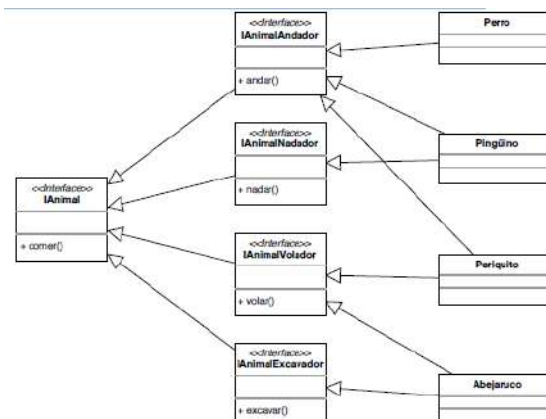


### Aplicación de ISP

Segregamos la interfaz según el cliente



Añadir un nuevo cliente no afecta a las interfaces de los demás clientes



## Principio de inversión de dependencias



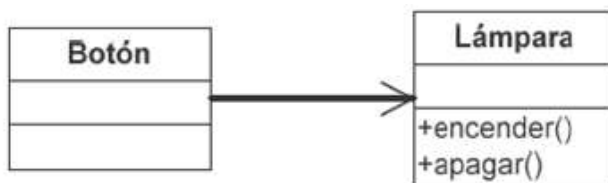
*“Depende de abstracciones; no dependas de implementaciones”*

*Robert C. Martin*

- Los módulos de alto nivel no deben depender de módulos de menor nivel. Ambos deben depender de abstracciones (interfaces)
- Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones
- Lo concreto cambia más que lo abstracto

### *Violación de DIP*

La clase Botón depende de Lámpara para su implementación.

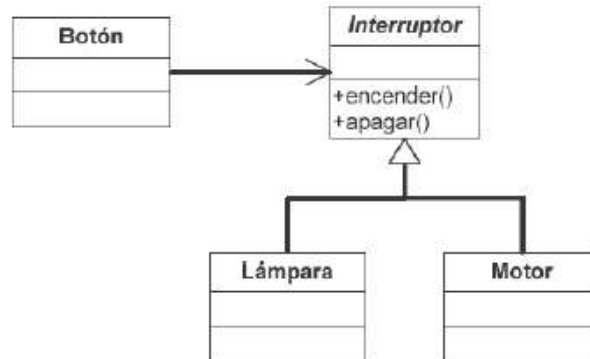


**Problema:** ¿y si queremos añadir otros dispositivos controlado por un botón?

**Solución:** crear una clase que abstraiga los dispositivos controlados por un interruptor.

## Aplicación de DIP

La interfaz Interruptor es propiedad del módulo superior. Ahora es posible añadir nuevas clases.



## Patrones de Diseño

### Motivación

Diseñar software orientado a objetos es difícil y diseñar software reutilizable orientado a objetos es aún más difícil.

El diseño debe ser...

- específico para el problema en cuestión,
- pero también lo suficientemente general como para abordar problemas y requisitos futuros.

Los **patrones de diseño** facilitan la reutilización de diseños y arquitecturas exitosas.

### Patrón

Los **patrones de diseño** plantean **soluciones** parciales (receta) a **problemas de diseño recurrentes**:

- No hay que reinventar la rueda -> reutilizar soluciones buenas
- Cada patrón describe a un conjunto de objetos y clases comunicadas.
- El conjunto se ajusta para resolver un problema en un contexto específico.
- Proporcionan un vocabulario compartido por los diseñadores

Son descripciones de **clases y objetos relacionados** para resolver un problema de **diseño general en contexto concreto**.

Son un esqueleto básico que cada diseñador **adapta** a las peculiaridades de su situación.

## Elementos de un patrón

Cada patrón tiene cuatro elementos:

- **Nombre**
  - Describe el problema y la solución en una o dos palabras.
- **Problema**
  - Explica el problema y su contexto: cuándo usar el patrón (en qué condiciones)
- **Solución**
  - No describe un diseño/solución concretos, sino una plantilla que se aplica a situaciones diferentes.
  - Elementos, responsabilidades, relaciones,..., sin implementación particular
- **Consecuencias**
  - Resultados y ventajas/inconvenientes de aplicar el patrón: espacio, tiempo, flexibilidad, portabilidad, críticas, costos y beneficios
  - Permite evaluar las alternativas de diseño.

		Propósito		
		De creación	Estructurales	De comportamiento
Ámbito	Clase	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

## Clasificación de patrones

El **propósito** la función del patrón:

- **Patrones de creación**
  - Cuándo y cómo crear objetos
- **Patrones estructurales**
  - Cómo combinar objetos en otros objetos mayores
- **Patrones de comportamiento**
  - Cómo distribuir las responsabilidades y cómo comunicar los objetos

Ya puedes sacarte tu B1/B2/C1 de inglés desde casa



Según su **ámbito** se clasifican en:

- **Patrones de Clases:** tratan relaciones entre las clases y sus subclases. Las relaciones se establecen por herencias y son estáticas en tiempo de compilación
- **Patrones de Objetos:** tratan las relaciones entre los objetos. Dichas relaciones pueden cambiar en el tiempo de ejecución y son dinámicas

Patrones de creación

- **Unitario (Singleton)**
- Fábrica abstracta (*Abstract Factory*)
- **Método de Fábrica (Factory Method)**
- Constructor (*Builder*)
- Prototipo (*Prototype*)

*El patrón Singleton – Problema*

A veces necesitamos **clases que tengan una única instancia**. Por ejemplo:

- contadores para asignar identificadores únicos
- controladores de colas de impresión, dispositivos, etc.

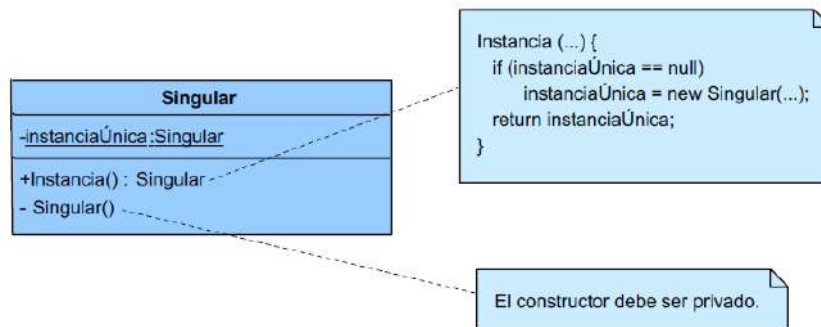
**¿Cómo aseguramos que sólo se crea una instancia?**

¿Cómo proporcionamos un mecanismo de acceso global a esa única instancia?

