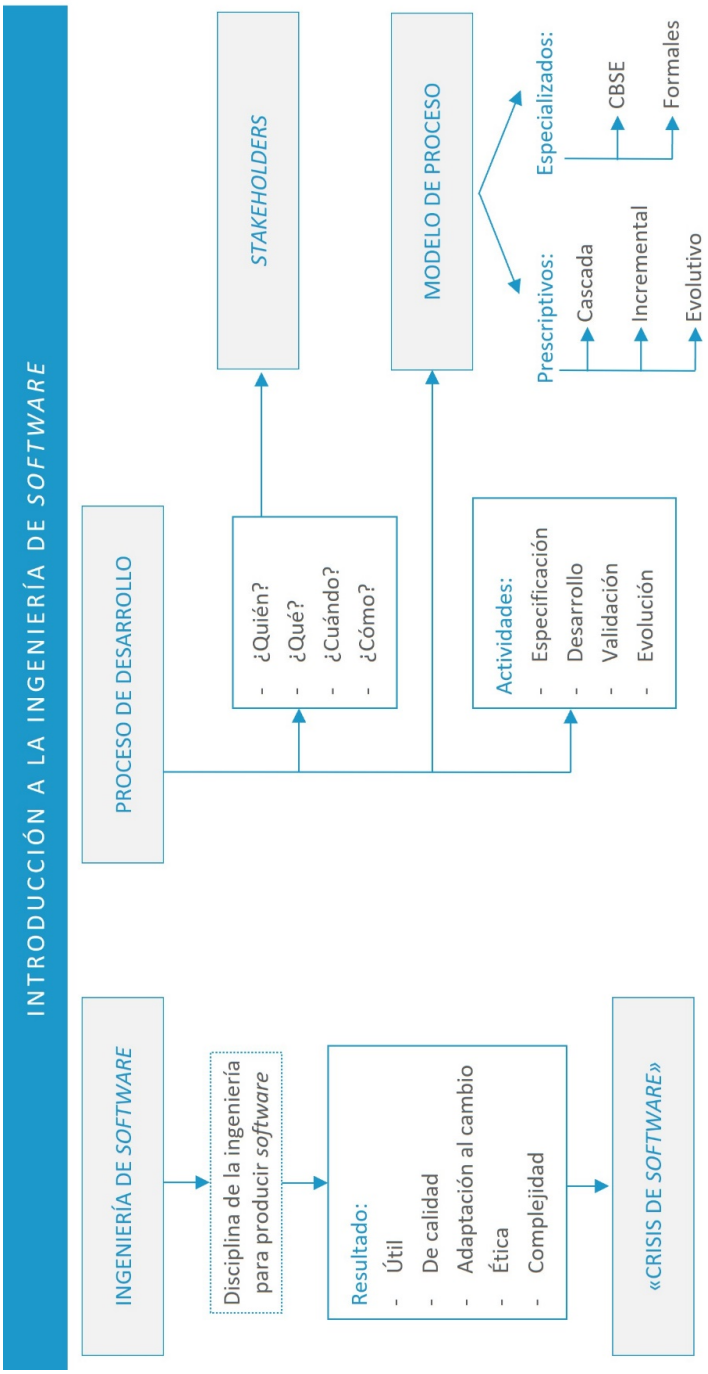
Metodologías, Desarrollo y Calidad en la Ingeniería de Software

Tema 1. Introducción a la ingeniería de software

Índice

Esquema



1.1. Introducción y objetivos

En este tema vamos a estudiar los conceptos básicos asociados a la **ingeniería de *software*** y el motivo de su denominación como «ingeniería». Con el estudio de este tema lo que se pretende es concienciar y entender sobre la necesidad de una disciplina que gestione y formalice el dominio del desarrollo de *software,* con el objetivo de implementar aplicaciones de calidad.

Para ello, se estudiará el significado de **proceso de desarrollo de *software,*** así como los modelos de proceso más comunes que se suelen aplicar a la hora de implementar una aplicación. Las características propias del proyecto *software* harán que, según el caso, se deba utilizar un modelo u otro de proceso.

