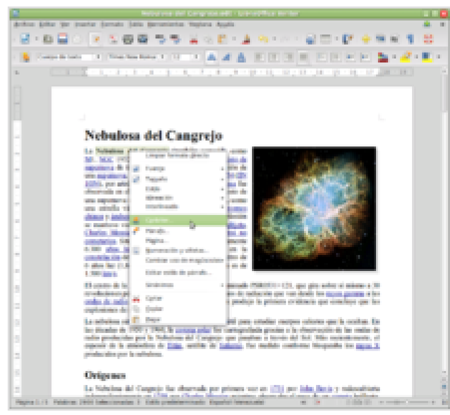
Software



Dentro de la categoría de *software* de aplicación están incluidos los procesadores de texto como LibreOffice Writer.

Se conoce como ***software*** (pronunciación en inglés: /ˈsɔftˌwɛr/),[1] **logicial** o **soporte lógico** al sistema formal de un sistema informático, que comprende el conjunto de los componentes lógicos necesarios que hace posible la realización de tareas específicas, en contraposición a los componentes físicos que son llamados *hardware*. La interacción entre el software y el hardware hace operativo un ordenador (u otro dispositivo), es decir, el *software* envía instrucciones que el *hardware* ejecuta, haciendo posible su funcionamiento.

Los componentes lógicos incluyen, entre muchos otros, las aplicaciones informáticas, tales como el procesador de texto, que permite al usuario realizar todas las tareas concernientes a la edición de textos; el llamado *software* de sistema, tal como el sistema operativo, que básicamente permite al resto de los programas funcionar adecuadamente, facilitando también la interacción entre los componentes físicos y el resto de las aplicaciones, y proporcionando una interfaz con el usuario.[2]

El *software*, en su gran mayoría, está escrito en lenguajes de programación de alto nivel, ya que son más fáciles y eficientes para que los programadores los usen, porque son más cercanos al lenguaje natural respecto del lenguaje de máquina.[3] Los lenguajes de alto nivel se traducen a lenguaje de máquina utilizando un compilador o un intérprete, o bien una combinación de ambos. El *software* también puede estar escrito en lenguaje ensamblador, que es de bajo nivel y tiene una alta correspondencia con las instrucciones de lenguaje máquina; se traduce al lenguaje de la máquina utilizando un ensamblador.

El anglicismo *software* es el más ampliamente difundido al referirse a este concepto, especialmente en la jerga técnica, en tanto que el término sinónimo «logicial», derivado del término francés *logiciel*, es utilizado mayormente en países y zonas de influencia francesa.

Etimología

Software (AFI: [ˈsoft.wer]) es una palabra proveniente del inglés, que en español no posee una traducción adecuada al contexto, por lo cual se la utiliza asiduamente sin traducir y así fue admitida por la Real Academia Española (RAE).[4] Aunque puede no ser estrictamente lo mismo, suele sustituirse por expresiones tales como *programas (informáticos)*, *aplicaciones (informáticas)* o *soportes lógicos*.[5]

Software es lo que se denomina *producto* en ingeniería de *software*.[6]

El término «logicial» es un calco léxico del término francés *logiciel*, neologismo que se formó en 1969 a partir de las palabras *logique* ('lógica') y *matériel* ('material') como traducción de la Delegación de la informática responsable del Plan Calcul.[7]

Definición de *software*

Existen varias definiciones similares aceptadas para *software*, pero probablemente la más formal sea la siguiente:

> Es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados, que forman parte de las operaciones de un sistema de computación.
>
> Extraído del estándar 729 del IEEE[8]

Considerando esta definición, el concepto de *software* va más allá de los programas de computación en sus distintos estados: código fuente, binario o ejecutable; también su documentación, los datos a procesar e incluso la información de usuario forman parte del *software*: es decir, *abarca todo lo intangible*, todo lo «no físico» relacionado.

El término *software* fue usado por primera vez en este sentido por John W. Tukey en 1957. En la ingeniería de *software* y las ciencias de la computación, el *software* es toda la información procesada por los sistemas informáticos: programas y datos.

El concepto de leer diferentes secuencias de instrucciones (programa) desde la memoria de un dispositivo para controlar los cálculos fue introducido por Charles Babbage como parte de su máquina diferencial. La teoría que forma la base de la mayor parte del *software* moderno fue propuesta por Alan Turing en su ensayo de 1936, «Los números computables», con una aplicación al problema de decisión.[9]

Clasificación del *software*

Si bien esta distinción es, en cierto modo arbitraria, y a veces confusa, a los fines prácticos se puede clasificar al *software* en tres tipos:[10]

Buscador de Programas en Ubuntu 13.10

- ***Software* de sistema:** Su objetivo es vincular adecuadamente al usuario y al programador de los detalles del sistema informático en particular que se use, aislándolo especialmente del procesamiento referido a las características internas de: memoria, discos, puertos y dispositivos de comunicaciones, impresoras, pantallas, teclados, etc. El *software* de sistema le procura al usuario y programador adecuadas interfaces de alto nivel, controladores, herramientas y utilidades de apoyo que permiten el mantenimiento del sistema global. Incluye entre otros:
  - Sistemas operativos
  - Controladores de dispositivos
  - Herramientas de diagnóstico
  - Herramientas de corrección y optimización
  - Servidores
  - Utilidades

- ***Software* de programación:** Es el conjunto de herramientas que permiten al programador desarrollar programas de informática, usando diferentes alternativas y lenguajes de programación, de una manera práctica. Incluyen en forma básica:
  - Editores de texto
  - Compiladores
  - Intérpretes
  - Enlazadores
  - Depuradores
  - Entornos de desarrollo integrados (IDE): Agrupan las anteriores herramientas, usualmente en un entorno visual, de forma tal que el programador no necesite introducir múltiples comandos para compilar, interpretar, depurar, etc. Habitualmente cuentan con una avanzada interfaz gráfica de usuario (GUI).
- ***Software* de aplicación**: Es aquel que permite a los usuarios llevar a cabo una o varias tareas específicas, en cualquier campo de actividad susceptible de ser automatizado o asistido, con especial énfasis en los negocios. Incluye entre muchos otros:
  - Aplicaciones para Control de sistemas y automatización industrial
  - Aplicaciones ofimáticas
  - *Software* educativo
  - *Software* empresarial[11]
  - Bases de datos
  - Telecomunicaciones (por ejemplo Internet y toda su estructura lógica)
  - Videojuegos
  - *Software* médico
  - *Software* de cálculo numérico y simbólico.
  - *Software* de diseño asistido (CAD)
  - *Software* de control numérico (CAM)

Proceso de creación del *software*

Se define como «proceso» al conjunto ordenado de pasos a seguir para llegar a la solución de un problema u obtención de un producto, en este caso particular, para lograr un producto *software* que resuelva un problema específico.

El proceso de creación de *software* puede llegar a ser muy complejo, dependiendo de su porte, características y criticidad del mismo. Por ejemplo la creación de un sistema operativo es una tarea que requiere proyecto, gestión, numerosos recursos y todo un equipo disciplinado de trabajo. En el otro extremo, si se trata de un sencillo programa (por ejemplo, la resolución de una ecuación de segundo orden), este puede ser realizado por un solo programador (incluso aficionado) fácilmente. Es así que normalmente se dividen en tres categorías según su tamaño (líneas de código) o costo: de «pequeño», «mediano» y «gran porte». Existen varias metodologías para estimarlo, una de las más populares es el sistema COCOMO que provee métodos y un *software* (programa) que calcula y provee una aproximación de todos los costos de producción en un «proyecto *software*» (relación horas/hombre, costo monetario, cantidad de líneas fuente de acuerdo a lenguaje usado, etc.).

Considerando los de gran porte, es necesario realizar complejas tareas, tanto técnicas como de gerencia, una fuerte gestión y análisis diversos (entre otras cosas), la complejidad de ello ha llevado a que desarrolle una ingeniería específica para tratar su estudio y realización: es conocida como ingeniería de *Software*.

En tanto que en los de mediano porte, pequeños equipos de trabajo (incluso un avezado analista-programador solitario) pueden realizar la tarea. Aunque, siempre en casos de mediano y gran porte (y a veces también en algunos de pequeño porte, según su complejidad), se deben seguir ciertas etapas que son necesarias para la construcción del *software*. Tales etapas, si bien deben existir, son flexibles en su forma de aplicación, de acuerdo a la metodología o proceso de desarrollo escogido y utilizado por el equipo de desarrollo o por el analista-programador solitario (si fuere el caso).

Los «procesos de desarrollo de *software*» poseen reglas preestablecidas, y deben ser aplicados en la creación del *software* de mediano y gran porte, ya que en caso contrario lo más seguro es que el proyecto no logre concluir o termine sin cumplir los objetivos previstos, y con variedad de fallos inaceptables (fracasan, en pocas palabras). Entre tales «procesos» los hay ágiles o livianos (ejemplo XP), pesados y lentos (ejemplo RUP), y variantes intermedias. Normalmente se aplican de acuerdo al tipo y porte del *software* a desarrollar, a criterio del líder (si lo hay) del equipo de desarrollo. Algunos de esos procesos son Programación Extrema (en inglés *eXtreme Programming* o XP), Proceso Unificado de Rational (en inglés Rational Unified Process o RUP), Feature Driven Development (FDD), etc.[12]

Cualquiera sea el «proceso» utilizado y aplicado al desarrollo del *software* (RUP, FDD, XP, etc), y casi independientemente de él, siempre se debe aplicar un «modelo de ciclo de vida».[13]

Se estima que, del total de proyectos *software* grandes emprendidos, un 28 % fracasan, un 46 % caen en severas modificaciones que lo retrasan y un 26 % son totalmente exitosos.[14]

Cuando un proyecto fracasa, rara vez es debido a fallas técnicas, la principal causa de fallos y fracasos es la falta de aplicación de una buena metodología o proceso de desarrollo. Entre otras, una fuerte tendencia, desde hace pocas décadas, es mejorar las metodologías o procesos de desarrollo, o crear nuevas y concientizar a los profesionales de la informática a su utilización adecuada. Normalmente los especialistas en el estudio y desarrollo de estas áreas (metodologías) y afines (tales como modelos y hasta la gestión misma de los proyectos) son los ingenieros en *software*, es su orientación. Los especialistas en cualquier otra área de desarrollo informático (analista, programador, Lic. en informática, ingeniero en informática, ingeniero de sistemas, etc.) normalmente aplican sus conocimientos especializados pero utilizando modelos, paradigmas y procesos ya elaborados.