*El patrón Singleton – Solución*

- Para asegurar que se crea una sola instancia
  - el constructor de la clase no debe ser público
- El almacenamiento de la instancia
  - se hará en una variable de clase (estática)
- El acceso a dicha instancia
  - se hará a través de un método de clase (estático) que devolverá una referencia a la instancia
- La creación de la instancia
  - se hará la primera vez que se invoque este método

## El patrón Singleton – Diagrama



## El patrón Singleton – Java

```
class Singular {
    private static Singular instanciaUnica = null;
    ... // declaración de atributos
    private Singular(...) {
        ... // inicialización de atributos
    }
    public static Singular Instancia() {
        if (instanciaUnica == null)
            instanciaUnica = new Singular(...);
        return instanciaUnica;
    }
    // métodos de instancia
}
```

## El patrón Singleton – Consecuencias

- No debe usarse en exceso, pues es similar a una variable global
  - pero no contamina el espacio global de nombres
- Se puede adaptar para que permita la creación y acceso de un *número acotado* de instancias
- La creación y acceso a la instancia *Singleton* deben serializarse en aplicaciones multihilo
  - Por ejemplo usando un monitor
- Introduce un estado global que dificulta el testeo unitario (*unit testing*)

## El patrón Factory Method – Problema

A veces necesitamos crear instancias de clases que aún no conocemos:

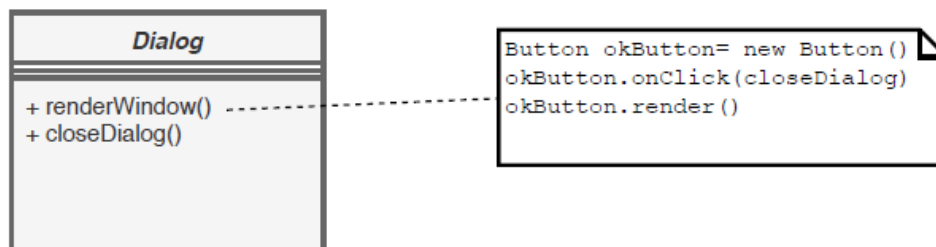
- Los *frameworks* y *toolkits* de aplicaciones trabajan con clases abstractas de las que derivan clases concretas
- Las clases concretas dependen de cada aplicación
- El *framework* o *toolkit* no conoce las clases concretas que usará una aplicación

¿Cómo creamos instancias de las clases concretas que aún no conocemos?

## El patrón Factory Method – Ejemplo

**Problema:** framework para crear interfaces gráficas multiplataforma (p.ej. Windows, web, Linux, etc).

- El framework define una abstracta *Dialog* para mostrar cuadros de diálogo con mensaje y un botón “OK”
- Existe una serie de pasos comunes a todos los sistemas (p.ej. crear botón, asignarle un callback y mostrarlo).
- Pero otros son dependientes de la plataforma (p.ej. renderizar el botón)



## El patrón Factory Method – Solución

Creamos instancias a través de una interfaz de fabricación que delega en sus subclases:

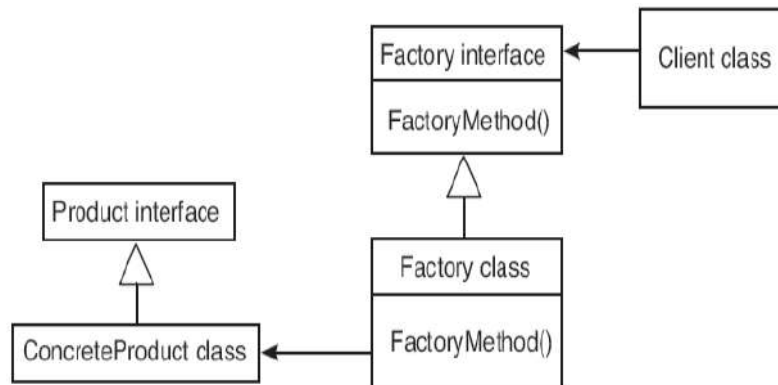
1. **Clase base:** definimos una interfaz de fabricación (*factory method*)
2. **Subclases:** implementan el *factory method*

Se comporta como un constructor virtual

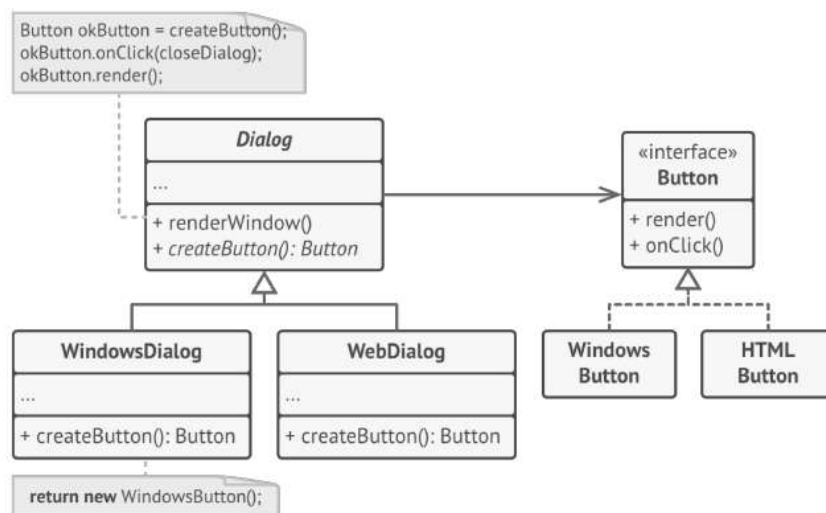
**Ejemplo:** subclases de *Dialog* redefinen método *createButton()*

- Devuelve el tipo de botón adecuado para esa plataforma.