Con el estudio de este tema pretendemos alcanzar los siguientes objetivos:

- Entender la importancia y necesidad de la disciplina ingeniería de *software* como la aplicación de un enfoque sistemático y disciplinado.
- Comprender el fundamento de la ingeniería de *software* y sus orígenes.
- Entender el proceso de desarrollo de *software* como un conjunto de actividades ordenadas, técnicas y métodos.
- Estudiar el concepto de «modelo de proceso de desarrollo de *software*» para entender la importancia de seleccionar el modelo más adecuado para cada desarrollo.
- Conocer a las personas u organizaciones que directa o indirectamente influyen en todo el proceso del desarrollo del *software.*

1.2. Ingeniería de software

La **ingeniería,** de manera genérica, se puede definir como:

> «[…] la profesión en la que el conocimiento de las ciencias naturales y matemáticas, ganado con estudio, experiencia y práctica, es aplicado con buen juicio para desarrollar formas de utilizar, económicamente, los materiales y las fuerzas de la naturaleza para el beneficio del género humano» (ABET, 1996).

Por otro lado, el ***software*** es «la suma total de los programas de ordenador, procedimientos, reglas, documentación asociada y datos que pertenecen a un sistema de cómputo» (Lewis, 1994). Dicho de otra manera, por el mismo autor, el *software* es «un producto diseñado para un usuario» (Lewis, 1994).

En este sentido, la **ingeniería de *software*** es entendida como:

> «[…] una disciplina de la ingeniería que comprende todos los aspectos de la producción de software, y que abarca desde las etapas iniciales de especificación del sistema, hasta su mantenimiento una vez puesto en funcionamiento» (Sommerville, 2005, p. 6).

A esta definición, cabría añadir, que la ingeniería de *software* ha de tener el objetivo de generar productos *software* de **calidad.**

Otra forma de entender la ingeniería de *software* es como el **estudio de métodos y herramientas** que se pueden utilizar para producir ***software* práctico** (es decir, *software* operacional desarrollado dentro de los límites económicos y organizativos de la empresa desarrolladora), **útil y de calidad** (es decir, *software* correcto y, si es posible, reutilizable).

Sin embargo, a pesar de estar clasificada y tratada como una ingeniería, la ingeniería de *software* tiene unas características muy particulares que le hacen distinguirse de las ingenierías clásicas, las cuales se listan a continuación:

- El producto generado es intangible.

- Implica mucho desarrollo, que no es lo mismo que fabricar en un sentido clásico, por lo que no sigue las pautas ingenieriles.
- Cada solución *software* puede requerir (y normalmente requiere) un enfoque distinto, específico, con uso de diferentes técnicas, tecnologías, herramientas...
- Los requisitos del *software* cambian constantemente. No se puede esperar a que el diseño esté completo para empezar a codificar.
- En mayor o menor medida, requiere describir y documentar los artefactos que se van a generar.

El no tener en cuenta todas estas propiedades inherentes al desarrollo de *software* y tratar de fabricar *software* como si de un producto físico se tratase, hará que surjan inevitablemente problemas a la hora **de planificar, gestionar y desarrollar** la solución. En las ingenierías clásicas hasta que no se tiene el diseño completo no se pasa a la construcción, como ocurre, por ejemplo, cuando se construye un edificio.

Sin embargo, en la ingeniería de *software*, debido a los frecuentes, cambios muchas veces el diseño y la codificación se acaban solapando. Por tanto, cuando se trata de desarrollar *software,* no se puede ser tan tajante en la separación de **diseño y codificación** como ocurre en el resto de las ingenierías en lo que respecta a la separación de diseño y construcción.

Otra diferencia fundamental entre las ingenierías clásicas y la ingeniería de *software* es que los productos fabricados en las ingenierías clásicas presentan un carácter continuo, es decir, pequeñas modificaciones en el producto producirán pequeños cambios en el comportamiento. Sin embargo, los productos *software* suelen tener un carácter discreto, es decir, pequeñas modificaciones en un producto *software* puede desencadenar en una gran cantidad de **efectos colaterales** que podrían producir cambios impredecibles e incontrolados en su comportamiento.

