

# Resumen Ingeniería del Software

Facultad de Informática UCM. Curso 2012-2013

Autor: Javier López

Revisor: Federico Peinado

## 1.- El producto

Hoy en día el **software** es el factor principal en el presupuesto. Los costes se centran principalmente en ingeniería y no en fabricación.

A diferencia del hardware, tiene una curva de fallos muy diferente (ver gráficos en las diapositivas).

Es de vital importancia la generación de software de calidad, fiable/robusto y fácilmente mantenible y reutilizable.

## 3.- El proceso

La **visión general de la IS** se centra en el desarrollo de software siguiendo un **proceso**. Existen varios **modelos** para este proceso, pero todos tienen 3 **fases** genéricas:

- **Definición:** se identifican los requisitos del sistema y software. Información a procesar, función y rendimiento, comportamiento, interfaces, restricciones de diseño, etc.
- **Desarrollo:** se define cómo han de diseñarse las estructuras de datos, la implementación de funciones, la traducción a un lenguaje de programación, pruebas, etc.
- **Mantenimiento:** centrada en el cambio a corrección de errores, adaptaciones por la evolución del entorno software o cambios solicitados por el cliente.

En este punto se vuelven a aplicar todas las fases pero sobre un producto software ya terminado. Aquí pueden llevarse a cabo varios cambios: corrección de defectos, adaptaciones a cambios software, modificaciones introducidas por el cliente o “cambios para facilitar los cambios”.

La IS tiene varias capas:

- **Capa de enfoque de calidad:** la IS se basa en calidad. Esto es, mejores técnicas de construcción de software.
- **Capa de proceso:** conjunto de actividades y resultados que sirven para construir un producto software. Una calidad y métodos.
- **Capa de métodos:** un método incluye análisis de requisitos, diseño, construcción, pruebas y mantenimiento y suele estar bastante ligado al proceso.
- **Capa de herramientas:** soporte automático o semiautomático para el proceso y los métodos. De gran importancia en la IS.

Como cada proyecto es diferente, durante el proceso se adapta al proyecto definiendo una serie de **tareas** para las **actividades estructurales (AEs)**. A cada conjunto de tareas se le denomina **acción de ingeniería del software**. Por ejemplo, para un proyecto la AE “**Ingeniería**” puede descomponerse en las acciones Análisis y Diseño. Y análisis, a su vez, puede descomponerse en las tareas de generar diagramas de flujo, casos de uso, etc. En cambio, para

otro proyecto, “Ingeniería” podría descomponerse en las acciones reingeniería, análisis y diseño.

El **proceso de software** es un conjunto de actividades y resultados asociados que sirven para construir un producto software. El proceso define un marco de trabajo compuesto por un conjunto de **actividades estructurales**, como vimos. Asimismo, también existe una serie de **actividades protectoras (AP)** que no generan software sino que facilitan su desarrollo y mejoran su calidad. Se integran mientras se realizan las AE, pero no son lo mismo. Ejemplos de AP son: seguimiento y control del proyecto software, revisiones técnicas formales, garantías de calidad, preparación y creación de documentos, gestión de riesgos o mediciones.

Algunos de los **modelos genéricos de desarrollo** software son:

- **Modelo “en cascada”**: se avanza de fase en fase pero nunca volviendo a una anterior (salvo al final del desarrollo). Su principal ventaja es que resulta muy sencillo de aprender, pero no es muy realista y un error en las últimas fases puede ser letal.
- **Modelo de desarrollo rápido de aplicaciones (DRA)**: centrado en mantener varios equipos paralelos y la unión de componentes. Cuenta con la ventaja de la rapidez de desarrollo pero con los inconvenientes del número de personas y la necesidad de una buena gestión general.
- **Modelo evolutivo**: este enfoque entrelaza las actividades de especificación, desarrollo y validación. Un sistema inicial se desarrolla rápidamente a partir de especificaciones abstractas. Éste se refina basándose en las peticiones del cliente para producir un sistema que satisfaga sus necesidades.
- **Prototipado**: se usa un prototipo para dar una visión al usuario del sistema en general. Hay dos tipos de prototipado:
  - Exploratorio: el prototipo inicial se refina hasta llegar a la versión final.
  - De usar y tirar: de cada prototipo se toman las mejores partes para hacer el siguiente pero cada prototipo se tira entero.

La principal ventaja de este modelo es que se pueden identificar los requisitos incrementalmente y tiene una alta visibilidad para desarrolladores y clientes. El problema es que el cliente no suele entender por qué tirar prototipos y el desarrollador puede terminar creando software de baja calidad.

- **En espiral**: se basa en la repetición continua de una serie de fases:
  - Determinar alternativas, objetivos y restricciones
  - Gestión de riesgos
  - Desarrollar el siguiente nivel del proyecto y testarlo
  - Planificar la siguiente fase

Su ventaja es que es más realista pero menos probado que el de cascada o prototipado.

- **Basado en componentes**: este enfoque se basa en la existencia de un número significativo de componentes reutilizables. El proceso de desarrollo del sistema se enfoca en integrar estos componentes en el sistema más que en desarrollarlos desde cero. Tiene como fases:

- Identificar los componentes candidatos
- Buscarlos y extraerlos de la biblioteca
- Construir los componentes que falten
- Añadirlos a la biblioteca
- Construir la iteración N del sistema

La mayor ventaja es la creación de componentes reutilizables y la desventaja el tamaño del catálogo de componentes.

#### Actividades habituales en un proceso de desarrollo de software:

- **Especificación** - establecer los requisitos y restricciones del sistema
- **Diseño** - producir un modelo en papel del sistema
- **Manufactura** - construir el sistema
- **Prueba** - verificar que el sistema cumpla con las especificaciones requeridas
- **Instalación** - entregar el sistema al usuario y asegurarse de que satisface sus necesidades
- **Mantenimiento** - reparar fallos en el sistema cuando sean descubiertos

**Proceso Unificado de Desarrollo:** el ***Rational Unified Process (RUP)*** –versión comercial frente al **OpenUP**, la libre- es la realización más detallada de este proceso y, por lo tanto, la que más se suele usar.

Está basado en componentes y muy ligado a UML. Está dirigido por casos de uso, centrado en la arquitectura y es iterativo e incremental.

- **Caso de uso:** es un fragmento de funcionalidad del sistema que proporciona al usuario un resultado importante (requisitos funcionales). Especifican una secuencia de acciones que pueden llevar a cabo el sistema interaccionando con sus actores<sup>1</sup>. Puede incluir secuencias alternativas.
- **Arquitectura:** es una vista del diseño completo con las características más importantes resaltadas. Se analiza continuamente y se refina hasta obtener una versión estable.
- **Iterativo e incremental:** se divide el proyecto en varios “mini-proyectos” que componen las iteraciones. Tras cada una se acoplan al producto final de manera incremental.

Está formado por cinco **flujos de trabajo (AEs)** que se iteran en cada fase:

- **Requisitos:** en primer lugar se procede a la captura de requisitos. Se enumeran los requisitos candidatos, se comprende el dominio del sistema y se capturan los funcionales y no funcionales. El papel de los requisitos varía en función de la fase. En inicio buscamos los más relevantes, en elaboración sirven como apoyo a la distribución de esfuerzos, en construcción para la implementación y en transición apenas se emplean.  
Un modelo de dominio captura los tipos de objetos y sucesos más relevantes del contexto del sistema. Por ejemplo, una factura, cuenta, pedido, etc.  
El modelado de negocio es una técnica para comprender los procesos de negocio de la organización. El modelo de casos de uso del negocio ofrece la visión de los procesos de negocio de una empresa de forma gráfica. Tendremos actores, objetos del modelo de dominio e interacción entre ambos.
- **Análisis:** durante esta fase se estudian los requisitos refinándolos y modelándolos. El objetivo es conseguir una descripción de los mismos que sea fácil de mantener y ayude

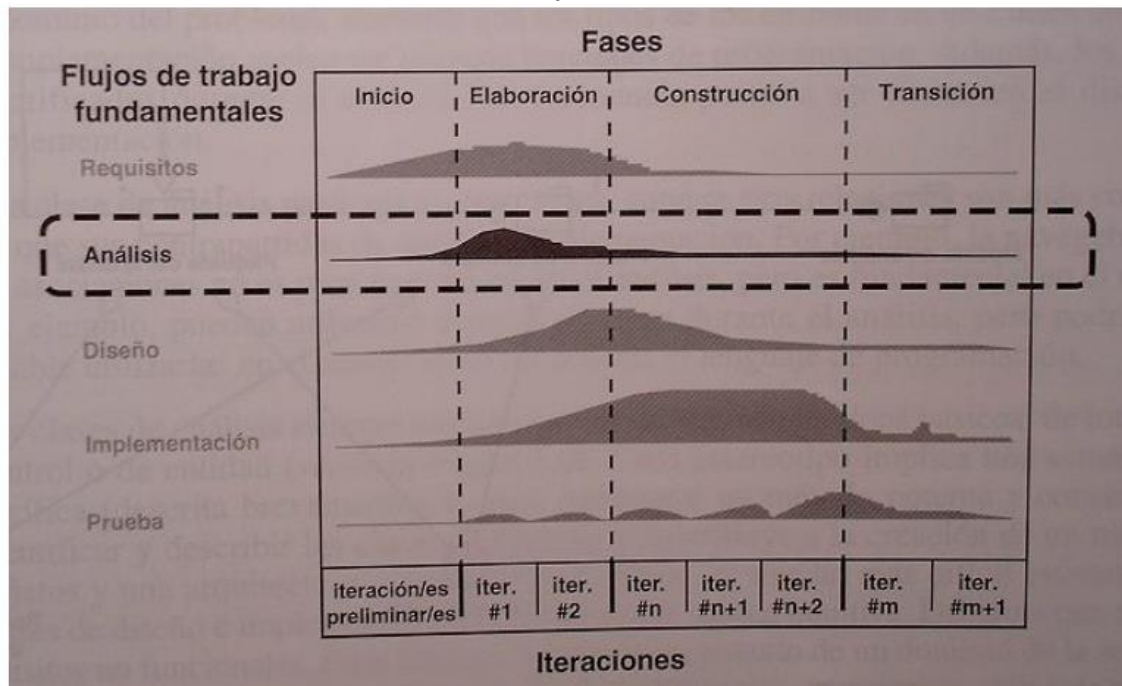
---

<sup>1</sup> **Actor:** cada tipo de usuario o sistema externo (de terceros) que interactúa con el sistema

a estructurar el sistema. En esta fase los requisitos se especifican de una forma más orientada al desarrollo.