### El patrón Factory Method – Diagrama



### El patrón Factory Method – Ejemplo 2



### El patrón Factory Method – Consecuencias

- El código cliente no crea objetos (**new**) de clases concretas
- El código cliente no trabaja con clases concretas
- Los objetos devueltos por un *factory method* no tienen por qué ser nuevos
- Se comporta como un constructor, pero es polimórfico, con nombres más descriptivos y comportamientos enriquecidos



Ya puedes sacarte tu B1/B2/C1 de inglés desde casa

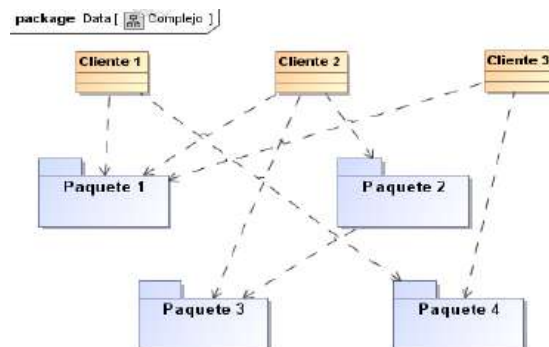


## Patrones estructurales

- Adaptador (*Adapter*)
- Puente (*Bridge*)
- Compuesto (*Composite*)
- Decorador (*Decorator*)
- Fachada (*Facade*)
- Peso mosca (*Flyweight*)
- Representante (*Proxy*)

### El patrón Facade – Problema

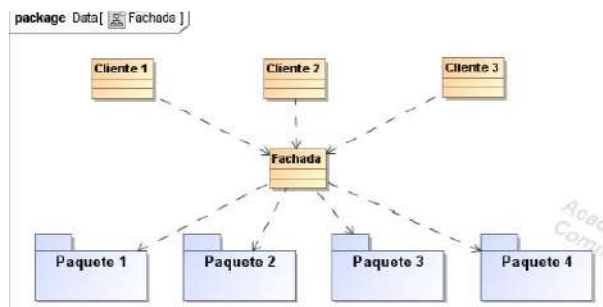
A veces necesitamos trabajar con sistemas (APIs, *frameworks*, *toolkits*) muy complejos:



¿Cómo usar la funcionalidad básica ignorando la complejidad del sistema?

### El patrón Facade – Solución

Definir una clase Fachada que facilite una interfaz de alto nivel, unificada y simplificada a todas las interfaces del sistema.



## El patrón Façade – Consecuencias

- Facilita una interfaz simple a un sistema complejo
- Permite disponer el sistema en capas
  - Cada fachada es un nivel de abstracción o capa
- Reduce el acoplamiento entre los clientes y los subsistemas
  - Aumenta la portabilidad, facilita mantenimiento
- Limita la funcionalidad
  - Pero se puede seguir accediendo al subsistema
- No añade nueva funcionalidad

## Patrones de comportamiento

- Cadena de responsabilidad (*Chain of responsibility*)
- Orden (*Command*)
- Intérprete (*Interpreter*)
- Iterador (*Iterator*)
- Mediador (*Mediator*)
- Memento (*Memento*)
- Observador (*Observer*)
- Estado (*State*)
- **Estrategia (*Strategy*)**
- Método plantilla (*Template Method*)
- Visitante (*Visitor*)

## El patrón Strategy – Problema

A veces necesitamos elegir un algoritmo o comportamiento adecuado:

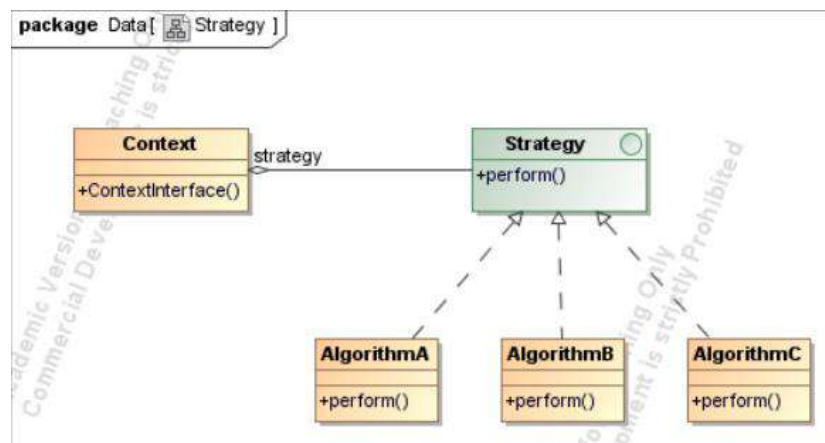
```
datos comprimir(datos f) {  
    switch(tipoCompresion) {  
        case zip : return ZIP(f);  
        case rar  : return RAR(f);  
    }  
}
```

¿Cómo añadir nuevos algoritmos sin modificar el código (principio OCP)?

## El patrón Strategy – Solución

- Definir una **interfaz estrategia**
  - Define los métodos comunes a todos los algoritmos (estrategias)
  - Todos los algoritmos implementan esta interfaz (estrategias concretas)
- Definir una **clase contexto**
  - Contiene una referencia a una estrategia concreta
  - Usa esta referencia para invocar el algoritmo elegido
- El código **cliente**
  - Crea la estrategia concreta (elige el algoritmo)
  - Configura el contexto (estrategia concreta y datos de entrada)
  - Se comunica con el contexto exclusivamente

## El patrón Strategy – Diagrama



## El patrón Strategy – Estrategias

```
public interface Compresor {
    public datos comprimir(datos f);
}

public class ZIP implements Compresor {
    public datos comprimir(datos f) {...}
}

public class RAR implements Compresor{
    public datos comprimir(datos f) {...}
}
```

### *El patrón Strategy – Contexto*

```
public class Contexto {  
    private Compresor estrategia;  
    public Contexto(Compresor comp){  
        estrategia= comp;  
    }  
    public setEstrategia(Compresor comp){  
        estrategia= comp;  
    }  
    public datos aplicar(datos f) {  
        return estrategia.comprimir(f)  
    }  
}
```

### *El patrón Strategy – Cliente*

```
public class Cliente {  
    datos original=...;  
    Contexto compresor;  
  
    compresor = new Contexto(new ZIP());  
    datos d1 = compresor.aplicar(original);  
  
    compresor.setEstrategia(new RAR());  
    datos d2 = compresor.aplicar(original);  
    ...  
}
```

Ya puedes sacarte tu B1/B2/C1 de inglés desde casa

#LinguaskillEnCasa



### *El patrón Strategy – Consecuencias*

- Familia de algoritmos e implementaciones
- Alternativa a:
  - Construcciones condicionales
  - Herencia
- El cliente debe conocer las estrategias
  - El cliente crea la estrategia concreta (**new**)
  - Solución: puede hacerlo el contexto (*factory method*)
- Comunicación entre contexto y estrategia
  - El contexto debe pasar la entrada a la estrategia
  - La estrategia debe devolver resultados al contexto
  - La estrategia puede tener una referencia al contexto

### *Ventajas de los patrones de diseño*

- **Son soluciones técnicas:**
  - Dada una determinada situación, los patrones indican cómo resolverla mediante un buen diseño
  - Existen patrones específicos para un lenguaje determinado, y otros de carácter más general
- **Se aplican en situaciones muy comunes:**
  - Proceden de la experiencia
  - Han demostrado su utilidad para resolver problemas que aparecen frecuentemente en el diseño
- **Facilitan la reutilización de las clases y del propio diseño:**
  - Los patrones favorecen la reutilización de clases ya existentes y la programación de clases reutilizables
  - La propia estructura del patrón es reutilizada cada vez que se aplica