Es común para el desarrollo de *software* de mediano porte que los equipos humanos involucrados apliquen «metodologías propias», normalmente un híbrido de los procesos anteriores y a veces con criterios propios.[15]

El proceso de desarrollo puede involucrar numerosas y variadas tareas,[13] desde lo administrativo, pasando por lo técnico y hasta la gestión y el gerenciamiento. Pero, casi rigurosamente, siempre se cumplen ciertas **etapas mínimas**; las que se pueden resumir como sigue:

- Captura, elicitación,[16] especificación y análisis de requisitos (ERS)
- Diseño
- Codificación
- Pruebas (unitarias y de integración)
- Instalación y paso a producción
- Mantenimiento

En las anteriores etapas pueden variar ligeramente sus nombres, o ser más globales, o contrariamente, ser más refinadas; por ejemplo indicar como una única fase (a los fines documentales e interpretativos) de «análisis y diseño»; o indicar como «implementación» lo que está dicho como «codificación»; pero en rigor, todas existen e incluyen, básicamente, las mismas tareas específicas.

En el apartado 4 del presente artículo se brindan mayores detalles de cada una de las etapas indicadas.

Modelos de proceso o ciclo de vida

Para cada una de las fases o etapas listadas en el ítem anterior, existen sub-etapas (o tareas). El modelo de proceso o modelo de ciclo de vida utilizado para el desarrollo, define el orden de las tareas o actividades involucradas,[13] también define la coordinación entre ellas, y su enlace y realimentación. Entre los más conocidos se puede mencionar: modelo en cascada o secuencial, modelo espiral, modelo iterativo incremental. De los antedichos hay a su vez algunas variantes o alternativas, más o menos atractivas según sea la aplicación requerida y sus requisitos.[14]

Modelo cascada

Este, aunque es más comúnmente conocido como modelo en cascada es también llamado «modelo clásico», «modelo tradicional» o «modelo lineal secuencial».

El modelo en cascada puro «difícilmente se utiliza tal cual», pues esto implicaría un previo y *absoluto* conocimiento de los requisitos, la no volatilidad de los mismos (o rigidez) y etapas subsiguientes libres de errores; ello solo podría ser aplicable a escasos y pequeños sistemas a desarrollar. En estas circunstancias, el paso de una etapa a otra de las mencionadas sería sin retorno, por ejemplo pasar del diseño a la codificación implicaría un diseño exacto y sin errores ni probable modificación o evolución: «codifique lo diseñado sin errores, no habrá en absoluto variantes futuras». Esto es utópico; ya que intrínsecamente «el *software* es de carácter evolutivo»,[17] cambiante y difícilmente libre de errores, tanto durante su desarrollo como durante su vida operativa.[13]

Algún cambio durante la ejecución de una cualquiera de las etapas en este modelo secuencial podría implicar reiniciar desde el principio todo el ciclo completo, lo cual redundaría en altos costos de tiempo y desarrollo. La Figura 2 muestra un posible esquema del modelo en cuestión.[13]

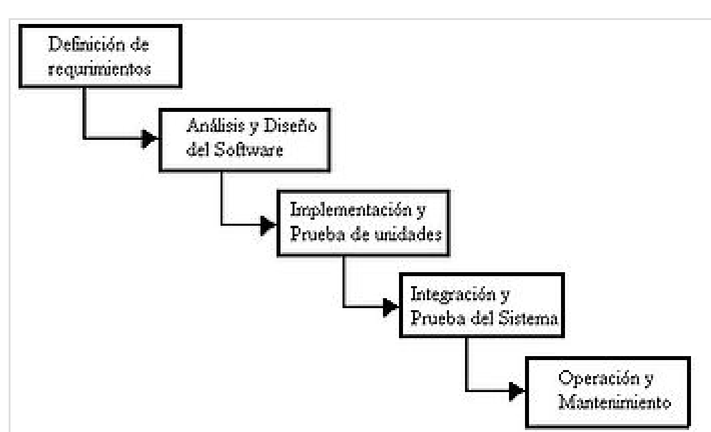


Figura 2: Modelo cascada puro o secuencial para el ciclo de vida del software.

Sin embargo, el modelo cascada en algunas de sus variantes es uno de los actualmente *más utilizados*,[18] por su eficacia y simplicidad, más que nada en *software* de pequeño y algunos de mediano porte; pero nunca (o muy rara vez) se lo usa en su "forma pura", como se dijo anteriormente. En lugar de ello, siempre se produce alguna realimentación entre etapas, que no es completamente predecible ni rígida; esto da oportunidad al desarrollo de productos *software* en los cuales hay ciertas incertezas, cambios o evoluciones durante el ciclo de vida. Así por ejemplo, una vez capturados y especificados los requisitos (primera etapa) se puede pasar al diseño del sistema, pero durante esta última

fase lo más probable es que se deban realizar ajustes en los requisitos (aunque sean mínimos), ya sea por fallas detectadas, ambigüedades o bien porque los propios requisitos han cambiado o evolucionado; con lo cual se debe retornar a la primera o previa etapa, hacer los reajustes pertinentes y luego continuar nuevamente con el diseño; esto último se conoce como realimentación. Lo normal en el modelo cascada es entonces la aplicación del mismo con sus etapas realimentadas de alguna forma, permitiendo retroceder de una a la anterior (e incluso poder saltar a varias anteriores) si es requerido.

De esta manera se obtiene el «modelo cascada realimentado», que puede ser esquematizado como lo ilustra la Figura 3.

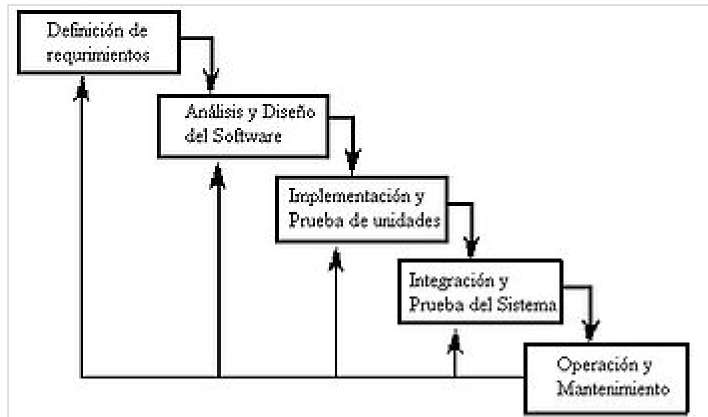


Figura 3: Modelo cascada realimentado para el ciclo de vida.

Lo dicho es, a grandes rasgos, la forma y utilización de este modelo, uno de los más usados y populares.[13] El modelo cascada realimentado resulta muy atractivo, hasta ideal, si el proyecto presenta alta rigidez (pocos cambios, previsto no evolutivo), los requisitos son muy claros y están correctamente especificados.[18]

Hay más variantes similares al modelo: refino de etapas (más etapas, menores y más específicas) o incluso mostrar menos etapas de las indicadas, aunque en tal caso la faltante estará dentro de alguna otra. El orden de esas fases indicadas en el ítem previo es el lógico y adecuado, pero adviértase, como se dijo, que normalmente habrá realimentación hacia atrás.

El modelo lineal o en cascada es el paradigma más antiguo y extensamente utilizado, sin embargo las críticas a él (ver desventajas) han puesto en duda su eficacia. Pese a todo, tiene un lugar muy importante en la ingeniería de *software* y continúa siendo el más utilizado; y siempre es mejor que un enfoque al azar.[18]

Desventajas del modelo cascada:[13]

- Los cambios introducidos durante el desarrollo pueden confundir al equipo profesional en las etapas tempranas del proyecto. Si los cambios se producen en etapa madura (codificación o prueba) pueden ser catastróficos para un proyecto grande.
- No es frecuente que el cliente o usuario final explicite clara y completamente los requisitos (etapa de inicio); y el modelo lineal así lo requiere. La incertidumbre natural en los comienzos es luego difícil de acomodar.[18]
- El cliente debe tener paciencia ya que el *software* no estará disponible hasta muy avanzado el proyecto. Un error importante detectado por el cliente (en fase de operación) puede ser desastroso, implicando reinicio del proyecto, con altos costos.

Modelos evolutivos

El *software* evoluciona con el tiempo.[19] [17] Los requisitos del usuario y del producto suelen cambiar conforme se desarrolla el mismo. Las fechas de mercado y la competencia hacen que no sea posible esperar a poner en el mercado un producto absolutamente completo, por lo que se aconseja introducir una versión funcional limitada de alguna forma para aliviar las presiones competitivas.

En esas u otras situaciones similares, los desarrolladores necesitan modelos de progreso que estén diseñados para acomodarse a una evolución temporal o progresiva, donde los requisitos centrales son conocidos de antemano, aunque no estén bien definidos a nivel detalle.

En el modelo cascada y cascada realimentado no se tiene demasiado en cuenta la naturaleza evolutiva del *software*,[19] se plantea como estático, con requisitos bien conocidos y definidos desde el inicio.[13]

Los evolutivos son modelos iterativos, permiten desarrollar versiones cada vez más completas y complejas, hasta llegar al objetivo final deseado; incluso evolucionar más allá, durante la fase de operación.

Los modelos «iterativo incremental» y «espiral» (entre otros) son dos de los más conocidos y utilizados del tipo evolutivo.[18]

Modelo iterativo incremental

En términos generales, se puede distinguir, en la figura 4, los pasos generales que sigue el proceso de desarrollo de un producto *software*. En el modelo de ciclo de vida seleccionado, se identifican claramente dichos pasos. La descripción del sistema es esencial para especificar y confeccionar los distintos incrementos hasta llegar al producto global y final. Las actividades concurrentes (especificación, desarrollo y validación) sintetizan el desarrollo pormenorizado de los incrementos, que se hará posteriormente.

El diagrama de la figura 4 muestra en forma muy esquemática, el funcionamiento de un ciclo iterativo incremental, el cual permite la entrega de versiones parciales a medida que se va construyendo el producto final.[13] Es decir, a medida que cada incremento definido llega a su etapa de operación y mantenimiento. Cada versión emitida incorpora a los anteriores incrementos las funcionalidades y requisitos que fueron analizados como necesarios.