- **Diseño:** se diseña el sistema y se modela para que soporte todos los requisitos. Una entrada importante al diseño es el análisis, ya que ofrece una visión más detallada y técnica de los requisitos. Los objetivos principales son la creación de un punto de partida para la implementación y la descomposición de los trabajos de implementación en fragmentos más manejables. En este caso contamos con diagramas que representan los diferentes módulos y la relación entre ellos.
- **Implementación:** comenzamos con el resultado del diseño e implementamos el sistema en términos de componentes, es decir, ficheros fuente, scripts, binarios, ejecutables, etc. Es el centro durante las iteraciones de implementación.
- **Prueba:** se verifica el resultado de la implementación probando cada construcción. Los objetivos de la prueba son los de diseñar y organizar las pruebas y llevar a cabo las necesarias en cada iteración – tanto de módulos simples como de integración -. Toda fase del RUP está sometida a pruebas, pero gozan de mayor importancia durante la construcción. Existen diversos tipos de pruebas: de caja negra, de estrés, de instalación, etc.

**Resumen** con la influencia de los diferentes flujos en cada fase:



La vida de un sistema según RUP es una serie de ciclos. Cada uno de estos consta de 4 **fases**:

- **Inicio:** se especifica la visión del proyecto. La idea inicial del desarrollo se lleva al punto de estar suficientemente bien fundamentada como para entrar en fase de elaboración.
- **Elaboración:** se definen la visión del producto y su arquitectura. Se expresan con claridad los requisitos del sistema para crear una sólida base arquitectónica. Asimismo, se planifican las actividades y recursos necesarios.
- **Construcción:** se construye el producto siguiendo una serie de iteraciones incrementales.
- **Transición:** el software se pone en manos de los usuarios.

Una vez finalizadas las 4 fases, al terminar cada ciclo, obtenemos una versión del sistema. Cada una de éstas versiones deber ser perfectamente funcional.

El modelo de casos de uso se expresa mediante los **diagramas de casos de uso**. Éstos muestran la relación entre dichos casos de uso y los actores.

Los **requisitos no funcionales** son aquellos que no pueden asociarse a ningún caso de uso. Por ejemplo, los **legales** (normativas), **físicos** (características físicas de un sistema, como la forma), de **implementación** (estándares, idiomas, etc.)...

Las principales **ventajas** del RUP son la tecnología de componentes y el modelo de proceso racional. Como **inconvenientes** cuenta con una dependencia muy grande del método y no incluye explícitamente actividades de gestión.

**Metodologías ágiles:** se caracterizan por priorizar al software funcionando frente a la documentación y la colaboración del cliente sobre el negocio. Este modelo es adaptable de forma incremental, necesita continua retroalimentación del cliente y se basa en la continua entrega de incrementos. Requiere ciertos requisitos por parte del equipo de desarrollo como pueden ser cierta autonomía, capacidad de colaboración y competencia técnica.

1. **XP:** es un modelo ágil que cuenta con 4 actividades estructurales cíclicas: diseño, codificación, prueba y evaluación. Se basa en pequeñas entregas y generalmente trabajo por parejas. También es normal que el cliente se encuentre en el lugar de desarrollo.
2. **Desarrollo Adaptativo de Software:** se basa en 3 actividades principales: especulación (análisis inicial), colaboración (recopilación de requisitos) y aprendizaje (desarrollo y prueba del incremento en conjunto).
3. **SCRUM:** se basa en equipos de trabajo organizados para maximizar la comunicación y minimizar gastos. Es un proceso adaptable a cambios sobre el producto. Se separa el trabajo en pequeños paquetes a llevar a cabo por cada grupo y posterior acoplamiento de los mismos. Incluye las actividades clásicas de requisitos, análisis, diseño, evolución y entrega.

Dentro de cada actividad las tareas se agrupan en **sprints**. Éstos consisten en rápidos desarrollos (30 días normalmente) de los módulos designados y acoplados al final de cada sprint para obtener nuevas funcionalidades en el producto.

Otra de las características principales de SCRUM es la realización de reuniones cortas muy frecuentes (15') para conocer el avance de cada grupo y los obstáculos encontrados.

Un miembro bajo el rol de maestro de SCRUM preside la reunión y evalúa las respuestas.

#### 4.- Gestión de proyectos

1. **Personal:** es importante contar con personal cualificado y motivado. En un proyecto se distinguen varios tipos de personal:
  - a. **Gestores funcionales superiores:** definen las reglas de negocio.
  - b. **Gestores técnicos del proyecto:** deben planificar, mover, motivar y organizar a los desarrolladores.
  - c. **Profesionales o trabajadores**
  - d. **Clientes:** especifican los requisitos del proyecto
  - e. **Usuarios finales:** interactúan con el software entregado

El **jefe de equipo**, que puede ser un gestor superior o técnico, es el encargado de liderar al equipo. Es vital que tenga buenas dotes de organización, motivación, conocimiento de tecnologías, etc.

El **jefe de personal** se encarga de elegir el personal destinado al desarrollo en base a los conocimientos, logros, experiencia, etc.

Existen diversos **modelos de organización para un equipo** de desarrollo, a continuación presentamos los 3 propuestos por *Mantei*:

- **Centralizado controlado (CC)**: existe un jefe de equipo único que resuelve todos los problemas y coordina al resto. Existe comunicación vertical entre ambos niveles.
- **Descentralizado controlado (DC)**: existe un jefe para coordinar las tareas principales y varios jefes secundarios para las distintas sub tareas. Hay comunicación horizontal entre los grupos y vertical entre los jefes secundarios y el principal.
- **Descentralizado democrático (DD)**: el jefe principal se asigna en función de la tarea concreta a desarrollar. Existe mayor comunicación y consenso con el grupo en general.

Hay varios factores que nos permitirán decantarnos por un modelo u otro: dificultad del problema, modularización, tamaño estimado de LDC, fecha de entrega, comunicación, etc.

2. **Producto**: el primer factor a determinar para el desarrollo es el **ámbito del software**. Esto es, el contexto junto con otros posibles sistemas, los objetivos de la información de entrada y salida y las funcionalidades requeridas.  
A continuación se procede a una **partición horizontal** en varios módulos.  
Tras esto, se lleva a cabo una nueva **partición vertical** en la cual se divide el sistema en funciones. Cada una de estas representa una transformación invocable por el usuario que transforma datos de entrada en salida.
3. **Proceso**: las AE se dividen en acciones de IS, que son agrupaciones de tareas. Para que estas tareas sean llevadas a cabo de forma efectiva es necesaria una **planificación temporal**. La **Estructura de Descomposición del Trabajo (EDT ó WBS en inglés)** hace referencia a la asignación de tareas a los individuos en plazos de tiempo acotados. Según Pressman, dicha EDT se puede representar empleando las acciones columnas (bajo las AEs) y las funciones como filas (al lado del módulo asociado). En cada casilla se colocan las tareas y personal.

AE	AE1				AE2				.....		AEn				
A	a1.1	a1.2	....	a1.m1	a2.1	a2.2	....	a2.m2	....	....	....	an.1	an.2	....	an.m
M1															
f1.1															
f1.2															
....															
f1.i1															
M2															
f2.1															
f2.2															
....															
f2.i2															
....										tarea <sub>1</sub> - rrhh - fecha inicio - fecha fin - entregables					
Mk															
fk.1															
fk.2															
....															
fk.ik															

4. **Proyecto:** para gestionar un proyecto debemos ser capaces de darnos cuenta de qué puede ir mal. Para ello hay varias señales de peligro como: no conocer las necesidades del cliente, ámbito del producto definido pobremente, cambios mal realizados, fechas de entrega poco realistas, quejas de usuarios o falta de técnicas y prácticas de IS. Por ello, para una correcta gestión, es importante hacer se preguntas como: ¿por qué se desarrolla el sistema?, ¿qué se realizará y cuándo?, ¿qué cantidad de recursos se necesitan para cada módulo? o ¿cómo se realizará el trabajo desde un punto de vista técnico y de gestión?

## 5.- Métricas

Una **medida** proporciona una indicación cuantitativa de la extensión, cantidad, dimensiones, capacidad o tamaño de algunos atributos de un proceso o producto. Por ejemplo, LDC.

La **medición** es el acto de determinar una medida.

Una **métrica** es una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado. Por ejemplo: LDC/pm.

Las medidas no sirven para comparar, necesitamos métricas para ello.

Las métricas son el fundamento de los indicadores. Un **indicador** es una métrica o combinación de métricas que proporcionan una visión profunda del proceso del software, del proyecto de software o del producto en sí. Por ejemplo, pasar de 500 LDC/pm a 250 LDC/pm.

Las métricas de I.S. **sirven** para:

- Ayudar en la estimación, en el control de calidad y en toda la gestión de un proyecto
- Mejorar el propio proceso de desarrollo
- Evaluar la calidad del producto realizado
- Controlar la productividad del personal

1. **Métricas de proyecto:** bastante ligadas a las de proceso. Permiten evaluar el estado del proyecto en curso, seguir la pista de riesgos potenciales o determinar ciertas áreas problemáticas y ajustar el flujo de trabajo. Podemos dividir esta sección en varios tipos de métricas:

- a. **Del proceso:** ayudan a evaluar y mejorar el proceso. Algunos ejemplos son el índice de defectos (tras el proceso), número de errores (durante el proceso), LDC, PF, etc.