## *Inconvenientes de los patrones de diseño*

- El uso de un patrón no se refleja claramente en el código:
  - A partir de la implementación es difícil determinar qué patrón de diseño se ha utilizado
  - No es posible hacer ingeniería inversa
- Es difícil reutilizar la implementación de un patrón:
  - Las clases del patrón son roles genéricos, pero en la implementación aparecen clases concretas
- Los patrones suponen cierta sobrecarga de trabajo a la hora de implementar:
  - Se usan más clases de las estrictamente necesarias
  - A menudo un mensaje se resuelve mediante delegación de varios mensajes a otros objetos

## Refactorización

### Concepto de refactorización

- Cambios realizados en la estructura interna de un producto software
  - para facilitar su comprensión
  - para hacer menos costosa su modificación
  - sin cambiar su comportamiento observable
- Suele involucrar una modificación pequeña del software
- El proceso de refactorización puede implicar varios cambios consecutivos

### Ventajas de refactorización

- Mejora el diseño del software
- Facilita su comprensión
- Ayuda a encontrar errores
- Ayuda a pasar del diseño a la implementación de manera más “ágil”

## ¿Cuándo refactorizar?

Aplicaremos refactorización cuando:

- se añada funcionalidad al sistema
- se necesite arreglar un error
- se haga una revisión de código

Después de refactorizar se deben volver a pasar las pruebas

## Limitaciones de la refactorización

Puede implicar cambios en la interfaz

- mantener las interfaces originales junto con las nuevas

Puede disminuir las prestaciones

- eficiencia vs. legibilidad y mantenibilidad

A veces es mejor empezar desde cero

- mantener la integridad del diseño

## Candidatos a la refactorización

- Duplicación de código:
  - en la misma clase
  - en distintas subclases de una clase dada
  - en clases no relacionadas directamente
- Métodos excesivamente grandes:
  - con muchos parámetros
  - con muchas variables temporales
  - con muchos bucles
- Clases grandes:
  - con muchas variables de instancia
  - con métodos duplicados
- Si un simple cambio produce muchos pequeños cambios en clases distintas
- Métodos que acceden en exceso a datos de clases ajenas
- Grupos cohesionados de datos
  - pueden constituir clases independientes
- Instrucciones de selección de casos
  - Vinculación dinámica
- Jerarquías de herencia paralelas

- Cambios divergentes
  - si una clase necesita modificarse de distintas formas, atendiendo a razones independientes
    - suele ser conveniente dividir la clase en varias
- Generalización especulativa
  - previsión de métodos que quizá puedan necesitarse en el futuro
- Uso excesivo de objetos intermediarios
- Alto grado de acoplamiento entre clases
  - a menudo motivado por un mal uso de la herencia
- Bibliotecas de clases incompletas
- Clases innecesarias

## Un catálogo de refactorizaciones

- Renombrar método
  - el nombre no ofrece información sobre su propósito
- Mover método
  - un método es usado más en otra clase que en su propia clase
- Extraer método
  - un fragmento de código se puede agrupar según algún criterio: definir método que lo encapsule
- Introducir asertos
  - una sección de código supone alguna propiedad sobre el estado del programa: hacerlo explícito con un assert
- Encapsular atributo
  - un atributo público debe definirse privado y proporcionar funciones de acceso (setter/getter)
- Preservar la unidad de los objetos
  - si se pasan como parámetros en una invocación datos sobre un mismo objeto, es conveniente enviar el objeto completo
- Extracción de clases
  - si una clase hace el trabajo que deberían hacer dos, debe crearse una nueva clase asociada a ella y trasladar allí los campos y métodos relevantes

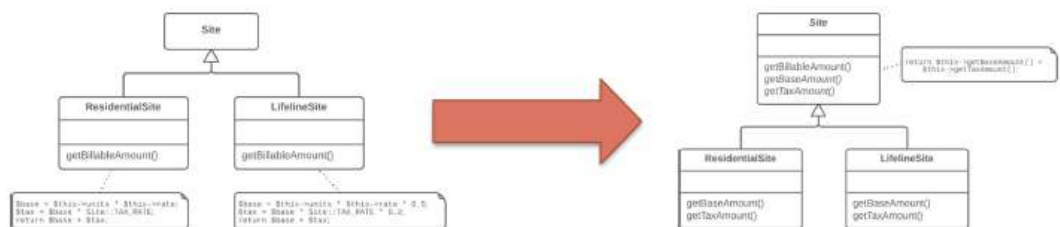




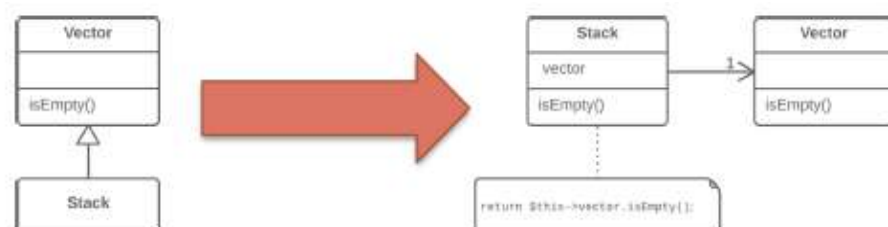
## Ya puedes sacarte tu B1/B2/C1 de inglés desde casa

#LinguaskillEnCasa

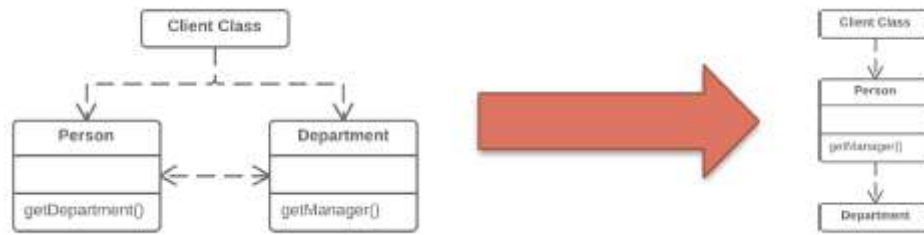
- Colapsar jerarquías de herencia
  - si una clase y su subclase no difieren significativamente, fundirlas en una sola
- Promoción de métodos hacia arriba
  - si dos métodos están definidos de forma similar en dos subclases, agruparlos en la clase padre común
- Construcción de una plantilla de método
  - si métodos en dos subclases realizan pasos similares en el mismo orden, pero los pasos son diferentes:
    - cada paso se encapsula en un método con igual signatura, de forma que los métodos originales lleguen a ser idénticos
    - entonces, dichos métodos pueden subir en la jerarquía



- Cambiar asociación bidireccional a unidireccional
  - si se tiene una asociación bidireccional pero una clase no necesita características de la otra
    - eliminar el extremo innecesario de la asociación
- Remplazar herencia con delegación
  - si una subclase utiliza solo parte de la interfaz de una superclase, o no quiere heredar datos
    - crear un campo para la superclase, ajustar métodos y delegar a la superclase, eliminando la subclasificación



- Ocultar delegados
  - si un cliente invoca una clase delegada de un objeto
    - crear métodos sobre el servidor, para ocultar el delegado (desacopla clases)



- Reemplazar condicional con polimorfismo
  - si una estructura condicional selecciona el comportamiento de un objeto dependiendo del tipo
    - cambiar cada rama del condicional a un método redefinido en una subclase
- Introducir objeto nulo
  - si se hacen comprobaciones repetitivas sobre un valor “null”,
    - reemplazar “null” con un objeto nulo.
    - ayuda a que el código sea más legible y corto.