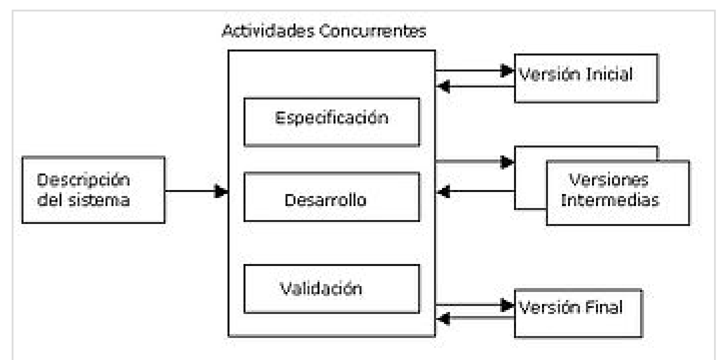


Figura 4: Diagrama genérico del desarrollo evolutivo incremental.

El incremental es un modelo de tipo evolutivo que está basado en varios ciclos cascada realimentados aplicados repetidamente, con una filosofía iterativa.[18] En la figura 5 se muestra un refino del diagrama previo, bajo un esquema temporal, para obtener finalmente el esquema del modelo de ciclo de vida iterativo incremental, con sus actividades genéricas asociadas. Aquí se observa claramente cada ciclo cascada que es aplicado para la obtención de un incremento; estos últimos se van integrando para obtener el producto final completo. Cada incremento es un ciclo cascada realimentado, aunque, por simplicidad, en la figura 5 se muestra como secuencial puro.

Se observa que existen actividades de desarrollo (para cada incremento) que son realizadas en paralelo o concurrentemente, así por ejemplo, en la Figura, mientras se realiza el diseño detalle del primer incremento ya se está realizando en análisis del segundo. La Figura 5 es solo esquemática, un incremento no necesariamente se iniciará durante la fase de diseño del anterior, puede ser posterior (incluso antes), en cualquier tiempo de la etapa previa. Cada incremento concluye con la actividad de «operación y mantenimiento» (indicada como «Operación» en la figura), que es donde se produce la entrega del producto parcial al cliente. El momento de inicio de cada incremento es dependiente de varios factores: tipo de

sistema; independencia o dependencia entre incrementos (dos de ellos totalmente independientes pueden ser fácilmente iniciados al mismo tiempo si se dispone de personal suficiente); capacidad y cantidad de profesionales involucrados en el desarrollo; etc.[15]

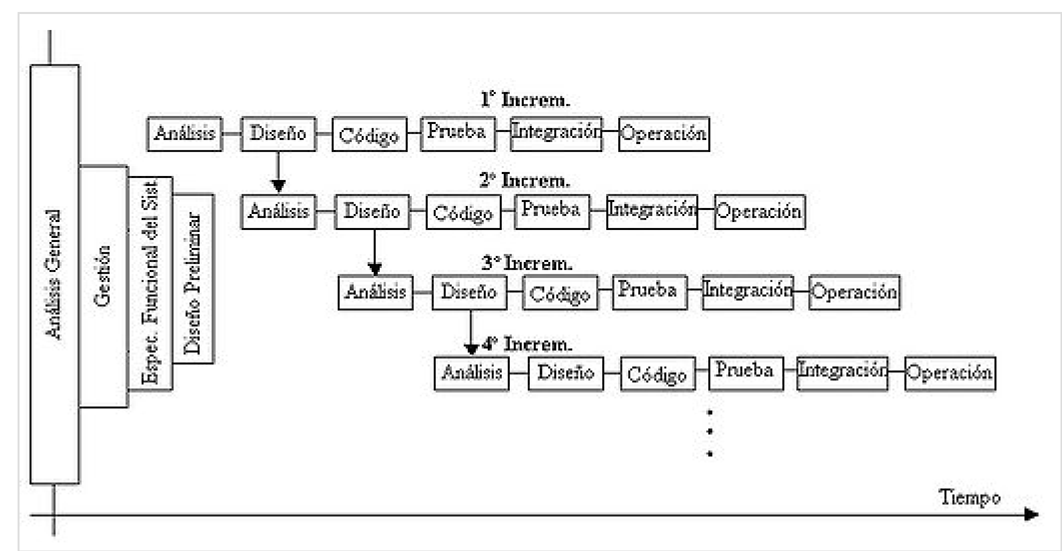


Figura 5: Modelo iterativo incremental para el ciclo de vida del *software*.

Bajo este modelo se entrega *software* «por partes funcionales más pequeñas», pero reutilizables, llamadas incrementos. En general cada incremento se construye sobre aquel que ya fue entregado.[13]

Como se muestra en la Figura 5, se aplican secuencias Cascada en forma escalonada, mientras progresa el tiempo calendario. Cada secuencia lineal o Cascada produce un incremento y a menudo el primer incremento es un sistema básico, con muchas funciones suplementarias (conocidas o no) sin entregar.

El cliente utiliza inicialmente ese sistema básico, intertanto, el resultado de su uso y evaluación puede aportar al plan para el desarrollo del/los siguientes incrementos (o versiones). Además también aportan a ese plan otros factores, como lo es la priorización (mayor o menor urgencia en la necesidad de cada incremento en particular) y la dependencia entre incrementos (o independencia).

Luego de cada integración se entrega un producto con mayor funcionalidad que el previo. El proceso se repite hasta alcanzar el *software* final completo.

Siendo iterativo, *con el modelo incremental se entrega un producto parcial pero completamente operacional en cada incremento*, y no una parte que sea usada para reajustar los requisitos (como si ocurre en el modelo de construcción de prototipos).[18]

El enfoque incremental resulta muy útil cuando se dispone de baja dotación de personal para el desarrollo; también si no hay disponible fecha límite del proyecto por lo que se entregan versiones incompletas pero que proporcionan al usuario funcionalidad básica (y cada vez mayor). También es un modelo útil a los fines de versiones de evaluación.

Nota: Puede ser considerado y útil, en cualquier momento o incremento incorporar temporalmente el paradigma MCP como complemento, teniendo así una mixtura de modelos que mejoran el esquema y desarrollo general.

Ejemplo:

> Un procesador de texto que sea desarrollado bajo el paradigma incremental podría aportar, en principio, funciones básicas de edición de archivos y producción de

documentos (algo como un editor simple). En un segundo incremento se le podría agregar edición más sofisticada, y de generación y mezcla de documentos. En un tercer incremento podría considerarse el agregado de funciones de corrección ortográfica, esquemas de paginado y plantillas; en un cuarto capacidades de dibujo propias y ecuaciones matemáticas. Así sucesivamente hasta llegar al procesador final requerido. Así, el producto va creciendo, acercándose a su meta final, pero desde la entrega del primer incremento ya es útil y funcional para el cliente, el cual observa una respuesta rápida en cuanto a entrega temprana; sin notar que la fecha límite del proyecto puede no estar acotada ni tan definida, lo que da margen de operación y alivia presiones al equipo de desarrollo.[15]

Como se dijo, el iterativo incremental es un modelo del tipo evolutivo, es decir donde se permiten y esperan probables cambios en los requisitos en tiempo de desarrollo; se admite cierto margen para que el *software* pueda evolucionar.[17] Aplicable cuando los requisitos son medianamente bien conocidos pero no son completamente estáticos y definidos, cuestión esa que si es indispensable para poder utilizar un modelo Cascada.

El modelo es aconsejable para el desarrollo de *software* en el cual se observe, en su etapa inicial de análisis, que posee áreas bastante bien definidas a cubrir, con suficiente independencia como para ser desarrolladas en etapas sucesivas. Tales áreas a cubrir suelen tener distintos grados de apremio por lo cual las mismas se deben priorizar en un análisis previo, es decir, definir cual será la primera, la segunda, y así sucesivamente; esto se conoce como «definición de los incrementos» con base en la priorización. Pueden no existir prioridades funcionales por parte del cliente, pero el desarrollador debe fijarlas de todos modos y con algún criterio, ya que basándose en ellas se desarrollarán y entregarán los distintos incrementos.

El hecho de que existan incrementos funcionales del *software* lleva inmediatamente a pensar en un esquema de desarrollo modular, por tanto este modelo facilita tal paradigma de diseño.

En resumen, un modelo incremental lleva a pensar en un desarrollo modular, con entregas parciales del producto *software* denominados «incrementos» del sistema, que son escogidos según prioridades predefinidas de algún modo. El modelo permite una implementación con refinamientos sucesivos (ampliación o mejora). Con cada incremento se agrega nueva funcionalidad o se cubren nuevos requisitos o bien se mejora la versión previamente implementada del producto *software*.

Este modelo brinda cierta flexibilidad para que durante el desarrollo se incluyan cambios en los requisitos por parte del usuario, un cambio de requisitos propuesto y aprobado puede analizarse e implementarse como un nuevo incremento o, eventualmente, podrá constituir una mejora/adecuación de uno ya planeado. Aunque si se produce un cambio de requisitos por parte del cliente que afecte incrementos previos ya terminados (detección/incorporación tardía) *se debe evaluar la factibilidad y realizar un acuerdo con el cliente, ya que puede impactar fuertemente en los costos.*[15]

La selección de este modelo permite realizar **entregas funcionales tempranas al cliente** (lo cual es beneficioso tanto para él como para el grupo de desarrollo). Se priorizan las entregas de aquellos módulos o incrementos en que surja la necesidad operativa de hacerlo, por ejemplo para cargas previas de información, indispensable para los incrementos siguientes.[18]

El modelo iterativo incremental no obliga a especificar con precisión y detalle absolutamente todo lo que el sistema debe hacer, (y cómo), antes de ser construido (como el caso del cascada, con requisitos congelados). Solo se hace en el incremento en desarrollo. Esto torna más manejable el proceso y reduce el impacto en los

costos. Esto es así, porque en caso de alterar o rehacer los requisitos, solo afecta una parte del sistema. Aunque, lógicamente, esta situación se agrava si se presenta en estado avanzado, es decir en los últimos incrementos. *En definitiva, el modelo facilita la incorporación de nuevos requisitos durante el desarrollo.*

Con un paradigma incremental se reduce el tiempo de desarrollo inicial, ya que se implementa funcionalidad parcial. También provee un impacto ventajoso frente al cliente, que es la entrega temprana de partes operativas del *software*.