- b. **Del software:** algunas valen también para el producto. En la tabla se reflejan la mayoría de ellas. Por ejemplo, la ventaja de las **LDC** (orientadas al tamaño) es que son fáciles de calcular y actualmente hay varias herramientas para ello. El problema es que no están comúnmente aceptadas y dependen mucho del tipo de software y lenguaje empleado. En cuanto a las métricas orientadas a función se obtienen considerando las medidas de productividad y normalizándolas por la funcionalidad, que se debe derivar de otras medidas directas. Por ejemplo, los **puntos de función** se obtienen con la expresión:  $PF = total * (0.65 + 0.01 * \sum Fi)$ . Donde total es se calcula en función de 5 parámetros (número de entradas de usuario, salidas, peticiones, archivos e interfaces externas) y Fi es un valor de ajuste de complejidad que se obtiene de responder a una serie de preguntas valorándolas entre 0 y 5. La medida de **punto de característica** (PC) es una ampliación de lo anterior usado en sistemas con fuerte componente funcional (p. ej. Tiempo real). La ventaja de PF y PC es que son independientes de lenguajes de programación, aunque tampoco son comúnmente aceptados y están basados en cálculos subjetivos. En cuanto a **métricas de calidad**, tenemos las de errores, corrección de defectos, eficacia de la eliminación de defectos ( $EED \rightarrow e / (e+d)$ ), etc.

MÉTRICAS DEL SOFTWARE	Productividad	Calidad
Tamaño	$\frac{\text{euros}}{\text{LDC}} \frac{\text{pgDoc}}{\text{KLDC}}$	$\frac{\text{errores}}{\text{KLDC}} \frac{\text{defectos}}{\text{KLDC}}$
PF (PF, PC, PF3D)	$\frac{\text{euros}}{\text{PF}} \frac{\text{pgDoc}}{\text{PF}}$	$\frac{\text{errores}}{\text{PF}} \frac{\text{defectos}}{\text{PF}}$
Otras	$\frac{\text{LDC}}{\text{per-mes}} \frac{\text{PF}}{\text{per-mes}} \frac{\text{euros}}{\text{pgDoc}}$	$\frac{\text{errores}}{\text{per-mes}} \frac{e}{e+d}$

## 2. Métricas del producto:

- Métricas para el modelo de análisis:** basadas en las métricas de proyecto.
- Métricas para el modelo de diseño:** por ejemplo, la medición de la complejidad de cada módulo en cuanto a relaciones con otros o datos manejados, tamaño de arcos y nodos del grafo de dependencia entre módulos, mediciones sobre los objetos que componen un sistema (complejidad, suficiencia, integración, etc.), medidas de herencia o medidas de operaciones y atributos de clases, etc.
- Métricas para pruebas:** como por ejemplo métricas de herencia (clases raíz/hijos/profundidad/padres o hijos directos), encapsulación, porcentaje público-privado (PPP), etc.
- Métricas para mantenimiento:** por ejemplo el índice de madurez del software (IMS) que, a grandes rasgos, se calcula teniendo en cuenta los módulos originales y los generados tras una ampliación (al igual que los eliminados).

## 6.- Estimaciones en proyectos software

Las estimaciones las lleva a cabo el gestor del proyecto. Requieren experiencia y confianza en las métricas. Hay cuatro **factores** principales que influyen en las estimaciones realizadas sobre un proyecto: el tamaño del mismo, la complejidad (relativa a proyectos anteriores), el grado de incertidumbre estructural (a mejor conocimiento, mayor rapidez de división y estimación) y la disponibilidad de información histórica.



El **objetivo** de la planificación es permitir al gestor organizar tareas, costes, plazos y recursos de forma efectiva. Estas estimaciones se llevan a cabo al principio del proyecto. Además, conviene definir escenarios del mejor caso y el peor.

Las estimaciones tienen varias **actividades asociadas**:

- **Ámbito del software**: comunicación continua con el cliente para definir los requisitos del software a desarrollar así como mantener actualizados los posibles cambios. Es importante conocer perfectamente la idea que el cliente tiene acerca del producto y mantener una continua comunicación y *feedback*. Durante las primeras reuniones es muy importante conocer la motivación del proyecto, los requisitos de la forma más precisa posible y toda la información relevante.
- **Recursos** : la segunda actividad es la estimación de recursos. Esto incluye tanto personal, como hardware o software. Es importante definir el tipo de recurso necesitado y el momento y duración del mismo (ventana temporal).

Pasaremos ahora a ver las diferentes **estimaciones** en proyectos software:

1. **Técnicas de estimación**: las estimaciones nunca será exacto dado que los requisitos cambian, los grados de competencia de los componentes de los equipos son diferentes, etc. Una posible técnica es la de **retrasar** todo lo posible la estimación. A mayor retraso mayor precisión. Otra sería la de **comparar** el proyecto con otros ya realizados o la **Ley de Parkinson** (extender el trabajo para rellenar todo el tiempo disponible).
2. **Técnicas de descomposición**: por un lado podemos llevar a cabo la estimación **basada en el problema**. Esto es, la descomposición del software en funciones y estimar el tamaño del mismo (basado en LDC o PF). La precisión de este método depende ante todo del grado en el que el planificador ha estimado correctamente el tamaño del software, la habilidad para traducir el tamaño en esfuerzo y dinero o la estabilidad de los requisitos y el entorno de la IS.  
Para llevar a cabo esto primero se hace un análisis de LDC o PF y a partir del mismo se generan las estimaciones.  
Otro tipo de **técnicas de descomposición** son aquellas basadas en el proceso. En este caso se realiza la estimación a partir del proceso a emplear, del cual obtendremos una serie de tareas y estimaremos el coste y esfuerzo de las mismas. En este caso primero se genera la EDT y a partir de ella se calculan las estimaciones.  
*NOTA: ejemplos de ambas en diapositivas 7, página 39 en adelante.*
3. **Modelos paramétricos de estimación**: se basan en fórmulas empíricas para predecir el esfuerzo como una función de LDC o PF. La validez se restringe únicamente al entorno donde se dedujo la fórmula. El modelo general es:  $Esfuerzo = A + Bx^C$  ( $x = PF/LDC$ ). Hay fórmulas como las de Boehm o Doty (orientadas a LDC) o la de Kemerer (orientada a PF).  
Otra posibilidad en este campo es el uso de herramientas automáticas para el cálculo,

que reciben como entrada las LDC o PF y ofrecen como salida los costes, duración y esfuerzo.

Por último, mencionar que en ocasiones puede no ser rentable la construcción de determinados módulos del software desde cero. En estos casos hay que analizar con cautela la posibilidad de comprar software de terceros o subcontratar nuevas empresas de IS.

## 7.- Ingeniería de requisitos

Es la ingeniería que permite identificar los servicios y restricciones (requisitos) de un sistema. Los **requisitos de usuario** son frases en lenguaje natural o diagramas que el sistema debe ofrecer así como las restricciones bajo las que debe operar. Los **requisitos de sistema** determinan los servicios del mismo y las restricciones en detalle (sirve como contrato). Ambas muestran lo mismo pero a distintos niveles de detalle. Por ejemplo, en el primer caso sería “el sistema debe hacer préstamos” y en el segundo “función: préstamo; entrada: cód. 1, salida:...”. Existen tres **tipos de requisitos** de sistema:

1. **Funcionales:** describen los servicios (funciones) que ofrece el sistema, la respuesta ante ciertas entradas y el comportamiento en situaciones particulares.  
*Ejemplo como en caso de uso: precondition, función, entradas, salidas, etc.*
2. **No funcionales:** son restricciones de los servicios o funciones que ofrece el sistema. Por ejemplo: una búsqueda no debe tardar más de 10 segundos.  
Hay 3 tipos de requisitos no funcionales: del producto (prestaciones, memoria, etc.), organizativos (lenguajes de programación, etc.) y externos (legislativos, éticos, etc.).
3. **Del dominio:** se derivan del dominio de la aplicación y reflejan las características de ese dominio. Pueden ser a su vez funcionales o no funcionales. Por ejemplo: el sistema de biblioteca de la UCM debe ser capaz de exportar datos en el formato de bibliotecas de España (LIBE).

La ingeniería de requisitos debe centrarse principalmente en la aplicación que hay que desarrollar y no en el cómo. Tiene varias **fases**:

1. **Estudio de viabilidad:** estimación de si se puede resolver el problema con los recursos disponibles y si sale rentable.
2. **Obtención y análisis de requisitos:** interacción con los interesados para determinar el dominio de la aplicación, requisitos funcionales y no funcionales. Es una tarea compleja ya que estos usuarios/clientes no tienen por qué conocer la jerga informática.
3. **Especificación de requisitos:** se fijan los requisitos tomados anteriormente de una forma mucho más formal y de manera que sirvan como contrato entre cliente y desarrollador. En este punto se realiza la Especificación de Requisitos Software.
4. **Validación de requisitos:** revisar si los requisitos se ajustan a los deseos de los interesados (*stakeholders*). Hay que comprobar la validez, la consistencia, completitud, realismo, etc.

Dado que los requisitos pueden cambiar es necesaria la **gestión de requisitos** como proceso para entender y controlar los cambios sobre los requisitos. Por ello, debemos ser capaces de identificar los requisitos (ligado a la ERS), tener procesos de gestión del cambio y disponer de políticas de traza (saber si el requisito está ligado a un interesado, a otro requisito, etc.).

Dada la importancia de este punto, existe un estándar **IEEE para** el desarrollo de documentos **ERS**. Éste estipula que este documento debe ser redactado durante la fase de análisis y que deben recogerse las siguientes características de funcionalidad, interfaces externas, prestaciones, atributos (mantenibilidad, portabilidad, etc.) y restricciones de diseño. Por supuesto, dicho documento debe estar redactado de forma coherente, trazable, modificable, verificable, etc.

## 8.- Planificación temporal de proyectos software

En un proyecto hay tareas más importantes que otras. Algunas son críticas y producen retrasos en el proyecto en caso de retrasarse. El objetivo del gestor es identificar las tareas (prestando especial atención a las críticas) y hacer un seguimiento de las mismas para realizar una **planificación temporal**.