## Tema 7 – Verificación y pruebas

Introducción.....	2
Conceptos básicos.....	2
Depuración de errores .....	2
Principios a seguir .....	3
¿Quién prueba el software? .....	3
Tipos de prueba .....	3
Clasificación de tipos de pruebas.....	3
Relación entre los tipos de pruebas.....	4
Pruebas de caja blanca y caja negra .....	4
Pruebas de caja blanca.....	5
Prueba de los caminos base.....	5
Pruebas de bucles .....	9
Pruebas de condición.....	10
Pruebas de caja negra.....	12
Partición de equivalencia.....	13
Análisis de valores límite.....	14
Test Driven Development (TDD) .....	15

---

# VERIFICACIÓN Y PRUEBAS

---

## Introducción

### Conceptos básicos

**Fallo software (bug):** El software no se comporta como se esperaba

**Causas:**

- *Requerimientos* incompletos, imprecisos o imposibles
- *Especificación, diseño o código* incompletos o incorrectos.

Las pruebas de software (**software testing**) se definen como:

El proceso que permite **identificar la corrección**, completitud, seguridad y calidad **del producto software desarrollado** antes de que éste sea usado.

**NO SON**

- Un método para demostrar que NO hay errores
- Un método para demostrar que el software funciona correctamente

Una prueba en sí es el proceso de ejecutar ciertos componentes software con el fin de **encontrar errores**.

**Un caso de prueba es:** Conjunto de condiciones bajo las cuales se puede determinar si los requisitos del software se cumplen parcial o totalmente, o bien no se cumplen. Un buen caso de prueba es aquel que tiene una **alta probabilidad de encontrar un error** no descubierto hasta entonces.

**Definición de error:** Discrepancia entre un valor o condición calculado, observado o medido y el valor o condición específica teóricamente correctos.

### Depuración de errores

**Tradicionalmente** la fase de pruebas de un proyecto tenía lugar después de la codificación y antes de la entrega del producto al cliente.

*Inconvenientes:* errores en etapas tempranas (requisitos, modelado) son muy costosos de reparar

**Alternativa:** Realizar pruebas en paralelo a la fase de desarrollo.

**Alternativa extrema: Test Driven Development:** las pruebas pasan a ser el centro del proceso de desarrollo.

## Principios a seguir

- Saber el resultado esperado del programa es fundamental en un caso de prueba.
- Cuanto más se modifique un programa, más hay que probarlo.
- Es fundamental documentar bien los casos de prueba.
- Definir los casos de prueba en la fase de diseño.
- Un programador debe evitar probar su programa.

## ¿Quién prueba el software?



**Desarrollador**



**Probadores Independientes**

Comprende el sistema pero probará “amablemente”. Está condicionado por la entrega

Debe comprender el sistema, pero intentará “romperlo”. Está dirigido por la calidad

## Tipos de prueba

### Clasificación de tipos de pruebas

**Pruebas unitarias** (de componentes aislados)

**Pruebas de integración** (varios componentes a la vez)

**Pruebas de validación** (¿se cumplen los requisitos? Pruebas alfa y beta)

**Pruebas de sistema** (sistema completo)

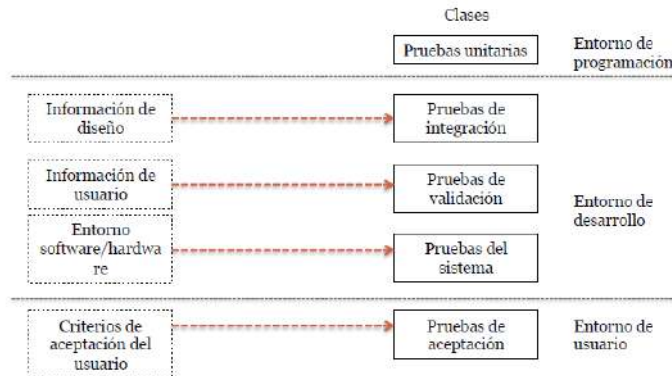
**Pruebas de aceptación** (validación por el usuario)

**Pruebas de regresión** (después de un cambio, verificar que no se han introducido errores)

**Pruebas de carga** (prueba bajo carga normal)

**Pruebas de estrés** (prueba bajo alta carga)

## Relación entre los tipos de pruebas



## Pruebas de caja blanca y caja negra

Son dos tipos de pruebas relacionadas con la ejecución de código.

### Pruebas de caja blanca:

- Comprueban el funcionamiento interno y la lógica del código.

### Pruebas de caja negra:

- Son pruebas guiadas por los datos de entrada y de salida, sin tener en cuenta los detalles de implementación.
- No se dispone de acceso al código fuente.

# Pruebas de caja blanca

**Idea básica:** Asegurar **que todas las sentencias y condiciones** se han **ejecutado al menos una vez**.

Hay que diseñar casos de prueba que garanticen que:

- Se ejecuten por lo menos una vez todos los **caminos independientes** de cada módulo.
- Se comprueben los **bucles** en sus límites
- Se comprueben todas las **decisiones lógicas** (V y F)
- Se analicen **estructuras internas de datos**



**Cobertura de código:** Porcentaje del código sometido a pruebas.

- Los caminos menos frecuentemente visitados son los que suelen contener mayor proporción de errores.

## Prueba de los caminos base

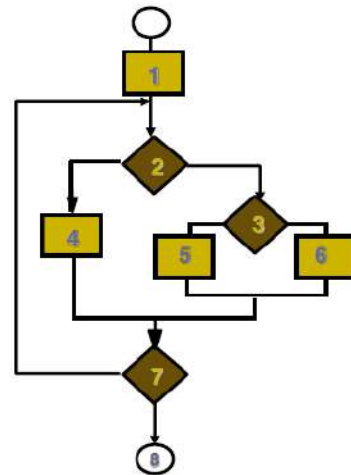
**Idea:** diseñar casos de prueba que ejecuten cada camino independiente del código al menos una vez.