El modelo proporciona todas las ventajas del modelo en cascada realimentado, reduciendo sus desventajas solo al ámbito de cada incremento.

El modelo incremental no es recomendable para casos de sistemas de tiempo real, de alto nivel de seguridad, de procesamiento distribuido, o de alto índice de riesgos.

Modelo espiral

El modelo espiral fue propuesto inicialmente por Barry Boehm. Es un modelo evolutivo que conjuga la naturaleza iterativa del modelo MCP con los aspectos controlados y sistemáticos del Modelo Cascada. Proporciona potencial para desarrollo rápido de versiones incrementales. En el modelo espiral el *software* se construye en una serie de versiones incrementales. En las primeras iteraciones la versión incremental podría ser un modelo en papel o bien un prototipo. En las últimas iteraciones se producen versiones cada vez más completas del sistema diseñado.[13] [18]

El modelo se divide en un número de Actividades de marco de trabajo, llamadas «regiones de tareas». En general existen entre tres y seis regiones de tareas (hay variantes del modelo). En la figura 6 se muestra el esquema de un modelo espiral con seis regiones. En este caso se explica una variante del modelo original de Boehm, expuesto en su tratado de 1988; en 1998 expuso un tratado más reciente.

Las regiones definidas en el modelo de la figura son:

- Región 1 — Tareas requeridas para establecer la comunicación entre el cliente y el desarrollador.
- Región 2 — Tareas inherentes a la definición de los recursos, tiempo y otra información relacionada con el proyecto.
- Región 3 — Tareas necesarias para evaluar los riesgos técnicos y de gestión del proyecto.
- Región 4 — Tareas para construir una o más *representaciones* de la aplicación *software*.
- Región 5 — Tareas para construir la aplicación, instalarla, probarla y proporcionar soporte al usuario o cliente (Ej. documentación y práctica).
- Región 6 — Tareas para obtener la reacción del cliente, según la evaluación de lo creado e instalado en los ciclos anteriores.

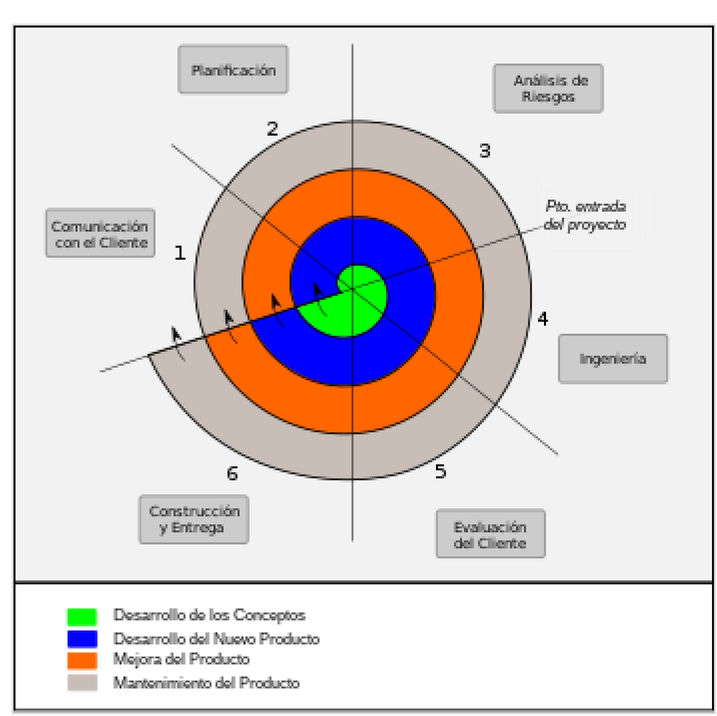


Figura 6: Modelo espiral para el ciclo de vida del *software*.

Las actividades enunciadas para el marco de trabajo son generales y se aplican a cualquier proyecto, grande, mediano o pequeño, complejo o no. Las regiones que definen esas actividades comprenden un «conjunto de tareas» del trabajo: ese conjunto sí se debe adaptar a las características del proyecto en particular a emprender. Nótese que lo listado en los ítems de 1 a 6 son conjuntos de tareas, algunas de las ellas normalmente dependen del proyecto o desarrollo en sí.

Proyectos pequeños requieren baja cantidad de tareas y también de formalidad. En proyectos mayores o críticos cada región de tareas contiene labores de más alto nivel de formalidad. En cualquier caso se aplican actividades de protección (por ejemplo, gestión de configuración del *software*, garantía de calidad, etc.).

Al inicio del ciclo, o proceso evolutivo, el equipo de ingeniería gira alrededor del espiral (metafóricamente hablando) comenzando por el centro (marcado con ๑ en la figura 6) y en el sentido indicado; el primer circuito de la espiral puede producir el desarrollo de una especificación del producto; los pasos siguientes podrían generar un prototipo y progresivamente versiones más sofisticadas del *software*.

Cada paso por la región de planificación provoca ajustes en el plan del proyecto; el coste y planificación se realimentan en función de la evaluación del cliente. El gestor de proyectos debe ajustar el número de iteraciones requeridas para completar el desarrollo.

El modelo espiral puede ir adaptándose y aplicarse a lo largo de todo el Ciclo de vida del *software* (en el modelo clásico, o cascada, el proceso termina a la entrega del *software*).

Una visión alternativa del modelo puede observarse examinando el «eje de punto de entrada de proyectos». Cada uno de los circulitos (⊙) fijados a lo largo del eje representan puntos de arranque de los distintos proyectos (relacionados); a saber:

- Un proyecto de «desarrollo de conceptos» comienza al inicio de la espiral, hace múltiples iteraciones hasta que se completa, es la zona marcada con verde.
- Si lo anterior se va a desarrollar como producto real, se inicia otro proyecto: «Desarrollo de nuevo Producto». Que evolucionará con iteraciones hasta culminar; es la zona marcada en color azul.
- Eventual y análogamente se generarán proyectos de «mejoras de productos» y de «mantenimiento de productos», con las iteraciones necesarias en cada área (zonas roja y gris, respectivamente).

Cuando la espiral se caracteriza de esta forma, está operativa hasta que el *software* se retira, eventualmente puede estar inactiva (el proceso), pero cuando se produce un cambio el proceso arranca nuevamente en el punto de entrada apropiado (por ejemplo, en «mejora del producto»).

El modelo espiral da un enfoque realista, que evoluciona igual que el *software*;[19] se adapta muy bien para desarrollos a gran escala.

El Espiral utiliza el MCP para reducir riesgos y permite aplicarlo en cualquier etapa de la evolución. Mantiene el enfoque clásico (cascada) pero incorpora un marco de trabajo iterativo que refleja mejor la realidad.

Este modelo *requiere considerar riesgos técnicos* en todas las etapas del proyecto; aplicado adecuadamente debe reducirlos antes de que sean un verdadero problema.

El Modelo evolutivo como el Espiral es particularmente apto para el desarrollo de Sistemas Operativos (complejos); también en sistemas de altos riesgos o críticos (Ej. navegadores y controladores aeronáuticos) y en todos aquellos en que sea necesaria una fuerte gestión del proyecto y sus riesgos, técnicos o de gestión.

Desventajas importantes:

- Requiere mucha experiencia y habilidad para la evaluación de los riesgos, lo cual es requisito para el éxito del proyecto.
- Es difícil convencer a los grandes clientes que se podrá controlar este enfoque evolutivo.

Este modelo no se ha usado tanto, como el Cascada (Incremental) o MCP, por lo que no se tiene bien medida su eficacia, es un paradigma relativamente nuevo y difícil de implementar y controlar.

Modelo espiral Win & Win

Una variante interesante del Modelo Espiral previamente visto (Figura 6) es el «Modelo espiral Win-Win»[14] (Barry Boehm). El Modelo Espiral previo (clásico) sugiere la comunicación con el cliente para fijar los requisitos, en que simplemente se pregunta al cliente qué necesita y él proporciona la información para continuar; pero esto es en un contexto ideal que rara vez ocurre. Normalmente cliente y desarrollador entran en una negociación, se negocia coste frente a funcionalidad, rendimiento, calidad, etc.

«Es así que la obtención de requisitos requiere una negociación, que tiene éxito cuando ambas partes ganan».

Las mejores negociaciones se fuerzan en obtener «Victoria & Victoria» (Win & Win), es decir que el cliente gane obteniendo el producto que lo satisfaga, y el desarrollador también gane consiguiendo presupuesto y fecha de entrega realista. Evidentemente, este modelo requiere fuertes habilidades de negociación.

El modelo Win-Win define un conjunto de actividades de negociación al principio de cada paso alrededor de la espiral; se definen las siguientes actividades:

1. Identificación del sistema o subsistemas clave de los directivos * (saber qué quieren).
2. Determinación de «condiciones de victoria» de los directivos (saber qué necesitan y los satisface).
3. Negociación de las condiciones «victoria» de los directivos para obtener condiciones «Victoria & Victoria» (negociar para que ambos ganen).

* Directivo: Cliente escogido con interés directo en el producto, que puede ser premiado por la organización si tiene éxito o criticado si no.

El modelo Win & Win hace énfasis en la negociación inicial, también introduce 3 hitos en el proceso llamados «puntos de fijación», que ayudan a establecer la completitud de un ciclo de la espiral, y proporcionan hitos de decisión antes de continuar el proyecto de desarrollo del *software*.

Etapas en el desarrollo del *software*

Captura, análisis y especificación de requisitos

Al inicio de un desarrollo (no de un proyecto), esta es la primera fase que se realiza, y, según el modelo de proceso adoptado, puede casi terminar para pasar a la próxima etapa (caso de Modelo Cascada Realimentado) o puede hacerse parcialmente para luego retomarla (caso Modelo Iterativo Incremental u otros de carácter evolutivo).

En simple palabras y básicamente, durante esta fase, se adquieren, reúnen y especifican las características funcionales y no funcionales que deberá cumplir el futuro programa o sistema a desarrollar.

Las bondades de las características, tanto del sistema o programa a desarrollar, como de su entorno, parámetros no funcionales y arquitectura dependen enormemente de lo bien lograda que esté esta etapa. Esta es, probablemente, la de mayor importancia y una de las fases más difíciles de lograr certeramente, pues no es automatizable, no es muy técnica y depende en gran medida de la habilidad y experiencia del analista que la realice.