La planificación temporal es una actividad que distribuye el esfuerzo estimado a lo largo de la duración prevista del proyecto, asignando el esfuerzo a las tareas de trabajo concretas. No es estática sino que cambia con el tiempo. Para llevar a cabo una buena planificación es importante identificar claramente las dependencias entre tareas, asignar correctamente los esfuerzos a cada una de ellas y asociarlas a hitos concretos del proyecto.

Es importante tener en cuenta que, a mayor **número de personas**, no obtendremos mayor rendimiento ya que entre ellas se abrirán cada vez más canales de comunicación. Esto provoca que se reduzca la productividad. Por ejemplo, al introducir nuevas personas en un desarrollo ya comenzado, se producirá una pérdida de productividad debida a la necesidad de instrucción y adaptación de éstas.

Otro aspecto importante es la **distribución de esfuerzos**. Una buena teoría es la del 40 – 20 – 40. 40% en análisis y diseño, 20% en codificación y 40% en pruebas. No obstante, variaciones en ciertos márgenes sobre esto también producirán buenos resultados.

Por otro lado, se puede llevar a cabo también una **descomposición de las acciones** para obtener un número de tareas variable. Estas tareas pueden seleccionarse o reorganizarse en función de las características del proyecto.

Hay que distinguir también entre las planificaciones **macroscópicas** (como lo anterior) y **microscópicas**, que se obtienen del refinamiento de las tareas principales. No suele llevarse a cabo ya que está implícita en los conocimientos de los desarrolladores y suele sobrecargar demasiado la planificación general.

Un posible modelo de representación de planificaciones pasa por el uso de **diagramas de Gantt** (que lleva la lista de tareas en la columna principal y los tiempos en la final principal).

También es posible una representación mediante una **red de tareas**, que es un grafo topológico de tareas dependientes. En este caso, suele marcarse en rojo el camino crítico.

Una vez generada la planificación, y durante el desarrollo del proyecto, es importante también llevar a cabo un **seguimiento y actualización** de la misma. Esto se puede lograr con reuniones periódicas, evaluando resultados de las revisiones, etc. Existen técnicas para el seguimiento del progreso como el análisis de costes empleados (no muy fiable) o la técnica del **valor ganado** (que lleva estadísticas de los esfuerzos y fechas y las relaciona).

Por último, es interesante también la realización de un **plan de proyecto software**. Este documento no debe ser especialmente largo y debe contener los aspectos más relevantes de qué se va a hacer y cómo y cuándo.

## 9.- Riesgo, configuración y calidad

### Riesgos

**Riesgo** es todo aquello susceptible de afectar negativamente al proyecto software. Está claro que hay unos más graves que otros.

Frente a ellos podemos tomar una **estrategia reactiva** (actuar una vez se produce el problema para sofocarlo – mala idea - ) o **proactiva** (generar un plan de identificación, priorización y contingencia contra riesgos antes del desarrollo).

Hay diferentes **tipos de riesgos**:

- **Del proyecto**: amenazan al plan del proyecto. Identifican problemas potenciales en el presupuesto, planificación, personal, recursos, etc.
- **Técnicos**: amenazan a la calidad del software. Aparecen porque el software puede ser más difícil de lo esperado. Detectan problemas en el diseño, implementación, requisitos, etc.
- **Del negocio**: amenazan a la viabilidad del proyecto. Por ejemplo la creación de software sin posibilidades de mercado, pérdida de inversores, etc.

Otra posible clasificación es:

- **Conocidos**: aquellos de los que el personal es consciente.
- **Desconocidos**: aquellos de los cuales el personal sería consciente si se siguiese un proceso de identificación.
- **Impredecibles**: aquellos que, en principio, no eran esperables.

En cuanto a la **gestión de riesgos**, para llevarla a cabo se siguen una serie de **pasos**:

#### 1. Valoración del riesgo

- a. **Identificación**: produce listas de riesgos que comprometan seriamente el éxito del proyecto. Existen tablas aceptadas internacionalmente para esta identificación.

- b. **Análisis:** determina la probabilidad y consecuencias asociadas a cada riesgo. Las probabilidades pueden estimarse o bien fijarse a partir de ciertas tablas de ayuda estándar.
- c. **Priorización:** produce una lista ordenada de elementos de riesgo identificados y analizados. Una posible implementación es la creación de una tabla de riesgo, que los ordena primero por probabilidad y segundo por consecuencia. Otra posible opción es la de emplear tablas como la inferior para categorizar cada uno de los posibles riesgos:

Probability Severity	Frequent	Probable	Occasional	Remote	Improbable
Catastrophic	IN	IN	IN	H	M
Critical	IN	IN	H	M	L
Serious	H	H	M	L	T
Minor	M	M	L	T	T
Negligible	M	L	T	T	T
LEGEND	T = Tolerable	L = Low	M = Medium	H = High	IN = Intolerable

## 2. Control del riesgo

- a. **Planificación de la gestión:** convierte la información sobre riesgos en decisiones y acciones para el futuro. Incluye la creación de un Plan de gestión de Riesgos. Se analiza la lista de riesgos y se decide si se evita el riesgo, si se mitiga, si se ignora o si se comparte. También deben proponerse planes de contingencia por si los riesgos se hacen reales.
- b. **Resolución del riesgo:** aquí se llevan a cabo los pasos para reducir y controlar los riesgos.
- c. **Monitorización:** se comprueba si se están llevando a cabo los pasos para la reducción del riesgo, si éste se está haciendo real (mediante métricas) y se toman decisiones correctivas.

La información esencial sobre riesgos puede incluirse en un Plan de Reducción, Supervisión y gestión de Riesgos (RSGR). Así, durante la **reducción** se aplican medidas para evitar que el riesgo se haga real, en la **supervisión** se monitoriza si el riesgo se ha hecho real y en la **gestión** se aplican las medidas para disminuir el riesgo.

## Configuración software

Al crear software los cambios son inevitables y aumentan el nivel de confusión del equipo. La Gestión de la Configuración Software es una actividad protectora encargada de

gestionar el cambio a lo largo del ciclo de vida del software.

Los elementos que componen toda la información generada como el proceso de IS (programas, datos y documentos) componen la denominada **configuración del software**.

Encontramos **fuentes fundamentales de cambio** en elementos como: fallos, nuevos negocios o condiciones, modificaciones presupuestarias o de recursos, etc.

Por culpa de estos cambios, es conveniente el establecimiento de una **línea base**, que es un concepto de GCS que nos ayuda a controlar los cambios sin perjuicio de aquellos que no sean afectados. Se define como una especificación que se ha revisado formalmente y que sirve de base para un desarrollo posterior y puede ser cambiada únicamente por procesos formales de control de cambios.

Cualquier **GCS debe** tener claro cómo identificar y gestionar las versiones de un programa para permitir modificaciones, quién es el responsable de gestionar los cambios, cómo garantizar que se han llevado a cabo de forma apropiada y qué mecanismos se emplean para avisar de los cambios.

Las **actividades** de la GCS son:

- **Identificación de elementos de configuración software (ECSs):** para ello se puede seguir un modelo enfocado a objetos. Tenemos objetos simples (unidades de texto como la ERS, un listado de código, etc.) y compuestos (conformados por varios objetos simples). Estos elementos tienen una serie de características como un nombre, descripción, etc. Además pueden organizarse en diagramas UML de relaciones entre ellos.
- **Control de versiones:** permite gestionar la versión del sistema. A su vez, ésta viene determinada por la versión de los ECSs. Puede haber variantes de las versiones si, por ejemplo, tenemos un sistema multiplataforma.
- **Control de cambios:** todo cambio produce caos. Por ello es necesario un mecanismo formal y eficiente para llevar a cabo los cambios.
- **Auditoría de la configuración:** se ocupa de han hecho los nuevos cambios, si se han seguido adecuadamente los estándares de la IS, se han reflejado los cambios en los ECSs, etc.
- **Informes de estado:** informan acerca de qué pasó, quién lo hizo y cuándo. Se deben generar para mantener informados acerca de los cambios.

## **Garantía de calidad del software**

Uno de los principales objetivos de la IS es la construcción de software de calidad. Hay dos **tipos** de calidad: de **diseño** (cumplimiento de las ERS y diseño) y de **concordancia** (grado de cumplimiento de las especificaciones durante la implementación).

Hay una serie de conceptos importantes sobre calidad:

- Definimos **calidad como concordancia** con requisitos funcionales y de rendimiento claramente establecidos, estándares de desarrollo bien documentados, etc.
- El **control de calidad** es una serie de inspecciones, revisiones y pruebas utilizadas a lo largo del proceso de software para asegurar que se cumplen una serie de requisitos.
- La **garantía de calidad** es el establecimiento de un marco de procedimientos organizativos que llevan a establecer una alta calidad de software.
- El **coste de calidad** incluye todos los costes derivados de la búsqueda de calidad y obtención de la misma. Pueden ser por prevención, evaluación o por fallos antes y después de la entrega final.
- El **equipo de calidad** se encarga del establecimiento de un plan de calidad para el proyecto (SQA), participar en el desarrollo de la descripción del proceso software, revisión de las actividades de ingeniería, evaluación de los productos software, registrar todo lo que no se ajuste a los requisitos y coordinar el control y gestión de cambios.

Asociado al tema de calidad encontramos la **Verificación y Validación (V&V)**, que son los procesos que se encargan de determinar el ajuste del producto de una determinada actividad a los requisitos especificados para la misma. La verificación se encarga de comprobar si estamos construyendo el producto correctamente y la validación de si estamos construyendo el producto correcto.