**Pregunta:** ¿cómo determinamos el nº de caminos independientes?

**Respuesta:** complejidad ciclomática del grafo de flujo.

**Pasos:**

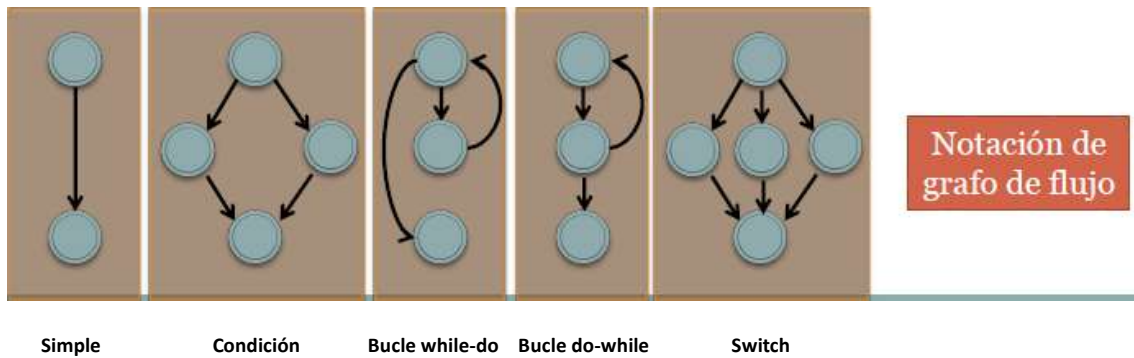
1. Construir el *grafo de flujo*
2. Determinar la **complejidad ciclomática**
3. Usar esta medida para la definición de un conjunto de **caminos base** de ejecución.
4. Elaborar **casos de prueba** que fuercen cada uno de los caminos base.



### Complejidad ciclomática

- Es una métrica que proporciona una medición cuantitativa de la complejidad lógica de un programa.
- Indica el número de caminos base de un grafo de ejecución de un programa

**Camino base:** Camino linealmente independiente dentro del código



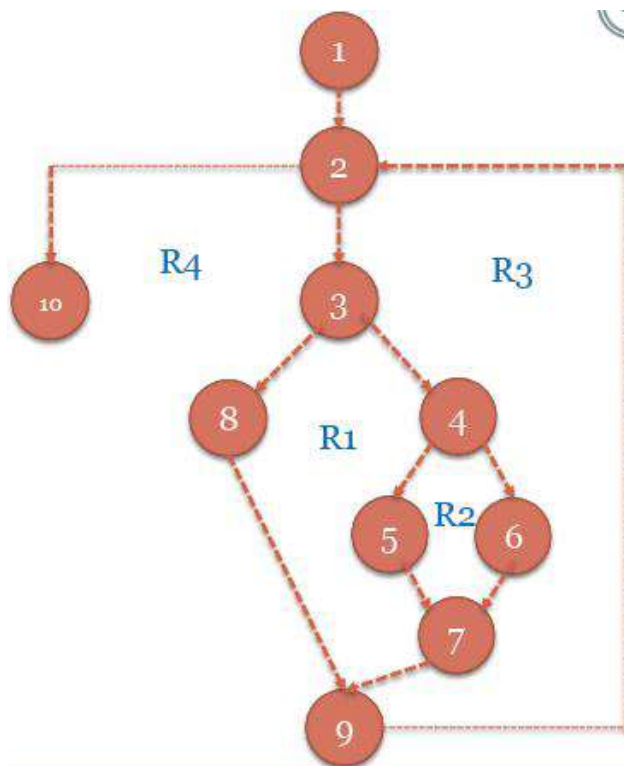
**Complejidad ciclomática:** límite superior para el número de pruebas que se deben realizar para asegurar que se ejecutan todas las sentencias al menos una vez.

Se puede calcular de tres formas:

- $V(G) = \text{Nº de regiones}$
- $V(G) = \text{Aristas} - \text{Nodos} + 2$
- $V(G) = P + 1$ , donde P es el número de nodos predicados (nodos que contienen una condición, por lo que dos o más aristas parten de ellos)
- El número de caminos base coincide con la complejidad ciclomática.

A mayor valor de complejidad ciclomática, mayor probabilidad de errores.

*Ejemplo*



$$V(G) = \text{nº regiones} = 4$$

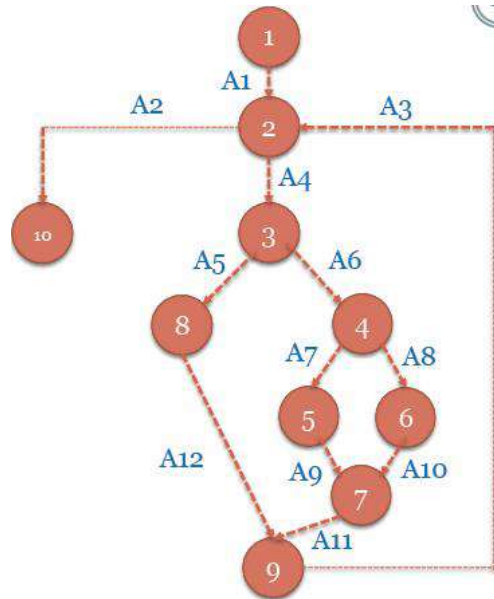


$$V(G) = A - N + 2$$

$$N = n^{\circ} \text{ nodos} = 10$$

$$A = n^{\circ} \text{ Aristas} = 12$$

$$V(G) = 12 - 10 + 2 = 4$$



$$V(G) = |P| + 1$$

$P$  = nodos predicados

$$P = \{2, 3, 4\}$$

$$V(G) = 3 + 1 = 4$$

• Caminos base:

- 1, 2, 10
- 1, 2, 3, 8, 9, 2, 10
- 1, 2, 3, 4, 6, 7, 9, 2, 10
- 1, 2, 3, 4, 5, 7, 9, 2, 10
- Los casos de prueba han de cubrir todos estos caminos
- Un analizador dinámico podría comprobar que estos caminos han sido ejecutados