Involucra fuertemente al usuario o cliente del sistema, por tanto tiene matices muy subjetivos y es difícil de modelar con certeza o aplicar una técnica que sea «la más cercana a la adecuada» (de hecho no existe «la estrictamente adecuada»). Si bien se han ideado varias metodologías, incluso *software* de apoyo, para captura, elicitación y registro de requisitos, no existe una forma infalible o absolutamente confiable, y deben aplicarse conjuntamente buenos criterios y mucho sentido común por parte del o los analistas encargados de la tarea; es fundamental también lograr una fluida y adecuada comunicación y comprensión con el usuario final o cliente del sistema.

El artefacto más importante resultado de la culminación de esta etapa es lo que se conoce como especificación de requisitos *software* o simplemente documento ERS.

Como se dijo, la habilidad del analista para interactuar con el cliente es fundamental; lo común es que el cliente tenga un objetivo general o problema que resolver, no conoce en absoluto el área (informática), ni su jerga, ni siquiera sabe con precisión qué debería hacer el producto *software* (qué y cuantas funciones) ni, mucho menos, cómo debe operar. En otros casos menos frecuentes, el cliente «piensa» que sabe precisamente lo que el *software* tiene que hacer, y generalmente acierta muy parcialmente, pero su empecinamiento entorpece la tarea de elicitación. El analista debe tener la capacidad para lidiar con este tipo de problemas, que incluyen relaciones humanas; tiene que saber ponerse al nivel del usuario para permitir una adecuada comunicación y comprensión.[20]

Escasas son las situaciones en que el cliente sabe con certeza e incluso con completitud lo que requiere de su futuro sistema, este es el caso más sencillo para el analista.

Las tareas relativas a captura, elicitación, modelado y registro de requisitos, además de ser sumamente importante, puede llegar a ser dificultosa de lograr acertadamente y llevar bastante tiempo relativo al proceso total del desarrollo; al proceso y metodologías para llevar a cabo este conjunto de actividades normalmente se las asume parte propia de la ingeniería de *software*, pero dada la antedicha complejidad, actualmente se habla de una ingeniería de requisitos,[21] aunque ella aún no existe formalmente.

Hay grupos de estudio e investigación, en todo el mundo, que están exclusivamente abocados a idear modelos, técnicas y procesos para intentar lograr la correcta captura, análisis y registro de requisitos. Estos grupos son los que normalmente hablan de la ingeniería de requisitos; es decir se plantea esta como un área o disciplina pero no como una carrera universitaria en sí misma.

Algunos requisitos no necesitan la presencia del cliente, para ser capturados o analizados; en ciertos casos los puede proponer el mismo analista o, incluso, adoptar unilateralmente decisiones que considera adecuadas (tanto en requisitos funcionales como no funcionales). Por citar ejemplos probables: Algunos requisitos sobre la arquitectura del sistema, requisitos no funcionales tales como los relativos al rendimiento, nivel de soporte a errores operativos, plataformas de desarrollo, relaciones internas o ligas entre la información (entre registros o tablas de datos) a almacenar en caso de bases o bancos de datos, etc. Algunos funcionales tales como opciones secundarias o de soporte necesarias para una mejor o más sencilla operatividad; etc.

La obtención de especificaciones a partir del cliente (u otros actores intervinientes) es un proceso humano muy interactivo e iterativo; normalmente a medida que se captura la información, se la analiza y realimenta con el cliente, refinándola, puliéndola y corrigiendo si es necesario; cualquiera sea el método de ERS utilizado. El analista siempre debe llegar a conocer la temática y el problema que resolver, dominarlo, hasta cierto punto, hasta el ámbito que el futuro sistema a desarrollar lo abarque. Por ello el analista debe tener alta capacidad para comprender problemas de muy diversas áreas o disciplinas de trabajo (que no son específicamente suyas); así por ejemplo, si el sistema a desarrollar será para gestionar información de una aseguradora y sus sucursales remotas, el analista se debe compenetrar en cómo ella trabaja y maneja su información, desde niveles muy bajos e incluso llegando hasta los gerenciales. Dada a gran diversidad de campos a cubrir, los analistas suelen ser asistidos por especialistas, es decir gente que conoce profundamente el área para la cual se desarrollará el *software*; evidentemente una única persona (el analista) no puede abarcar tan vasta cantidad de áreas del conocimiento. En empresas grandes de desarrollo de productos *software*, es común tener analistas especializados en ciertas áreas de trabajo.

Contrariamente, no es problema del cliente, es decir él no tiene por qué saber nada de *software*, ni de diseños, ni otras cosas relacionadas; solo se debe limitar a aportar objetivos, datos e información (de mano propia o de sus registros, equipos, empleados, etc) al analista, y guiado por él, para que, en primera instancia, defina el «**Universo de Discurso**», y con posterior trabajo logre confeccionar el adecuado documento ERS.

Es bien conocida la presión que sufren los desarrolladores de sistemas informáticos para comprender y rescatar las necesidades de los clientes/usuarios. Cuanto más complejo es el contexto del problema más difícil es lograrlo, a veces se fuerza a los desarrolladores a tener que convertirse en casi expertos de los dominios que analizan.

Cuando esto no sucede es muy probable que se genere un conjunto de requisitos[22] erróneos o incompletos y por lo tanto un producto de *software* con alto grado de desaprobación por parte de los clientes/usuarios y un altísimo costo de reingeniería y mantenimiento. *Todo aquello que no se detecte, o resulte mal entendido en la etapa inicial provocará un fuerte impacto negativo en los requisitos, propagando esta corriente degradante a lo largo de todo el proceso de desarrollo e* ***incrementando su perjuicio cuanto más tardía sea su detección*** (Bell y Thayer 1976)(Davis 1993).

Procesos, modelado y formas de elicitación de requisitos

Siendo que la captura, elicitación y especificación de requisitos, es una parte crucial en el proceso de desarrollo de *software*, ya que de esta etapa depende el logro de los objetivos finales previstos, se han ideado modelos y diversas metodologías de trabajo para estos fines. También existen herramientas *software* que apoyan las tareas relativas realizadas por el ingeniero en requisitos.

El estándar IEEE 830-1998 brinda una normalización de las «Prácticas recomendadas para la especificación de requisitos *software*».[23]

A medida que se obtienen los requisitos, normalmente se los va analizando, el resultado de este análisis, con o sin el cliente, se plasma en un documento, conocido como ERS o Especificación de requisitos *software*, cuya estructura puede venir definida por varios estándares, tales como CMMI.

Un primer paso para realizar el relevamiento de información es el conocimiento y definición acertada lo que se conoce como «Universo de Discurso» del problema, que se define y entiende por:

Universo de Discurso (UdeD): es el contexto general en el cual el *software* deberá ser desarrollado y deberá operar. El UdeD incluye todas las fuentes de información y todas las personas relacionadas con el *software*. Esas personas son conocidas también como **actores** de ese universo. El UdeD es la realidad circunstanciada por el conjunto de objetivos definidos por quienes demandaron el *software*.

A partir de la extracción y análisis de información en su ámbito se obtienen todas las especificaciones necesarias y tipos de requisitos para el futuro producto *software*.

El objetivo de la ingeniería de requisitos (IR) es sistematizar el proceso de definición de requisitos permitiendo elicitar, modelar y analizar el problema, generando un compromiso entre los ingenieros de requisitos y los clientes/usuarios, ya que ambos participan en la generación y definición de los requisitos del sistema. La IR aporta un conjunto de métodos, técnicas y herramientas que asisten a los ingenieros de requisitos (analistas) para obtener requisitos lo más seguros, veraces, completos y oportunos posibles, permitiendo básicamente:

- Comprender el problema
- Facilitar la obtención de las necesidades del cliente/usuario
- Validar con el cliente/usuario
- Garantizar las especificaciones de requisitos

Si bien existen diversas formas, modelos y metodologías para elicitar, definir y documentar requisitos, no se puede decir que alguna de ellas sea mejor o peor que la otra, suelen tener muchísimo en común, y todas cumplen el mismo objetivo. Sin embargo, lo que si se puede decir sin dudas es que es indispensable utilizar alguna de ellas para documentar las especificaciones del futuro producto *software*. Así por ejemplo, hay un grupo de investigación argentino que desde hace varios años ha propuesto y estudia el uso del LEL (Léxico Extendido del Lenguaje) y Escenarios como metodología, aquí[24] se presenta una de las tantas referencias y bibliografía sobre ello. Otra forma, más ortodoxa, de capturar y documentar requisitos se puede obtener en detalle, por ejemplo, en el trabajo de la Universidad de Sevilla sobre «Metodología para el Análisis de Requisitos de Sistemas Software».[25]

En la Figura 7 se muestra un esquema, más o menos riguroso, aunque no detallado, de los pasos y tareas a seguir para realizar la captura, análisis y especificación de requisitos *software*. También allí se observa qué artefacto o documento se obtiene en cada etapa del proceso. En el diagrama no se explicita metodología o modelo a utilizar, sencillamente se pautan las tareas que deben cumplirse, de alguna manera.

Una posible lista, general y ordenada, de tareas recomendadas para obtener la definición de lo que se debe realizar, los productos a obtener y las técnicas a emplear durante la actividad de elicitación de requisitos, en fase de Especificación de requisitos *software* es:

1. Obtener información sobre el dominio del problema y el sistema actual (UdeD).
2. Preparar y realizar las reuniones para elicitación/negociación.