Dentro del proceso V&V existen dos aproximaciones complementarias para el **análisis**:

- **Revisiones del software** (estáticas): son un filtro para el proceso de IS. Purifican las actividades estructurales. Nos centraremos en las **Revisiones Técnicas Formales (RTFs)** cuyo objetivo es detectar errores antes de que se conviertan en defectos. La RTF se lleva a cabo mediante una reunión formal. Esta RTF se centra específicamente en determinadas partes software. En las reuniones se evalúan los productos resultantes de las tareas de trabajo. Para ello se designan una serie de revisores que, tras la exposición del producto por parte del productor, exponen las pegas al mismo. Al final de la reunión se deciden las medidas a tomar en función de todo lo discutido. También es importante dejar bien documentados todos los aspectos más relevantes de estas reuniones.
- **Pruebas del software** (dinámicas).

La **garantía de calidad estadística** agrupa y clasifica la información sobre los fallos del software, intenta encontrar la causa subyacente a cada fallo y actúa para corregir fallos vitales. Otra métrica interesante es el **índice de errores**, que permite cuantificar la magnitud de fallos durante el proceso de desarrollo.

En cuanto a la **fiabilidad del software**, puede ser medida teniendo en cuenta la ausencia o presencia de fallos. Existen mediciones como la **probabilidad de fallo bajo demanda**, que mide la probabilidad de que un sistema falle cuando se le solicita un determinado servicio. Otra es la

**frecuencia de fallo**, que es la frecuencia de aparición de fallos durante la ejecución del sistema. También es interesante medir el **tiempo medio de fallo**, que mide el tiempo transcurrido de media entre fallos del sistema.

Por último, mencionar que existen diversos enfoques de calidad como los ISO 9000 o el español AENOR.

## 10.- El modelo de objetos

La finalidad de esta sección es mostrar el modelo basado en objetos más allá de la POO. Hay una serie de **conceptos clave** para comprender los sistemas complejos:

- **Descomposición:** durante el diseño es necesario descomponer el sistema en partes más pequeñas, y hacer lo mismo con éstas hasta llegar a las partes primitivas.
- **Abstracción:** sirve para enfrentarse a la complejidad. Para ello ignoramos los detalles de una entidad trazando sólo un modelo generalizado e idealizado.
- **Jerarquía:** caracteriza las relaciones entre los subsistemas generados durante la descomposición. Podemos distinguir entre una **estructura de objetos** (empleando relaciones de colaboración entre componentes –parte de-) y **estructura de clases** (-es un-). La arquitectura de un sistema se compone tanto de la estructura de clases como la de objetos.

En cuanto al **diseño de sistemas complejos**, tiene como objetivo determinar un modelo del sistema completo. La ventaja fundamental es poder atacar el problema del desarrollo desde una perspectiva más amigable.

La **Programación Orientada a Objetos** es un modelo de programación basada en la organización de los programas como objetos que interaccionan entre sí.

Hay cuatro **elementos básicos en el modelo de objetos**:

- **Abstracción:** denota las características principales de un objeto que lo diferencian del resto. Con ella se llega a un modelo contractual de programación, en el que cada objeto está caracterizado por los contratos que ofrece a otros objetos y que a su vez pueden ser llevados a cabo con contratos con otros objetos.
- **Encapsulación:** es complementaria a la abstracción, solo que se centra en la implementación. Se logra mediante el ocultamiento de la información.
- **Modularidad:** en los programas OO las clases y objetos forman la estructura lógica de un sistema. Estas abstracciones se sitúan en la arquitectura para generar los módulos de un sistema. En Java por ejemplo se usan los *packages*. La cohesión mide la solidez de las relaciones de un módulo con sus componentes y el acoplamiento la fuerza de interconexión entre módulos.
- **Jerarquía:** es una clasificación u ordenación de las abstracciones. Por ejemplo con herencia o agregación.



- **Tipos:** son caracterizaciones precisas de propiedades estructurales o de comportamiento que comparten una serie de entidades.
- **Concurrencia:** ejecución de un programa en varios hilos simultáneamente.
- **Persistencia:** un programa ocupa una cantidad de espacio y existe durante un determinado periodo de tiempo.

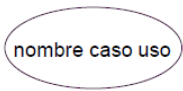


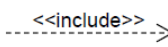
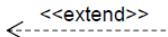
Por último, algunas **definiciones adicionales:**

- **Objeto:** elemento que tiene un estado, comportamiento (funcionalidad) e identidad (propiedad que lo distingue de los demás).
- **Relaciones entre objetos:** pueden ser **enlaces** (conexiones físicas o conceptuales entre objetos) o **agregaciones** (relaciones *cliente-servidor*).
- **Relaciones entre clases:** herencia, asociación (relación uno a uno), agregación (el ciclo de vida de A no está relacionado con el de B, relación uno a muchos, por ejemplo un array de elementos), dependencia y metaclases (cuyas instancias son clases).


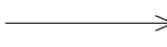

## 11.- Lenguaje Unificado de Modelado (UML)

A continuación se presenta un breve conjunto de elementos de este lenguaje.

### Casos de uso

- Caso de uso: 
- Actor: 
- Relación:
  - Generalización: 
  - Inclusión: 
  - Extensión: 

### Diagramas de flujo

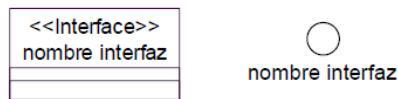
- Estado: 
- Transición: 
- Estado inicial: ●
- Estado final: ◎
- Bifurcación: 

## Diagramas de clases

– Clases:



– Interfaces:



– Relaciones:

- Generalización:
- Asociación:
- Agregación:

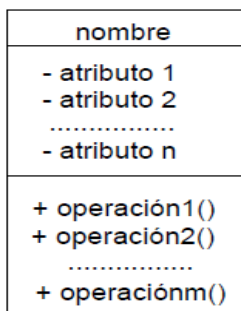
• Dependencia:

• Instanciación:

• Metacalse:

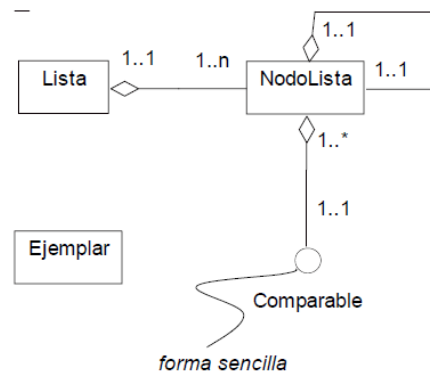
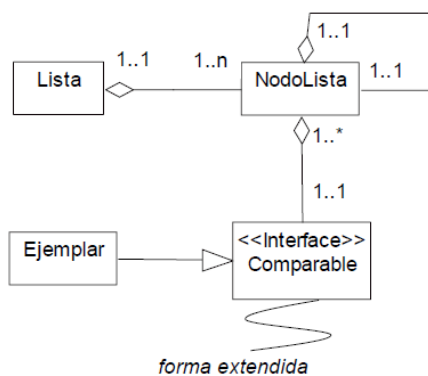
• Realización:

## Clases



- Public: +
- Protected: #
- Private: -

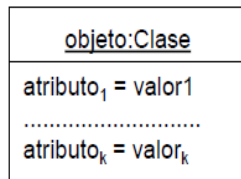
## Icono de clase



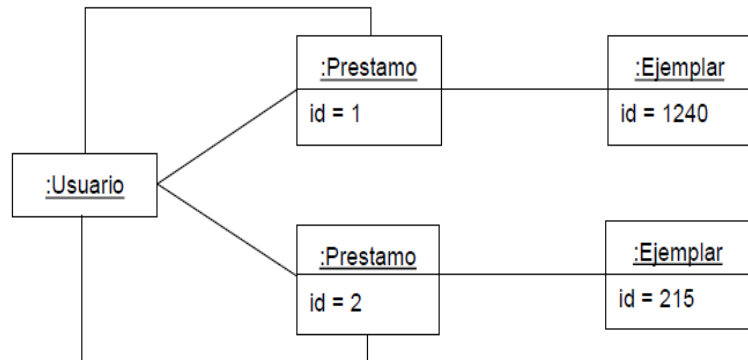
## Ejemplo de realización de un interfaz

## Diagramas de objetos

– Objeto:



– Enlace:



## Diagramas de secuencia

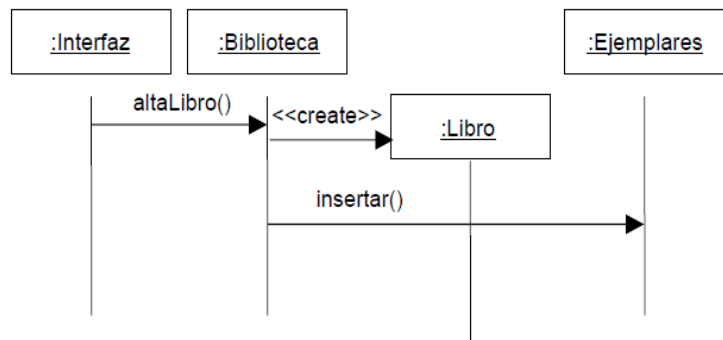
– Objeto:



– Mensaje síncrono:  $\xrightarrow{f()}$

– Retorno:  $\xleftarrow{\hspace{1cm}}$

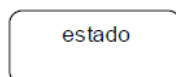
– Mensaje asíncrono:  $\xrightarrow{s}$



Alta de libro en la biblioteca

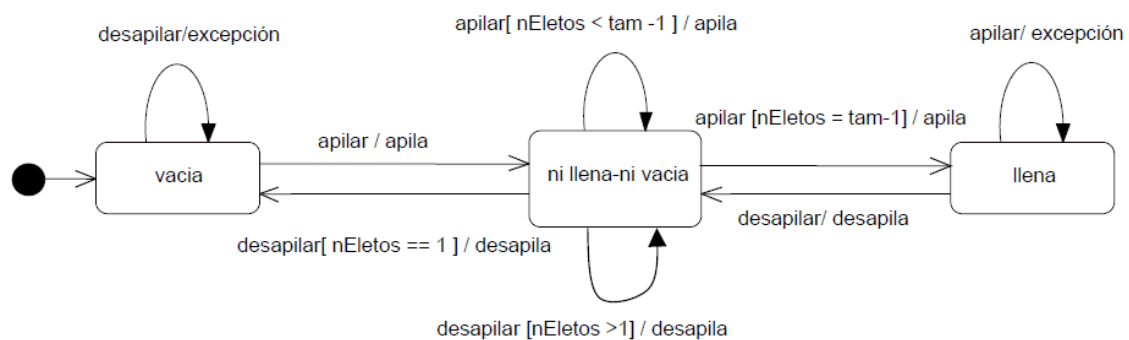
## Diagramas de estados

– Estado:

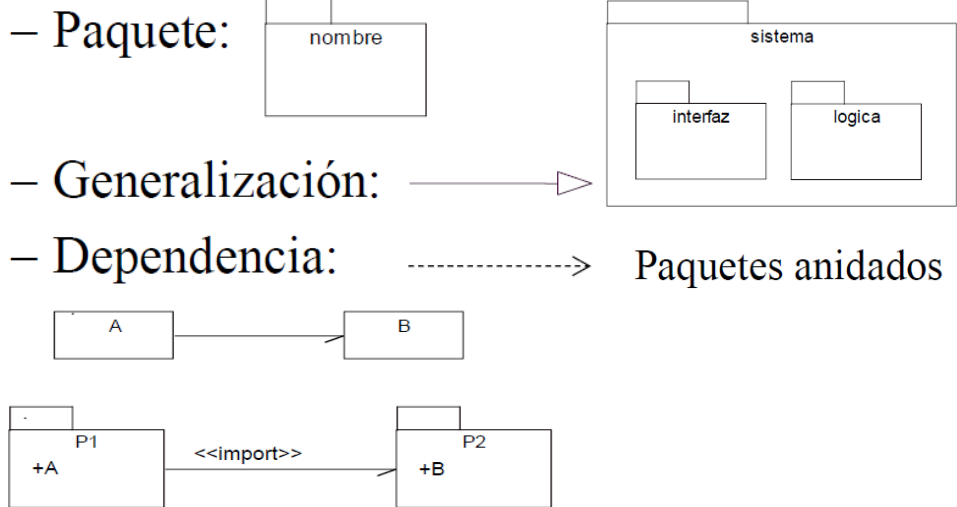


– Estado inicial: ●

– Transición:  $\xrightarrow{\text{evento[condición]/acción}}$



## Paquetes



## Importación entre paquetes

**Importante:** A partir del año 2013-2014 usaremos como referencia de UML las fichas que aparecen aquí: <http://www.holub.com/goodies/uml/>

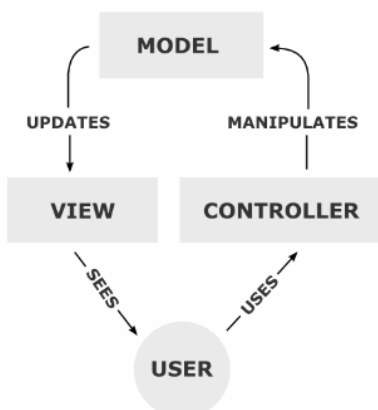
### 12.- Teoría segundo cuatrimestre

Aunque puede que en el primer cuatrimestre se vean cuestiones de arquitectura del software, patrones arquitectónicos, etc.

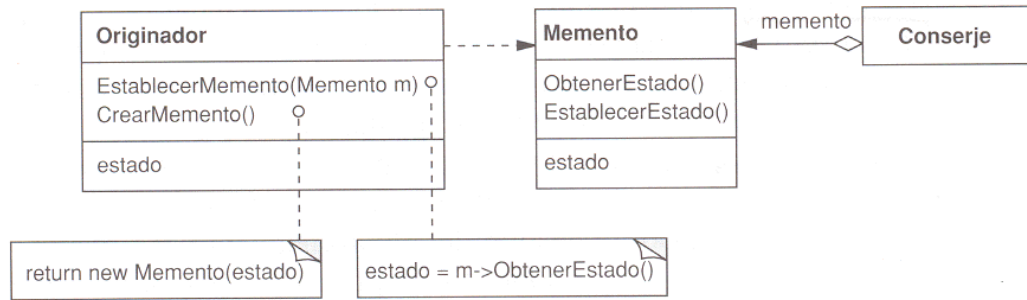
### 13.- Patrones de diseño

- **Modelo-Vista-Controlador:** Más que un patrón de diseño software, es un patrón arquitectónico (de más alto nivel). Este “patrón” separa los datos y la lógica del negocio de la interfaz de usuario y define una arquitectura de diseño para el sistema centrada en 3 elementos principales:
  - **Modelos:** son la representación específica de la información con la cual el sistema opera, por lo tanto gestionan todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones. Envía a la 'vista' aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'.
  - **Vistas:** presentan el 'modelo' (información y *lógica de negocio*) en un formato adecuado para interactuar (usualmente la interfaz de usuario) y, por tanto, requieren de dicho 'modelo' la información que debe representar como salida.

- **Controladores:** responden a eventos (usualmente acciones del usuario) e invocan peticiones al 'modelo' cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una base de datos). También pueden enviar comandos a su 'vista' asociada si se solicita un cambio en la forma en que se presenta de 'modelo' (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos), por tanto se podría decir que el 'controlador' hace de intermediario entre la 'vista' y el 'modelo'
- **Singleton:** este patrón consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.  
Se implementa creando en nuestra clase un método que genera una instancia de la clase si no existe una ya para garantizar su unicidad. Además, se restringe el acceso a la constructora haciendo uso de limitadores como *protected* o *private*. Tiene como ventaja principal la reducción del consumo de recursos y se emplea mucho en secciones donde se debe acceder a un recurso único (ratón, fichero en modo exclusivo, etc.).
- **Memento:** su finalidad es la de almacenar el estado de un objeto (o del sistema completo) en un momento dado de manera que se pueda restaurar en ese punto de manera sencilla. Para ello se mantiene almacenado el estado del objeto para un instante de tiempo en una clase independiente de aquella a la que pertenece el objeto (pero sin romper la encapsulación), de forma que ese recuerdo permita que el objeto sea modificado y pueda volver a su estado anterior.  
Para implementarlo se hace uso de 3 clases principales:
  - **Memento:** permite almacenar (parte de) el estado de Creador. Ofrece dos interfaces distintas: una estrecha para el conserje (*Caretaker*) y una extendida para *Creador*.
  - **Caretaker (conserje):** se encarga de mantener los objetos Memento y en ningún caso accede a su información interna.
  - **Originator (creador):** es capaz de crear objetos Memento con información sobre su estado y restaurar este a partir de un Memento.



Un ejemplo de uso es para pilas de acciones, por ejemplo de un editor de texto, donde nos interesa volver a puntos anteriores o posteriores.



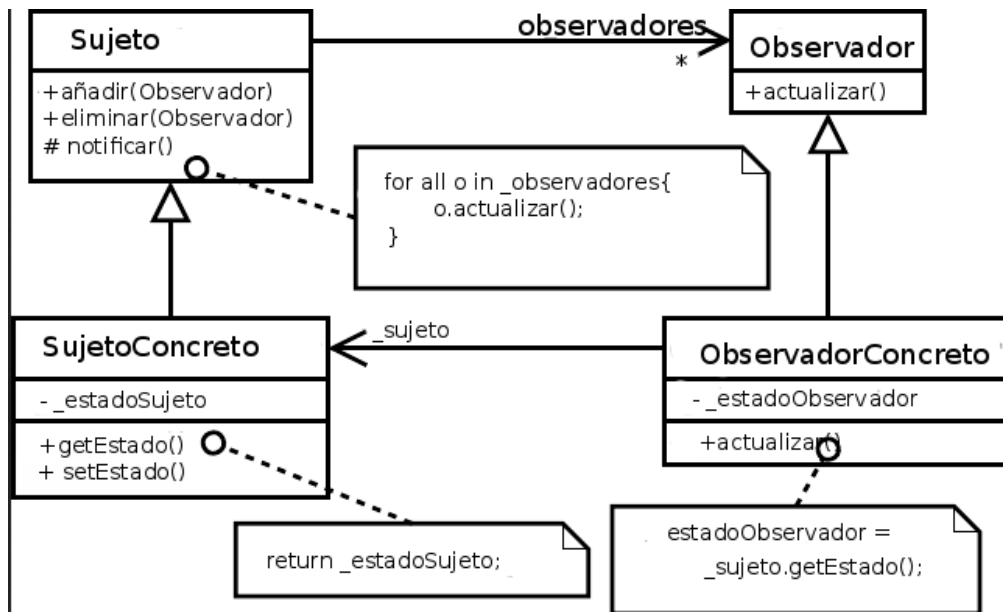
- **Observer:** define una dependencia del tipo *uno-a-muchos* entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes.

Tendremos sujetos concretos cuyos cambios pueden resultar interesantes a otros y observadores a los que al menos les interesa estar pendientes de un elemento y en un momento dado, reaccionar ante sus notificaciones de cambio. Todos los sujetos tienen en común que un conjunto de objetos quieren estar pendientes de ellos. Cualquier elemento que quiera ser observado tiene que permitir indicar:

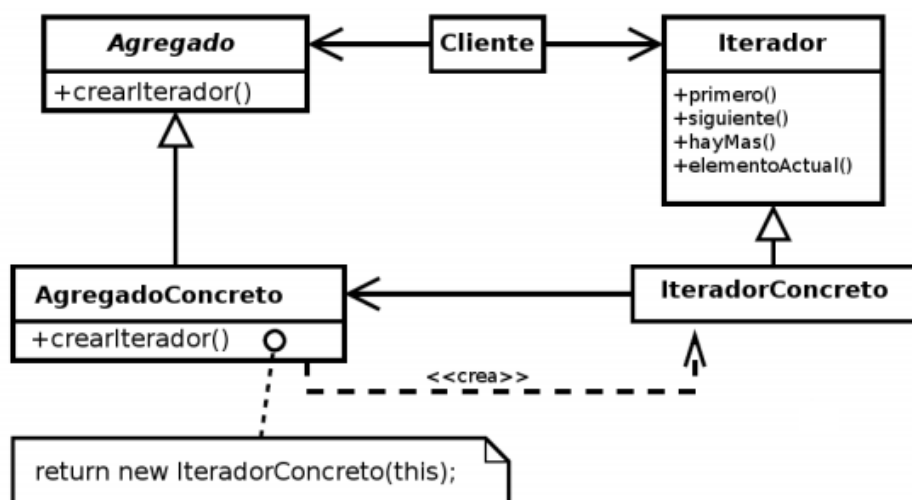
1. “Estoy interesado en tus cambios”.
2. “Ya no estoy interesado en tus cambios”.