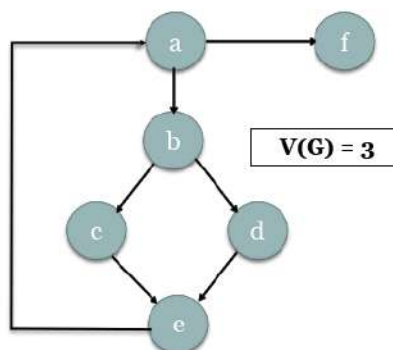
### Ejemplo 2

Dada la siguiente función, que calcula el MCD de dos números, obtener el **grafo** correspondiente, la **complejidad ciclomática** y los **caminos base**:

```
public int MCD (int x, int y) {
    while (x != y)          // a
    {
        if (x > y)          // b
            x = x - y;      // c
        else
            y = y - x;      // d
        }                  // e
    return x;              // f
}
```

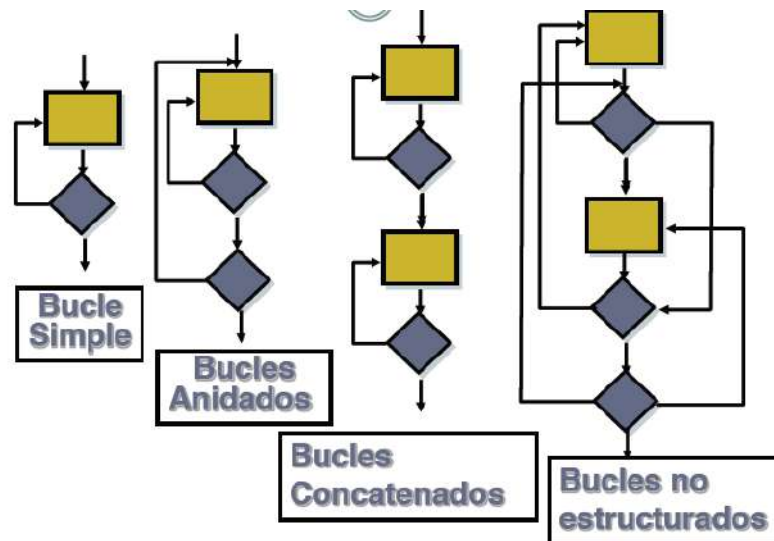
### • Solución:

```
public int MCD (int x, int y) {
    while (x != y) // a
    {
        if (x > y) // b
            x = x - y; // c
        else
            y = y - x; // d
        } // e
    return x; // f
}
```



Caminos base	Casos de prueba
a, f	$\text{mcd}(x=1, y=1) \Rightarrow 1$
a, b, c, e, a, f	$\text{mcd}(x=4, y=2) \Rightarrow 2$
a, b, d, e, a, f	$\text{mcd}(x=2, y=4) \Rightarrow 2$

## Pruebas de bucles



### *Bucles simples*

Siendo  $n$  el número máximo de pasos

- Saltar el bucle
- Pasar una sola vez
- Pasar dos veces
- Hacer  $m$  pasos, siendo  $m < n$
- Hacer  $n - 1$ ,  $n$  y  $n + 1$  pasos (**casos límite**)

Total: 7 pruebas

### *Bucles anidados*

Impráctico seguir el esquema de bucles simples: el número de pruebas aumentaría geométricamente conforme el nivel de anidamiento.

Aproximación práctica:

- Comenzar por el bucle más interior y los demás al mínimo
- Pruebas de bucles simples para el bucle más interior, resto a mínimos
- Progresar hacia fuera y para cada bucle realizar pruebas de bucles simples manteniendo los bucles externos en mínimos y los bucles internos en sus valores medios o típicos

### *Bucles concatenados*

Si son independientes, se prueba como bucles simples. Si no, se prueban como bucles anidados

### *Bucles no estructurados*

Son bucles mal diseñados: rediseñar de forma correcta

## Pruebas de condición

Las condiciones de una sentencia pueden ser:

### Simple:

- Una variable lógica que se evalúa a verdadero o falso
- Una expresión relacional del tipo  $a \text{ op } b$ , donde  $\text{op}$  puede ser  $>$ ,  $>=$ ,  $<$ ,  $<=$ ,  $=$ ,  $<>$

### Compuestas:

- Varias condiciones simples, operadores lógicos (AND, OR, NOT) y paréntesis

Tipos de pruebas relacionadas con condiciones y decisiones:

- De cobertura de **condición** (condiciones simples).
- De cobertura de **decisión** (condiciones compuestas).
- De cobertura de **decisión/condición**.

En las decisiones hay que probar las ramificaciones:

- Probar la rama verdadera
- Probar la rama falsa
- Probar cada condición simple

### *Pruebas de condición. Ejemplo*

```
public boolean comprobarHora(int hora, int minuto, int segundo)
{
    boolean horaTieneFormatoCorrecto = false;
    if ((hora >= 0) && (hora <= 23)) {
        if ((minuto >= 0) && (minuto <= 59)) {
            if ((segundo >= 0) && (segundo <= 59)) {
                horaTieneFormatoCorrecto = true
            }
        }
    }
    return horaTieneFormatoCorrecto ;
}
```

### Tres decisiones:

D1:  $((h \geq 0) \ \&\& \ (h \leq 23))$

D2:  $((m \geq 0) \ \&\& \ (m \leq 59))$

D3:  $((s \geq 0) \ \&\& \ (s \leq 59))$

### Casos de prueba: ejemplos de valores

	Verdadero	Falso
D1	$h=10$	$h=24$
D2	$m=25$	$m=65$
D3	$s=12$	$s=70$

### Casos de prueba para cubrir las decisiones:

Caso 1: D1 = verdadero, D2 = verdadero, D3 = verdadero

Caso 2: D1 = verdadero, D2 = verdadero, D3 = falso

Caso 3: D1 = verdadero, D2 = falso

Caso 4: D1 = falso

### Tres decisiones, dos condiciones por decisión:

- D1:  $((h \geq 0) \ \&\& \ (h \leq 23))$ 
  - C1.1:  $(h \geq 0)$
  - C1.2:  $(h \leq 23)$
- D2:  $((m \geq 0) \ \&\& \ (m \leq 59))$ 
  - C2.1:  $(m \geq 0)$
  - C2.2:  $(m \leq 59)$
- D3:  $((s \geq 0) \ \&\& \ (s \leq 59))$ 
  - C3.1:  $(s \geq 0)$
  - C3.2:  $(s \leq 59)$

Hay que garantizar que cada condición tome al menos una vez el valor verdadero y otra el falso, garantizando la cobertura de la decisión.