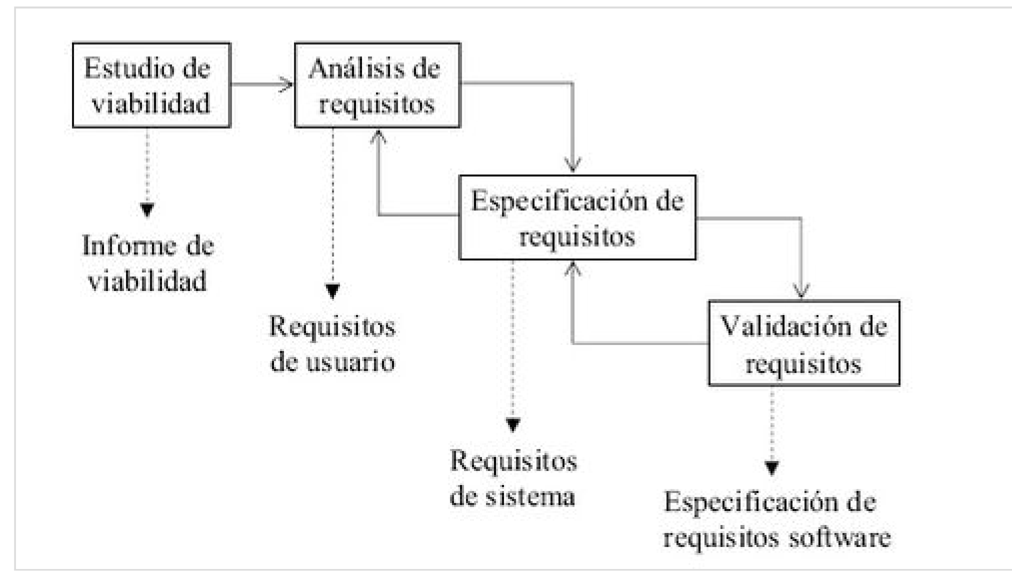


Figura 7: Diagrama de tareas para captura y análisis de requisitos.

3. Identificar/revisar los objetivos del usuario.
4. Identificar/revisar los objetivos del sistema.
5. Identificar/revisar los requisitos de información.
6. Identificar/revisar los requisitos funcionales.
7. Identificar/revisar los requisitos no funcionales.
8. Priorizar objetivos y requisitos.

Algunos principios básicos a tener en cuenta:

- Presentar y entender cabalmente el dominio de la información del problema.
- Definir correctamente las funciones que debe realizar el *software*.
- Representar el comportamiento del *software* a consecuencias de acontecimientos externos, particulares, incluso inesperados.
- Reconocer requisitos incompletos, ambiguos o contradictorios.
- Dividir claramente los modelos que representan la información, las funciones y comportamiento y características no funcionales.

Clasificación e identificación de requisitos

Se pueden identificar dos formas de requisitos:

- Requisitos de usuario: Los requisitos de usuario son frases en lenguaje natural junto a diagramas con los servicios que el sistema debe proporcionar, así como las restricciones bajo las que debe operar.
- Requisitos de sistema: Los requisitos de sistema determinan los servicios del sistema y pero con las restricciones en detalle. Sirven como contrato.

Es decir, ambos son lo mismo, pero con distinto nivel de detalle.

Ejemplo de requisito de usuario: El sistema debe hacer préstamos. Ejemplo de requisito de sistema: Función préstamo: entrada código socio, código ejemplar; salida: fecha devolución; etc.

Se clasifican en tres los tipos de requisitos de sistema:

- Requisitos funcionales

Los requisitos funcionales describen:

- Los servicios que proporciona el sistema (funciones).
- La respuesta del sistema ante determinadas entradas.
- El comportamiento del sistema en situaciones particulares.

- Requisitos no funcionales

Los requisitos no funcionales son restricciones de los servicios o funciones que ofrece el sistema (ej. cotas de tiempo, proceso de desarrollo, rendimiento, etc.)

> Ejemplo 1. La biblioteca Central debe ser capaz de atender simultáneamente a todas las bibliotecas de la Universidad
> Ejemplo 2. El tiempo de respuesta a una consulta remota no debe ser superior a 1/2 s

A su vez, hay tres tipos de requisitos no funcionales:

- Requisitos del producto. Especifican el comportamiento del producto (Ej. prestaciones, memoria, tasa de fallos, etc.)
- Requisitos organizativos. Se derivan de las políticas y procedimientos de las organizaciones de los clientes y desarrolladores (Ej. estándares de proceso, lenguajes de programación, etc.)
- Requisitos externos. Se derivan de factores externos al sistema y al proceso de desarrollo (Ej. requisitos legislativos, éticos, etc.)

- Requisitos del dominio.

Los requisitos del dominio se derivan del dominio de la aplicación y reflejan características de dicho dominio.

Pueden ser funcionales o no funcionales.

Ej. El sistema de biblioteca de la Universidad debe ser capaz de exportar datos mediante el Lenguaje de Intercomunicación de Bibliotecas de España (LIBE). Ej. El sistema de biblioteca no podrá acceder a bibliotecas con material censurado.

Diseño del sistema

En ingeniería de *software*, el diseño es una fase de ciclo de vida del *software*. Se basa en la especificación de requisitos producido por el análisis de los requisitos (fase de análisis), el diseño define *cómo* estos requisitos se cumplirán, la estructura que debe darse al sistema de *software* para que se haga realidad.

El diseño sigue siendo una fase separada de la programación o codificación, esta última corresponde a la traducción en un determinado lenguaje de programación de las premisas adoptadas en el diseño.

Las distinciones entre las actividades mencionadas hasta ahora no siempre son claras cómo se quisiera en las teorías clásicas de ingeniería de *software*. El diseño, en particular, puede describir el funcionamiento interno de un sistema en diferentes niveles de detalle, cada una de ellos se coloca en una posición intermedia entre el análisis y codificación.

Normalmente se entiende por "diseño de la arquitectura" al diseño de "muy alto nivel", que solo define la estructura del sistema en términos de la módulos de *software* de que se compone y las relaciones macroscópicas entre ellos. A este nivel de diseño pertenecen fórmulas como cliente-servidor o “tres niveles”, o, más generalmente, las decisiones sobre el uso de la arquitectura de hardware especial que se utilice, el sistema operativo, DBMS, Protocolos de red, etc.

Un nivel intermedio de detalle puede definir la descomposición del sistema en módulos, pero esta vez con una referencia más o menos explícita al modo de descomposición que ofrece el particular lenguaje de programación con el que el desarrollo se va a implementar, por ejemplo, en un diseño realizado con la tecnología de objetos, el proyecto podría describir al sistema en términos de clases y sus interrelaciones.

El diseño detallado, por último, es una descripción del sistema muy cercana a la codificación (por ejemplo, describir no solo las clases en abstracto, sino también sus atributos y los métodos con sus tipos).

Debido a la naturaleza "intangible" del *software*, y dependiendo de las herramientas que se utilizan en el proceso, la frontera entre el diseño y la codificación también puede ser virtualmente imposible de identificar. Por ejemplo, algunas herramientas CASE son capaces de generar código a partir de diagramas UML, los que describen gráficamente la estructura de un sistema *software*.

Codificación del *software*

Durante esta etapa se realizan las tareas que comúnmente se conocen como programación; que consiste, esencialmente, en llevar a código fuente, en el lenguaje de programación elegido, todo lo diseñado en la fase anterior. Esta tarea la realiza el programador, siguiendo por completo los lineamientos impuestos en el diseño y en consideración siempre a los requisitos funcionales y no funcionales (ERS) especificados en la primera etapa.

Es común pensar que la etapa de programación o codificación (algunos la llaman implementación) es la que insume la mayor parte del trabajo de desarrollo del *software*; sin embargo, esto puede ser relativo (y generalmente aplicable a sistemas de pequeño porte) ya que las etapas previas son cruciales, críticas y pueden llevar bastante más tiempo. Se suele hacer estimaciones de un 30 % del tiempo total insumido en la programación, pero esta cifra no es consistente ya que depende en gran medida de las características del sistema, su criticidad y el lenguaje de programación elegido.[14] En tanto menor es el nivel del lenguaje mayor será el tiempo de programación requerido, así por ejemplo se tardaría más tiempo en codificar un algoritmo en lenguaje ensamblador que el mismo programado en lenguaje C.

Mientras se programa la aplicación, sistema, o *software* en general, se realizan también tareas de depuración, esto es la labor de ir liberando al código de los errores factibles de ser hallados en esta fase (de semántica, sintáctica y lógica). Hay una suerte de solapamiento con la fase siguiente, ya que para depurar la lógica es necesario realizar pruebas unitarias, normalmente con datos de prueba; claro es que no todos los errores serán encontrados solo en la etapa de programación, habrá otros que se encontrarán durante las etapas subsiguientes. La aparición de algún error funcional (mala respuesta a los requisitos) tarde o temprano puede llevar a retornar a la fase de diseño antes de continuar la codificación.

Durante la fase de programación, el código puede adoptar varios estados, dependiendo de la forma de trabajo y del lenguaje elegido, a saber:

- Código fuente: es el escrito directamente por los programadores en editores de texto, lo cual genera el programa. Contiene el conjunto de instrucciones codificadas en algún lenguaje de

alto nivel. Puede estar distribuido en paquetes, procedimientos, bibliotecas fuente, etc.

- Código objeto: es el código binario o intermedio resultante de procesar con un compilador el código fuente. Consiste en una **traducción completa** y de una sola vez de este último. El código objeto no es inteligible por el ser humano (normalmente es formato binario) pero tampoco es directamente ejecutable por la computadora. Se trata de una representación intermedia entre el código fuente y el código ejecutable, a los fines de un enlace final con las rutinas de biblioteca y entre procedimientos o bien para su uso con un pequeño intérprete intermedio [a modo de distintos ejemplos véase EUPHORIA, (intérprete intermedio), FORTRAN (compilador puro) *MSIL (Microsoft Intermediate Language)* (intérprete) y BASIC (intérprete puro, intérprete intermedio, compilador intermedio o compilador puro, depende de la versión utilizada)].
  - El código objeto **no existe** si el programador trabaja con un lenguaje **a modo de intérprete puro**, en este caso el mismo intérprete se encarga de traducir y ejecutar línea por línea el código fuente (de acuerdo al flujo del programa), en tiempo de ejecución. En este caso **tampoco existe** el o los archivos de código ejecutable. Una desventaja de esta modalidad es que la ejecución del programa o sistema es un poco más lenta que si se hiciera con un intérprete intermedio, y bastante más lenta que si existe el o los archivos de código ejecutable. Es decir no favorece el rendimiento en velocidad de ejecución. Pero una gran ventaja de la modalidad intérprete puro, es que él está forma de trabajo facilita enormemente la tarea de depuración del código fuente (frente a la alternativa de hacerlo con un compilador puro). Frecuentemente se suele usar una forma mixta de trabajo (si el lenguaje de programación elegido lo permite), es decir inicialmente trabajar a modo de intérprete puro, y una vez depurado el código fuente (liberado de errores) se utiliza un compilador del mismo lenguaje para obtener el código ejecutable completo, con lo cual se agiliza la depuración y la velocidad de ejecución se optimiza.
- Código ejecutable: Es el código binario resultado de enlazar uno o más fragmentos de código objeto con las rutinas y bibliotecas necesarias. Constituye uno o más archivos binarios con un formato tal que el sistema operativo es capaz de cargarlo en la memoria RAM (eventualmente también parte en una memoria virtual), y proceder a su ejecución directa. Por lo anterior se dice que el código ejecutable es directamente «inteligible por la computadora». El código ejecutable, también conocido como código máquina, no existe si se programa con modalidad de «intérprete puro».