El observable tiene que tener, además, un mecanismo de aviso a los interesados. A continuación tenemos a los participantes de forma desglosada:

- **Sujeto (Subject):** el sujeto concreto proporciona una interfaz para agregar (attach) y eliminar (detach) observadores. El Sujeto conoce a todos sus observadores.
- **Observador (Observer):** define el método que usa el sujeto para notificar cambios en su estado (update/notify).
- **Sujeto Concreto (ConcreteSubject):** mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado. No tienen porque ser elementos de la misma jerarquía.
- **Observador Concreto (ConcreteObserver):** mantiene una referencia al sujeto concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio.



- **Iterator**: define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección. Las entidades participantes en el diseño propuesto por el patrón iterador son:
  - **Iterador (Iterator)** define la interfaz para recorrer el agregado de elementos y acceder a ellos, de manera que el cliente no tenga que conocer los detalles y sea capaz de manejarlos de todos modos.
  - **Iterador Concreto (ConcreteIterator)** implementa la interfaz propuesta por el Iterador. Es el que se encarga de mantener la posición actual en el recorrido de la estructura.
  - **Agregado (Aggregate)** define la interfaz para el método de fabricación de iteradores.
  - **Agregado Concreto (ConcreteAggregate)** implementa la estructura de datos y el método de fabricación de iteradores que crea un iterador específico para su estructura.

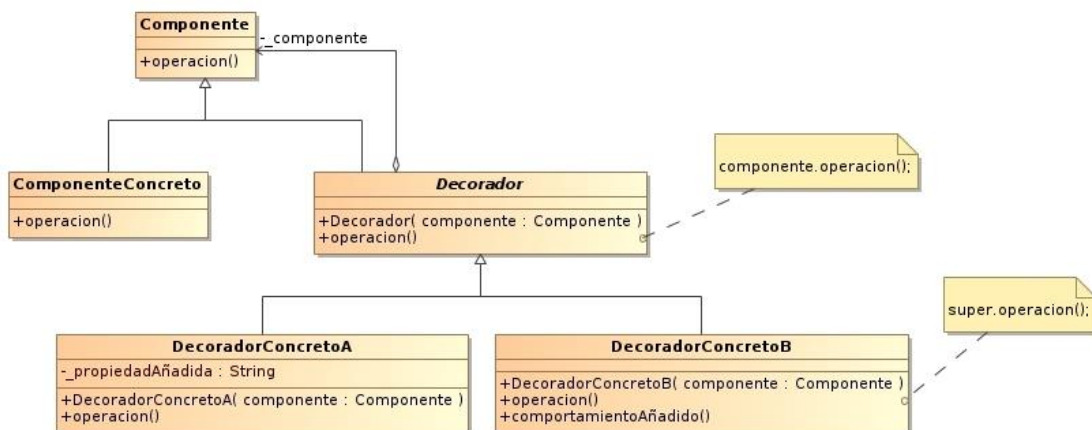


- **Decorator**: responde a la necesidad de añadir dinámicamente funcionalidad a un Objeto. Esto nos permite no tener que crear sucesivas clases que hereden de la

primera incorporando la nueva funcionalidad, sino otras que la implementan y se asocian a la primera.

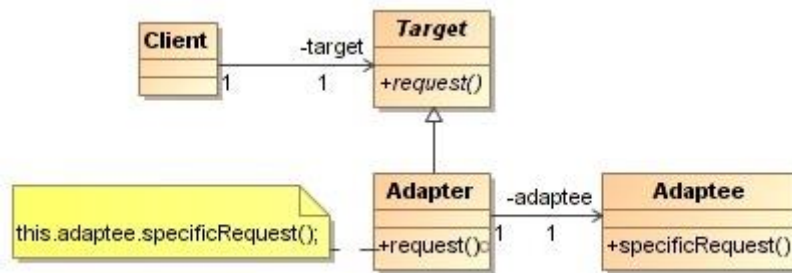
- **Componente:** define la interfaz para los objetos que pueden tener responsabilidades añadidas.
- **Componente Concreto:** define un objeto al cual se le pueden agregar responsabilidades adicionales.
- **Decorator:** mantiene una referencia al componente asociado. Implementa la interfaz de la superclase Componente delegando en el componente asociado.
- **Decorator Concreto:** añade responsabilidades al componente.

```
ComponenteConcreto c = new ComponenteConcreto();
    DecoradorConcretoA d1 = new
DecoradorConcretoA(c);
    DecoradorConcretoB d2 = new
DecoradorConcretoB(d1);
    d2.operacion();
```

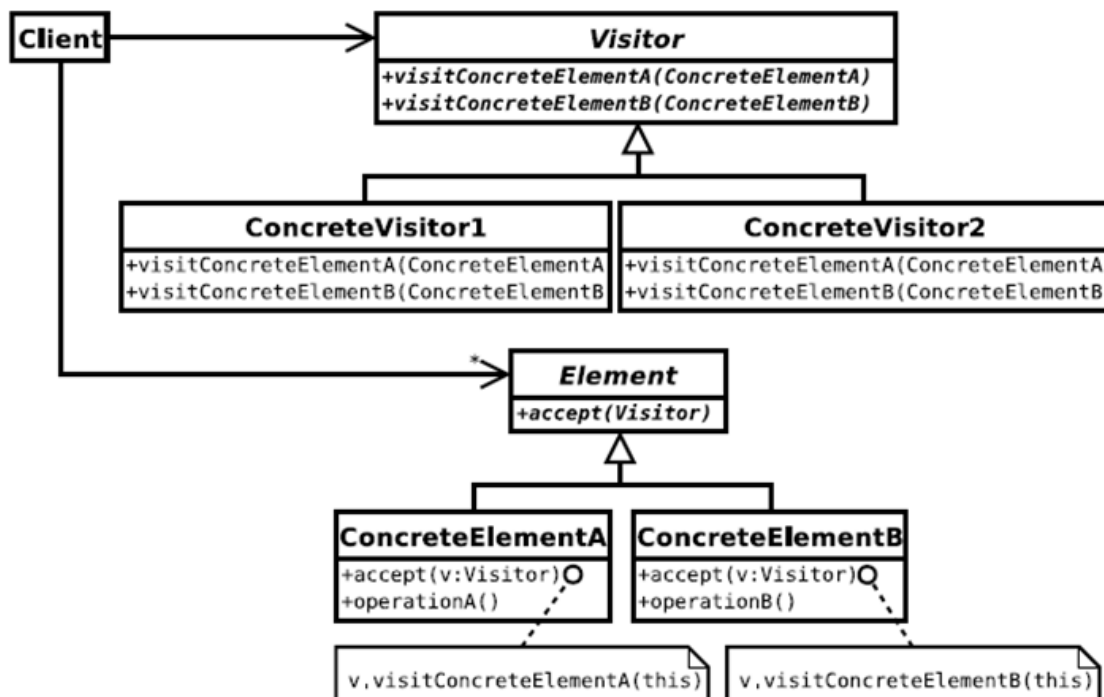


- **Adapter:** se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.
  - **Target:** define la interfaz específica del dominio que *Client* usa.
  - **Client:** colabora con la conformación de objetos para la interfaz *Target*.
  - **Adaptee:** define una interfaz existente que necesita adaptarse
  - **Adapter:** adapta la interfaz de *Adaptee* a la interfaz *Target*





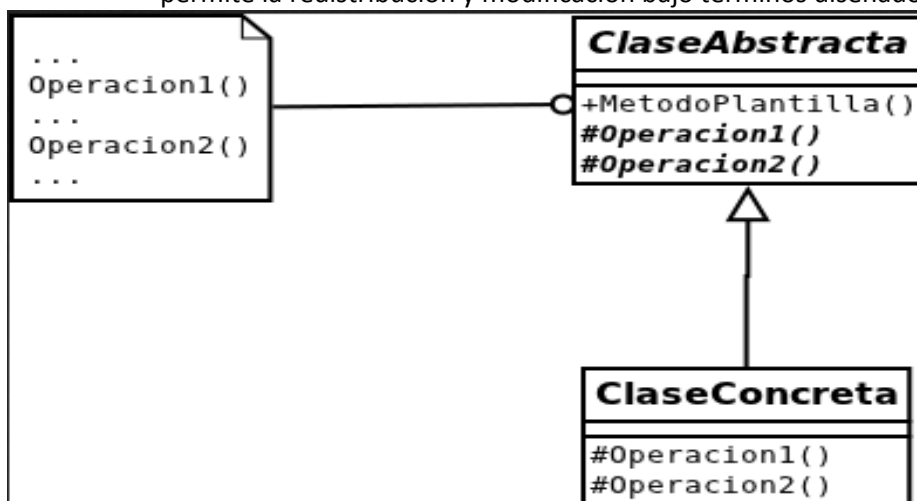
- **Visitor:** es una forma de separar el algoritmo de la estructura de un objeto. La idea básica es que se tiene un conjunto de clases elemento que conforman la estructura de un objeto. Cada una de estas clases elemento tiene un método aceptar (*accept()*) que recibe al objeto visitante (*visitor*) como argumento. El visitante es una interfaz que tiene un método *visit* diferente para cada clase elemento; por tanto habrá implementaciones de la interfaz visitor de la forma: *visitorClase1*, *visitorClase2*... *visitorClaseN*. El método *accept* de una clase elemento llama al método *visit* de su clase. Clases concretas de un visitante pueden entonces ser escritas para hacer una operación en particular.
  - **Visitante (Visitor):** declara una operación de visita para cada elemento concreto en la estructura de objetos, que incluye el propio objeto visitado
  - **Visitante Concreto (ConcreteVisitor1/2):** implementa las operaciones del visitante y acumula resultados como estado local
  - **Elemento (Element):** define una operación "Accept" que toma un visitante como argumento
  - **Elemento Concreto (ConcreteElementA/B):** implementa la operación "Accept"