	Verdadero	Falso
C1.1	$h=10$	$h = -1$
C1.2	$h=10$	$h=24$
C2.1	$m=30$	$m=-1$
C2.2	$m=30$	$m=60$
C3.1	$s=50$	$s=-1$
C3.2	$s=50$	$s=70$

Casos de prueba:

Caso 1: C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = V, C3.1 = V, C3.2 = V

Caso 2: C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = V, C3.1 = V, C3.2 = F

Caso 3 C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = V, C3.1 = F, C3.2 = V

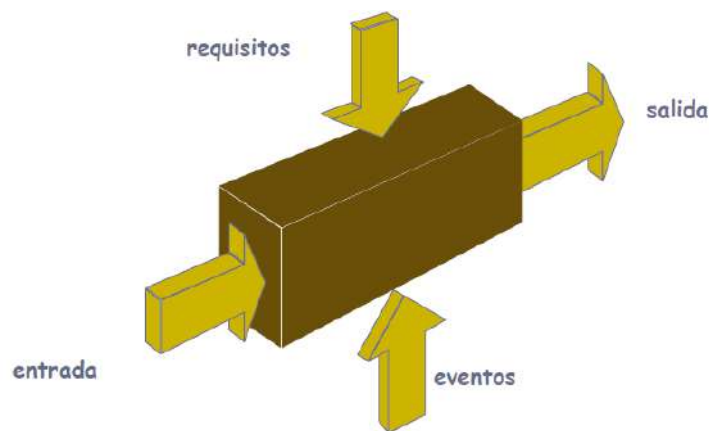
Caso 4: C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = F

Caso 5: C1.1 = V, C1.2 = V, C2.1 = F, C2.2 = V

Caso 6: C1.1 = V, C1.2 = F

Caso 7: C1.1 = F, C1.2 = V

### Pruebas de caja negra



Se centran en los **requisitos funcionales** del software

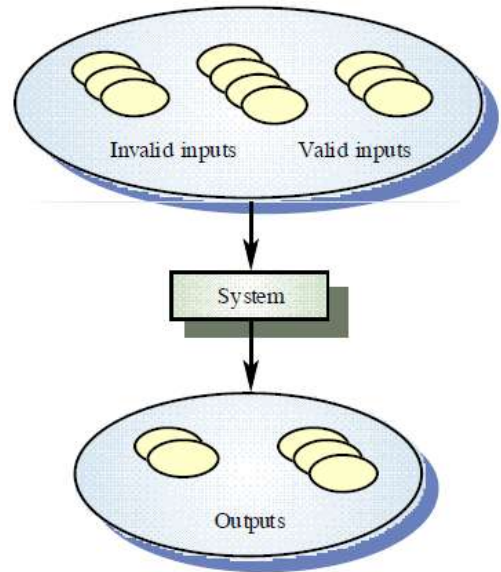
- Obtener conjuntos de condiciones de entrada que prueben todos los requisitos funcionales del programa.
- No es una alternativa a las pruebas de caja blanca
  - Es un enfoque complementario
  - Suelen descubrir tipos de errores diferentes a los obtenidos con las técnicas de caja blanca.

Trata de encontrar errores en:

- Funciones incorrectas o ausentes
- Errores de interfaz
- Errores en estructuras de datos o en accesos a bases de datos externas
- Errores de rendimiento
- Errores de inicialización y de terminación

Ignora intencionadamente la estructura de control. Se suele aplicar durante las fases finales de las pruebas.

Están conducidas por los datos de entrada y de salida. Hay que conocer de antemano las salidas correctas.



**Dos técnicas:**

- Partición de equivalencia
- Análisis de valores límites

## Partición de equivalencia

**Idea:**

- Hay que definir casos que descubran clases de errores
- Dividir el dominio de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba.

Evaluar clases de equivalencia para una condición de entrada.

- Una clase de equivalencia representa un conjunto de estados válidos o inválidos para condiciones de entrada.

Se pueden definir de acuerdo con las siguientes directrices:

- Un rango: define una clase de equivalencia válida y dos inválidas
- Un valor específico: una válida y dos no válidas
- Un miembro de un conjunto: una válida y una no válida
- Lógica: una válida y una no válida

### *Ejemplo 1:*

Una entrada es un entero de 5 dígitos entre 10.000 y 99.999.

Las particiones equivalentes son

- $<10.000$
- $10.000 - 99.999$
- $> 100.000$

Luego hay que elegir casos de prueba en los límites de estos conjuntos:

- 00000, 09999, 10000, 99999, 100000, 100001

### *Ejemplo 2:*

Aplicación que acepta un nº de mes e imprime su nombre

Las particiones equivalentes son

- $<1$  (inválido)
- $1 - 12$  (válido)
- $12$  (inválido)

### *Ejemplo 3:*

Formulario de autenticación con usuario y password

**Usuario:** puede ser verdadero o falso. Particiones equivalentes:

- Vacío
- Válido
- Inválido

**Password:** puede ser verdadero o falso. Particiones equivalentes:

- Vacío
- Válido
- Inválido

## Análisis de valores límite

Los errores tienden a darse más en los límites del campo de entrada que en el centro

- Consecuentemente, hay que plantear casos de pruebas que ejerciten los valores límite.

Esta estrategia es complementaria a la partición equivalente



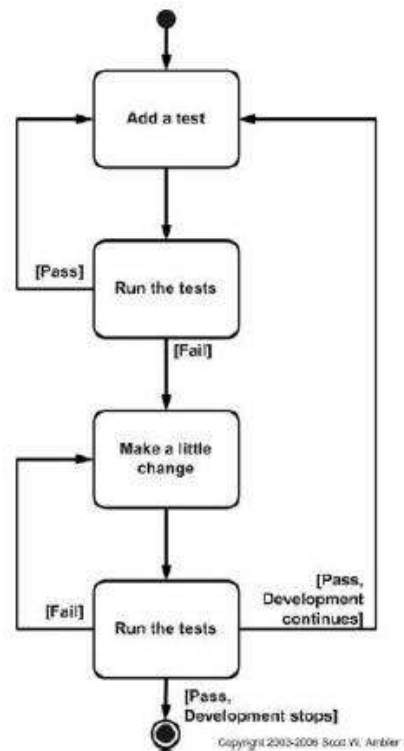
# Test Driven Development (TDD)

**Desarrollo dirigido por pruebas.** Característica clave de la **Extreme Programming**.

Proceso de desarrollo software basado en el siguiente ciclo:

1. Escritura de un caso de prueba para una funcionalidad a incluir en el software
2. Implementar el código que pase el test
3. Refactorizar el código

**Herramientas:** Junit



## Ventajas

- Al centrarse en las pruebas, el código tiene menos errores y se aumenta la productividad.
- Separación entre el comportamiento esperado y la implementación para conseguirlo
- Tests de regresión: al desarrollar de forma incremental los tests de regresión permiten encontrar errores
- Se reduce la necesidad de depurar código

## Inconvenientes

- Frecuentemente el diseñador de los tests implementa el código
- Sensación de seguridad que puede ser irreal