Pruebas (unitarias y de integración)

Entre las diversas pruebas que se le efectúan al *software* se pueden distinguir principalmente:

- Prueba unitarias: Consisten en probar o testear piezas de *software* pequeñas; a nivel de secciones, procedimientos, funciones y módulos; aquellas que tengan funcionalidades específicas. Dichas pruebas se utilizan para asegurar el correcto funcionamiento de secciones de código, mucho más reducidas que el conjunto, y que tienen funciones concretas con cierto grado de independencia.
- Pruebas de integración: Se realizan una vez que las pruebas unitarias fueron concluidas *exitosamente*; con éstas se intenta asegurar que el sistema completo, incluso los subsistemas que componen las piezas individuales grandes del *software* funcionen correctamente al operar e interoperar en conjunto.

Las pruebas normalmente se efectúan con los llamados datos de prueba, que es un conjunto seleccionado de datos típicos a los que puede verse sometido el sistema, los módulos o los bloques de código. También se escogen: Datos que llevan a condiciones límites al *software* a fin de probar su tolerancia y robustez; datos de utilidad para mediciones de rendimiento; datos que provocan condiciones eventuales o particulares poco

comunes y a las que el *software* normalmente no estará sometido pero pueden ocurrir; etc. Los «datos de prueba» no necesariamente son ficticios o «creados», pero normalmente sí lo son los de poca probabilidad de ocurrencia.

Generalmente, existe un fase probatoria final y completa del *software*, llamada Beta Test, durante la cual el sistema instalado en condiciones normales de operación y trabajo es probado exhaustivamente a fin de encontrar errores, inestabilidades, respuestas erróneas, etc. que hayan pasado los previos controles. Estas son normalmente realizadas por personal idóneo contratado o afectado específicamente a ello. Los posibles errores encontrados se transmiten a los desarrolladores para su depuración. En el caso de *software* de desarrollo «a pedido», el usuario final (cliente) es el que realiza el Beta Test, teniendo para ello un período de prueba pactado con el desarrollador.

Instalación y paso a producción

La instalación del *software* es el proceso por el cual los programas desarrollados son transferidos apropiadamente al computador destino, inicializados, y, finalmente, configurados; todo ello con el propósito de ser ya utilizados por el usuario final. Constituye la etapa final en el desarrollo propiamente dicho del *software*. Luego de esta el producto entrará en la fase de funcionamiento y producción, para el que fuera diseñado.

La instalación, dependiendo del sistema desarrollado, puede consistir en una simple copia al disco rígido destino (casos raros actualmente); o bien, más comúnmente, con una de complejidad intermedia en la que los distintos archivos componentes del *software* (ejecutables, bibliotecas, datos propios, etc.) son descomprimidos y copiados a lugares específicos preestablecidos del disco; incluso se crean vínculos con otros productos, además del propio sistema operativo. Este último caso, comúnmente es un proceso bastante automático que es creado y guiado con herramientas *software* específicas (empaquetado y distribución, instaladores).

En productos de mayor complejidad, la segunda alternativa es la utilizada, pero es realizada o guiada por especialistas; puede incluso requerirse la instalación en varios y distintos computadores (instalación distribuida).

También, en *software* de mediana y alta complejidad normalmente es requerido un proceso de configuración y chequeo, por el cual se asignan adecuados parámetros de funcionamiento y se testea la operatividad funcional del producto.

En productos de venta masiva las instalaciones completas, si son relativamente simples, suelen ser realizadas por los propios usuarios finales (tales como sistemas operativos, paquetes de oficina, utilitarios, etc.) con herramientas propias de instalación guiada; incluso la configuración suele ser automática. En productos de diseño específico o «a medida» la instalación queda restringida, normalmente, a personas especialistas involucradas en el desarrollo del *software* en cuestión.

Una vez realizada exitosamente la instalación del *software*, el mismo pasa a la fase de producción (operatividad), durante la cual cumple las funciones para las que fue desarrollado, es decir, es finalmente utilizado por el (o los) usuario final, produciendo los resultados esperados.

Mantenimiento

El mantenimiento de *software* es el proceso de control, mejora y optimización del *software* ya desarrollado e instalado, que también incluye depuración de errores y defectos que puedan haberse filtrado de la fase de pruebas de control y beta test. Esta fase es la última (antes de iterar, según el modelo empleado) que se aplica al ciclo de vida del desarrollo de *software*. La fase de mantenimiento es la que viene después de que el *software* está operativo y en producción.

De un buen diseño y documentación del desarrollo dependerá cómo será la fase de mantenimiento, tanto en costo temporal como monetario. Modificaciones realizadas a un *software* que fue elaborado con una documentación indebida o pobre y mal diseño puede llegar a ser tanto o más costosa que desarrollar el *software* desde el inicio. Por ello, es de fundamental importancia respetar debidamente todas las tareas de las fases del desarrollo y mantener adecuada y completa la documentación.

El período de la fase de mantenimiento es normalmente el mayor en todo el ciclo de vida.[14] Esta fase involucra también actualizaciones y evoluciones del *software*; no necesariamente implica que el sistema tuvo errores. Uno o más cambios en el *software*, por ejemplo de adaptación o evolutivos, puede llevar incluso a rever y adaptar desde parte de las primeras fases del desarrollo inicial, alterando todas las demás; dependiendo de cuán profundos sean los cambios. El modelo cascada común es particularmente costoso en mantenimiento, ya que su rigidez implica que cualquier cambio provoca regreso a fase inicial y fuertes alteraciones en las demás fases del ciclo de vida.

Durante el período de mantenimiento, es común que surjan nuevas revisiones y versiones del producto; que lo liberan más depurado, con mayor y mejor funcionalidad, mejor rendimiento, etc. Varias son las facetas que pueden ser alteradas para provocar cambios deseables, evolutivos, adaptaciones o ampliaciones y mejoras.

Básicamente se tienen los siguientes tipos de cambios:

- Perfectivos: Aquellos que llevan a una mejora de la calidad interna del *software* en cualquier aspecto: Reestructuración del código, definición más clara del sistema y su documentación; optimización del rendimiento y eficiencia.
- Evolutivos: Agregados, modificaciones, incluso eliminaciones, necesarias en el *software* para cubrir su expansión o cambio, según las necesidades del usuario.
- Adaptivos: Modificaciones que afectan a los entornos en los que el sistema opera, tales como: Cambios de configuración del hardware (por actualización o mejora de componentes electrónicos), cambios en el *software* de base, en gestores de base de datos, en comunicaciones, etc.
- Correctivos: Alteraciones necesarias para corregir errores de cualquier tipo en el producto de *software* desarrollado.

Carácter evolutivo del *software*

El *software* es el producto derivado del proceso de desarrollo, según la ingeniería de *software*. Este producto es intrínsecamente evolutivo durante su ciclo de vida: en general, evoluciona generando versiones cada vez más completas, complejas, mejoradas, optimizadas en algún aspecto, adecuadas a nuevas plataformas (sean de *hardware* o sistemas operativos), etc.[26]

Cuando un sistema deja de evolucionar, tarde o temprano cumplirá con su ciclo de vida, entrará en obsolescencia e inevitablemente, tarde o temprano, será reemplazado por un producto nuevo.

El *software* evoluciona sencillamente porque se debe adaptar a los cambios del entorno, sean funcionales (exigencias de usuarios), operativos, de plataforma o arquitectura *hardware*.

La dinámica de evolución del *software* es el estudio de los cambios del sistema. La mayor contribución en esta área fue realizada por Meir M. Lehman y Belady, comenzando en los años 70 y 80. Su trabajo continuó en la década de 1990, con Lehman y otros investigadores[27] de relevancia en la realimentación en los procesos de evolución (Lehman, 1996; Lehman et al., 1998; lehman et al., 2001). A partir de esos estudios propusieron un conjunto de leyes (conocidas como leyes de Lehman)[17] respecto de los cambios producidos en los sistemas. Estas leyes (en realidad son hipótesis) son invariantes y ampliamente aplicables.

Lehman y Belady analizaron el crecimiento y la evolución de varios sistemas *software* de gran porte; derivando finalmente, según sus medidas, las siguientes ocho leyes:

1. Cambio continuo: Un programa que se usa en un entorno real necesariamente debe cambiar o se volverá progresivamente menos útil en ese entorno.
2. Complejidad creciente: A medida que un programa en evolución cambia, su estructura tiende a ser cada vez más compleja. Se deben dedicar recursos extras para preservar y simplificar la estructura.
3. Evolución prolongada del programa: La evolución de los programas es un proceso autorregulativo. Los atributos de los sistemas, tales como tamaño, tiempo entre entregas y la cantidad de errores documentados son aproximadamente invariantes para cada entrega del sistema.
4. Estabilidad organizacional: Durante el tiempo de vida de un programa, su velocidad de desarrollo es aproximadamente constante e independiente de los recursos dedicados al desarrollo del sistema.
5. Conservación de la familiaridad: Durante el tiempo de vida de un sistema, el cambio incremental en cada entrega es aproximadamente constante.
6. Crecimiento continuado: La funcionalidad ofrecida por los sistemas tiene que crecer continuamente para mantener la satisfacción de los usuarios.
7. Decremento de la calidad: La calidad de los sistemas *software* comenzará a disminuir a menos que dichos sistemas se adapten a los cambios de su entorno de funcionamiento.
8. Realimentación del sistema: Los procesos de evolución incorporan sistemas de realimentación multiagente y multibucle y estos deben ser tratados como sistemas de realimentación para lograr una mejora significativa del producto.