- **Template:** se caracteriza por la definición, dentro de una operación de una superclase, de los pasos de un algoritmo, de forma que todos o parte de estos pasos son redefinidos en las subclases herederas de la citada superclase. Permite que ciertos pasos de un algoritmo definido en una operación de una superclase, sean redefinidos en las subclases sin necesidad de tener que sobrescribir la operación entera.
  - **Clase Abstracta:** proporciona la definición de una serie de operaciones primitivas (normalmente abstractas) que implementan los pasos de un algoritmo y que serán definidas en las subclases. Se encarga también de la implementación de un método desde el cual son invocadas, entre otras, las operaciones primitivas. Dicho método actúa a modo de plantilla, de ahí el nombre de este patrón, definiendo la secuencia de operaciones de un algoritmo.
  - **Clase Concreta:** implementa las operaciones primitivas definidas en la clase abstracta de la cual hereda, quedando así determinado el comportamiento específico del algoritmo definido en el método plantilla, para cada subclase.

### Software libre

- **Definición:** es la denominación del software que se refiere a la libertad de los usuarios para ejecutar, copiar, distribuir, y estudiar el mismo, e incluso modificar el software y distribuirlo modificado. El software libre suele estar disponible gratuitamente, o al precio de costo de la distribución a través de otros medios; sin embargo no es obligatorio que sea así, por lo tanto no hay que asociar **software libre** a "*software gratuito*" (denominado usualmente freeware), ya que, conservando su carácter de libre, puede ser distribuido comercialmente ("software comercial"). Análogamente, el "software gratis" o "gratuito" incluye en ocasiones el código fuente; no obstante, este tipo de software *no es libre* en el mismo sentido que el software libre, a menos que se garanticen los derechos de modificación y redistribución de dichas versiones modificadas del programa.
- **Tipos de licencias:**
  1. **GPL:** una de las más utilizadas es la Licencia Pública General de GNU (GNU GPL). El autor conserva los derechos de autor (copyright), y permite la redistribución y modificación bajo términos diseñados para



asegurarse de que todas las versiones modificadas del software permanecen bajo los términos más restrictivos de la propia GNU GPL. Esto hace que sea imposible crear un producto con partes no licenciadas GPL: el conjunto tiene que ser GPL. Es decir, la licencia GNU GPL posibilita la modificación y redistribución del software, pero únicamente bajo esa misma licencia. Y añade que si se reutiliza en un mismo programa código "A" licenciado bajo licencia GNU GPL y código "B" licenciado bajo otro tipo de licencia libre, el código final "C", independientemente de la cantidad y calidad de cada uno de los códigos "A" y "B", debe estar bajo la licencia GNU GPL. En la práctica esto hace que las licencias de software libre se dividan en dos grandes grupos, aquellas que pueden ser mezcladas con código licenciado bajo GNU GPL (y que inevitablemente desaparecerán en el proceso, al ser el código resultante licenciado bajo GNU GPL) y las que no lo permiten al incluir mayores u otros requisitos que no contemplan ni admiten la GNU GPL y que por lo tanto no pueden ser enlazadas ni mezcladas con código gobernado por la licencia GNU GPL. Aproximadamente el 60% del software licenciado como software libre emplea una licencia GPL o de manejo

2. **AGPL:** la Licencia Pública General de Affero (en inglés Affero General Public License, también Affero GPL o AGPL) es una licencia copyleft derivada de la Licencia Pública General de GNU diseñada específicamente para asegurar la cooperación con la comunidad en el caso de software que corra en servidores de red. La Affero GPL es íntegramente una GNU GPL con una cláusula nueva que añade la obligación de distribuir el software si éste se ejecuta para ofrecer servicios a través de una red de ordenadores.
3. **BSD:** llamadas así porque se utilizan en gran cantidad de software distribuido junto a los sistemas operativos BSD. El autor, bajo tales licencias, mantiene la protección de copyright únicamente para la renuncia de garantía y para requerir la adecuada atribución de la autoría en trabajos derivados, pero permite la libre redistribución y modificación, incluso si dichos trabajos tienen propietario. Son muy permisivas, tanto que son fácilmente absorbidas al ser mezcladas con la licencia GNU GPL con quienes son compatibles. Puede argumentarse que esta licencia asegura "verdadero" software libre, en el sentido que el usuario tiene libertad ilimitada con respecto al software, y que puede decidir incluso redistribuirlo como no libre. Otras opiniones están orientadas a destacar que este tipo de licencia no contribuye al desarrollo de más software libre (normalmente utilizando la siguiente analogía: "una licencia BSD es más libre que una GPL si y sólo si se opina también que un país que permita la esclavitud es más libre que otro que no la permite").
4. **MPL:** esta licencia es de Software Libre y tiene un gran valor porque fue el instrumento que empleó Netscape Communications Corp. para liberar su Netscape Communicator 4.0 y empezar ese proyecto tan

importante para el mundo del Software Libre: Mozilla. Se utilizan en gran cantidad de productos de software libre de uso cotidiano en todo tipo de sistemas operativos. La MPL es Software Libre y promueve eficazmente la colaboración evitando el efecto "viral" de la GPL (si usas código licenciado GPL, tu desarrollo final tiene que estar licenciado GPL). Desde un punto de vista del desarrollador la GPL presenta un inconveniente en este punto, y lamentablemente mucha gente se cierra en banda ante el uso de dicho código. No obstante la MPL no es tan excesivamente permisiva como las licencias tipo BSD. Estas licencias son denominadas de copyleft débil. La NPL (luego la MPL) fue la primera licencia nueva después de muchos años, que se encargaba de algunos puntos que no fueron tenidos en cuenta por las licencias BSD y GNU. En el espectro de las licencias de software libre se la puede considerar adyacente a la licencia estilo BSD, pero perfeccionada.

5. **CopyLeft:** hay que hacer constar que el titular de los derechos de autor (copyright) de un software bajo licencia copyleft puede también realizar una versión modificada bajo su copyright original, y venderla bajo cualquier licencia que desee, además de distribuir la versión original como software libre. Esta técnica ha sido usada como un modelo de negocio por una serie de empresas que realizan software libre (por ejemplo MySQL); esta práctica no restringe ninguno de los derechos otorgados a los usuarios de la versión copyleft. En España, toda obra derivada está tan protegida como una original, siempre que la obra derivada parta de una autorización contractual con el autor. En el caso genérico de que el autor retire las licencias "copyleft", no afectaría de ningún modo a los productos derivados anteriores a esa retirada, ya que no tiene efecto retroactivo. En términos legales, el autor no tiene derecho a retirar el permiso de una licencia en vigencia. Si así sucediera, el conflicto entre las partes se resolvería en un pleito convencional.

- **Modelo de negocio:** se caracteriza principalmente por la oferta de servicios adicionales al software como:

**1. Doble Licenciamiento:** que se basa en distribuir un software bajo una licencia libre (habitualmente GPL) y otra propietaria, por tanto su negocio se basa principalmente en ofrecer servicios adicionales o soporte dedicado al producto con licencia privativa. El ejemplo más claro es MySQL.

**2. Open Core:** aquí se dispone del núcleo del software con licencia libre, y una serie de plugins o funcionalidades adicionales con licencia propietaria (y que son habitualmente las que hacen que el software libre tenga una amplia funcionalidad o resulte de utilidad). Este modelo tiene gran controversia a su alrededor, ya que se beneficia de las aportaciones realizadas al núcleo por parte de la comunidad, sin embargo no distribuyen de forma libre los *plugins*

adicionales que suponen grandes mejoras en el software. Un ejemplo es SugarCRM.

**3. Especialistas del Producto:** se basa en el amplio conocimiento en una herramienta libre (habitualmente desarrollada por esa compañía), por tanto pueden obtener beneficios de la formación y la consultoría entorno a dicho producto. Ejemplo: Alfresco.

**4. Platform providers:** su negocio se basa en proveer de soporte, integración, testing y servicios, de herramientas libres. Como por ejemplo: RedHat.

**5. Compañías de selección y consultoría:** selección de personal y consultoría de proyectos, no desarrollan software libre, sino que están orientadas a buscar personal adecuado para proyectos de software libre.

**6. Proveedores de soporte:** proporcionan soporte de primer nivel a diferentes productos libres. Por ejemplo: OpenLogic.

**7. Consultoría Legal:** el mundo de las licencias de software, así como las patentes, es un mundo que requiere una especialización para poder tratarlo, ya que es complejo y extenso. Por tanto surge un modelo de negocio que proporciona asesoría legal y consultoría para el estudio de licencias, infracciones, etc. Por ejemplo: Palamida.

**8. Documentación y formación.**

**9. Compartición de costes de I+D:** con el objetivo de reducir los costes de I+D se basan en unificar los costes de investigación con el desarrollo con la comunidad para poder obtener un buen producto, que es distribuido bajo licencia libre. Por ejemplo: Maemo.

**10. Beneficios indirectos:** el objetivo no es el software en sí sino valores externos que le proporcionarán una vía de ingresos a la compañía a través de ese software.

**11. Donaciones y patrocinios**

- **Ejemplos de software libre:** MySQL, Ares, eMule, Audacity, XAMPP, Notepad++, Linux, Apache, FileZilla, etc.

- Charlas TED
- Enfrentamiento a entrevistas con clientes
- Servidores
- MySQL y MongoDB
- Test units

- **Ramas (branches)**
- **Sistema de tickets (issues)**