Referencias

1. Diccionario de la lengua española 2005 (2010). wordreference.com, ed. «software» (http://www.wordreference.com/definicion/software) (diccionario). Espasa-Calpe. Consultado el 1 de febrero de 2010.
2. «SYSTEM SOFTWARE» (https://web.archive.org/web/20010530092843/http://home.olemiss.edu/~misbook/sfsysfm.htm). 30 de mayo de 2001. Archivado desde el original (http://home.olemiss.edu/~misbook/sfsysfm.htm) el 30 de mayo de 2001. Consultado el 10 de febrero de 2018.
3. «Onderwijs Informatica en Informatiekunde | Department of Information and Computing Sciences» (https://web.archive.org/web/20131102143144/http://www.cs.uu.nl/education/vak.php?vak=INFOMCCO). *www.cs.uu.nl*. Archivado desde el original (http://www.cs.uu.nl/education/vak.php?vak=INFOMCCO) el 2 de noviembre de 2013. Consultado el 10 de febrero de 2018.
4. Real Academia Española. «software» (https://dle.rae.es/software). *Diccionario de la lengua española* (23.ª edición). Consultado el 14 de marzo de 2008.

5. Real Academia Española y Asociación de Academias de la Lengua Española (2023). «software» (https://www.rae.es/dpd/software). *Diccionario panhispánico de dudas* (2.ª edición, versión provisional). Consultado el 8 de febrero de 2009.
6. Pressman, Roger S. (2003). «El producto». *Ingeniería del software, un enfoque práctico, Quinta edición edición*. México: Mc Graw Hill.
7. Pierre Mounier-Kuhn, *L'informatique en France, de la seconde guerre mondiale au Plan Calcul. L'émergence d'une science* (http://pups.paris-sorbonne.fr/catalogue/centre-roland-mousnier/linformatique-en-france-de-la-seconde-guerre-mondiale-au-plan-calcul/), Paris, PUPS, 2010, ch. 4.
8. IEEE Std, IEEE Software Engineering Standard: Glossary of Software Engineering Terminology. IEEE Computer Society Press, 1993
9. «Definición de Software» (https://www.ecured.cu/Software#:~:text=Probablemente%20la%20definici%C3%B3n%20m%C3%A1s%20formal%20de%20software%20sea%20la%20siguiente%3A&text=El%20t%C3%A9rmino%20%C2%ABsoftware%C2%BB%20fue%20usado,sistemas%20inform%C3%A1ticos%3A%20programas%20y%20datos).
10. «Componentes de la Clasificación del Software» (https://en.calameo.com/read/005519904c27a4a327cd1).
11. Euromundo Global. «El uso de un software reduce considerablemente los errores en la gestión de nóminas» (https://www.euromundoglobal.com/noticia/418780/ciencia-y-tecnologia/el-uso-de-un-software-reduce-considerablemente-los-errores-en-la-gestion-de-nominas.html). Consultado el 10 de diciembre de 2019.
12. Fernández del Busto De La Rosa, Gabriela. *Las TIC en la Educación*.
13. «Ciclo de Vida del Software» (https://web.archive.org/web/20130614204403/http://alarcos.inf-cr.uclm.es/doc/ISOFTWAREI/Tema03.pdf). Grupo Alarcos - Escuela Superior de Informática de Ciudad Real. Archivado desde el original (http://alarcos.inf-cr.uclm.es/doc/ISOFTWAREI/Tema03.pdf) el 14 de junio de 2013.
14. Pressman, Roger S. (2003). «El proceso». *Ingeniería del Software, un enfoque Práctico, Quinta edición edición*. México: Mc Graw Hill.
15. Alejandro Gutiérrez Díaz. «Ingeniería de Software Avanzada» (https://www.academia.edu/9540544/INGENIER%C3%8DA_DE_SOFTWARE_AVANZADA).
16. «Término "Elicitar" » (https://es.wiktionary.org/wiki/elicitar). 1ra. acepción - Wiktionary. Consultado el 15 de diciembre de 2008.
17. «Leyes de evolución del Software» (http://cnx.org/content/m17406/latest/). Connexions - Educational content repository.
18. «Ciclo de vida del Software y Modelos de desarrollo» (https://web.archive.org/web/20100215155237/http://www.cepeu.edu.py/LIBROS_ELECTRONICOS_3/lpcu097%20-%2001.pdf). Instituto de Formación Profesional - Libros Digitales. Archivado desde el original (http://www.cepeu.edu.py/LIBROS_ELECTRONICOS_3/lpcu097%20-%2001.pdf) el 15 de febrero de 2010. Texto «lugar: Asunción del Paraguay» ignorado (ayuda)
19. «Evolución del Software» (http://cnx.org/content/m17405/latest/). Connexions - Educational content repository.
20. Adolfo González Marín, Gustavo. *SOFTWARE PARA EL CONTROL Y MANEJO DEL FLUJO DE DATOS EN EL AREA DE PRODUCCION Y BODEGA DE LA EMPRESA SIMETRIA*. Pereira.
21. Software Requirements Engineering, 2nd Edition, IEEE Computer Society. Los Alamitos, CA, 1997 (Compendio de *papers* y artículos en ingeniería de requisitos)
22. «III Workshop de Engenharia de Requisitos» (http://www.informatik.uni-trier.de/~ley/db/conf/wer/wer2000.html). WER 2000, Río de Janeiro, 2000.
23. «Recommended Practice for Software Requirements Specification» (https://code.google.com/p/changecontrol/downloads/detail?name=IEEE%20830-1998%20Recommended%20Practice%20for%20Software%20Requirements%20Specifications.pdf&can=2&q=). *IEEE-SA Standards Board*.
24. «LEL y Escenarios como metodología en Ingeniería de Requisitos» (https://web.archive.org/web/20110728115910/http://ficcte.unimoron.edu.ar/wicc/Trabajos/III%20-%20isbd/673-Ridao_Doorn_wicc06.pdf). Univ. de Morón, Buenos Aires. Archivado desde el original (http://ficcte.unimoron.edu.ar/wicc/T

rabajos/III%20-%20isbd/673-Ridao_Doorn_wicc06.pdf) el 28 de julio de 2011. Consultado el 23 de julio de 2008.
25. «Metodología para el análisis de Requisitos de Sistemas Software» (http://www.infor.uva.es/~mlaguna/is1/materiales/metodologia_analisis.pdf). Univ. de Sevilla, 2001.
26. Sommerville, Ian (2005). «21-Evolución del software». *Ingeniería del Software*. España: Pearson Educación S.A.
27. «ACM Fellow Profile for Meir M. (Manny) Lehman» (https://web.archive.org/web/20110928015833/http://www.sigsoft.org/SEN/lehman.html). ACM. 31 de mayo de 2007. Archivado desde el original (http://www.sigsoft.org/SEN/lehman.html) el 28 de septiembre de 2011. Consultado el 27 de noviembre de 2011.

Bibliografía

Libros

- JACOBSON, Ivar; BOOCH, Grady; RUMBAUGH, James (2000). *El Proceso Unificado de Desarrollo de Software*. Pearson Addisson-Wesley.
- Pressman, Roger S. (2003). *Ingeniería del Software, un enfoque Práctico* (Quinta edición edición). Mc Graw Hill. ISBN 84-481-3214-9.
- JACOBSON; BOOCH; RUMBAUGH (1999). *UML - El Lenguaje Unificado de Modelado*. Pearson Addisson-Wesley. Rational Software Corporation, Addison Wesley Iberoamericana. ISBN 84-7829-028-1.
- Haeberer, A. M.; P. A. S. Veloso, G. Baum (1988). *Formalización del proceso de desarrollo de software* (Ed. preliminar edición). Buenos Aires: Kapelusz. ISBN 950-13-9880-3.
- Fowler, Martin; Kendall Sccott (1999). *UML Gota a Gota*. Addison Wesley. ISBN 9789684443648.
- Loucopoulos, Pericles; Karakostas, V. (1995). *System Requirements Engineering* (https://archive.org/details/systemrequiremen0000louc) (en inglés). London: McGraw-Hill Companies. pp. 160 (https://archive.org/details/systemrequiremen0000louc/page/160) p. ISBN 978-0077078430. Consultado el 2008.
- Sommerville, Ian; P. Sawyer (1997). *Requirements Engineering: A Good Practice Guide* (https://archive.org/details/requirementsengi0000somm/page/404) (en inglés) (1ra. edition edición). Wiley & Sons. pp. 404 p. (https://archive.org/details/requirementsengi0000somm/page/404) ISBN 978-0471974444. Consultado el 2008.
- Gottesdiener, Ellen; P. Sawyer (2002). *Requirements by Collaboration: Workshops for Defining Needs* (https://archive.org/details/requirementsbyco0000gott) (en inglés). Addison-Wesley Professional. pp. 368 p. ISBN 978-0201786064. Consultado el 2008.
- Sommerville, Ian (2005). *Ingeniería del software* (7ma. edición). Madrid: Pearson Educación S.A. ISBN 84-7829-074-5.

Artículos y revistas

- Weitzenfeld - «El Proceso para Desarrollo de Software» - 2002
- Carlos Reynoso - «Métodos Heterodoxos en Desarrollo de Software» - 2004
- Grupo ISSI - Univ. Politécnica de Valencia - «Metodologías Ágiles en el Desarrollo de Software» - 2003

- Martin Fowler - «La Nueva Metodología» - 2003
- Cutter IT Journal – «Requirements Engineering and Management». August 25, 2000. Cutter Consortium.
- «Software Requirements Engineering», 2nd Edition, IEEE Computer Society. Los Alamitos, CA, 1997 (Compendio de *papers* y artículos en ingeniería de requisitos).
- Lehman, M.M. - «Laws of Software Evolution Revisited», pos. pap., EWSPT96, Oct. 1996, LNCS 1149, Springer Verlag, 1997, pp. 108-124

Véase también

- Ingeniería de *software*
- Programa informático
- Aplicación informática
- Programación
- Fases del desarrollo de *software*
- *Software* colaborativo
- *Software* libre
- Gratis versus libre
- Ingeniería informática
- Hediondez del código

Modelos de ciclo de vida

- Modelo en cascada o secuencial
- Modelo iterativo incremental
- Modelo evolutivo espiral
- Modelo de prototipos
- Modelo de desarrollo rápido

Enlaces externos

- Portal:Software. Contenido relacionado con **Software**.
- Wikimedia Commons alberga una galería multimedia sobre **Software**.
- Wikcionario tiene definiciones y otra información sobre **software**.

Obtenido de «https://es.wikipedia.org/w/index.php?title=Software&oldid=158908821»

-