Además, las ingenierías clásicas se basan en unas teorías (normalmente

matemáticas o físicas) que permiten **verificar** las propiedades por medio de cálculo, algo que no se da en la ingeniería de *software,* la Tabla 1 refleja esta situación. En la ingeniería de *software* se parte de muchos requisitos y, a la hora de desarrollar una determinada aplicación, no se sabe realmente qué es lo que se debería calcular para asegurarse de que el *software* va a estar bien construido.

Ingeniería	Producto	Teoría	Propiedades
Civil	Puente	Mecánica clásica	Carga máxima, resistencia lateral...
Aeronáutica	Avión	Dinámica de fluidos	Relación de planeo, turbulencias...
Software	*Software*	¿?	¿?

Tabla 1. Propiedades inherentes de diferentes Ingenierías. Fuente: elaboración propia.

Al fin y al cabo, la ingeniería de *software* es una disciplina que pretende ofrecer **formalidad, sistematización y determinismo** en el proceso de construcción de sistemas *software*. En general, está comprobado que la falta de disciplina o formalidad en un proceso de construcción deriva en productos poco fiables y de poca calidad.

Para lograr esta pretendida formalidad y sistematización, la ingeniaría de *software* enmarca la actividad de desarrollo de *software* en un proceso ordenado por diversas fases, junto con la aplicación de una serie de métodos, técnicas y prácticas basadas en resultados probados, o bases teóricas y sólidas que justifican su utilización.

Actualmente sabemos que no existe un enfoque de ingeniería de *software* ideal y que no será igual para todos los sistemas que se construyen. La gran diversidad de tipos de sistemas *software* y la cantidad de escenarios para su utilización, implica a una amplia gama de técnicas de desarrollo de *software.* Sin embargo, las nociones

fundamentales de **procesos y organización** son comunes para cualquier tipo de técnica de construcción de *software* y constituyen la esencia de la ingeniería de *software.*

Una de las principales labores del ingeniero de *software* será la de seleccionar el conjunto de prácticas y métodos, así como la técnica más apropiada para cada tipo de sistema que se pretenda construir.

Además, independientemente del tipo de sistema, el **ingeniero de *software*** estará **obligado por principios** a intentar adoptar un enfoque sistemático y organizado en su labor, ya que se considera que es la forma más conveniente de llegar al **alcanzar la calidad exigida.**

Otro aspecto que también debe quedar muy claro es que el término 'ingeniero de *software'* **no es sinónimo** de 'programador', aunque en muchos casos se asume que la única responsabilidad de un ingeniero de *software* es la de escribir código bien estructurado (Parnas, 1999) y las ofertas de empleo van en esa línea cuando se demandan ingenieros de *software* (¿para qué se demanda un ingeniero de *software* cuando lo que se necesita es realmente un programador? ¿Es más *cool* decir 'ingeniero de s*oftware"*?).

Además de la programación, un ingeniero de *software* ha de conocer el entorno en el que el producto *software* va a estar operativo y entenderlo a la perfección, para implementar soluciones que cumplan los requisitos y funcionen correctamente en el entorno real. Por tanto, los ingenieros de *software* han de aprender **aspectos científicos** y los métodos necesarios para poder aplicarlos.

Parnas (1999), resume de manera muy precisa las **responsabilidades** de un ingeniero de *software:*

- **Conocer el dominio de la aplicación** para poder definir y documentar los requisitos

que se han de satisfacer, de una manera precisa, bien organizada y que se pueda consultar con facilidad.

- **Participar en el diseño** de la configuración de los sistemas, para determinar qué funcionalidad será llevada a cabo a través del *hardware* y qué funcionalidad será implementada a través de *software.*
- **Analizar el rendimiento del diseño propuesto** (ya sea de manera analítica o a través de simulación), para asegurar que el sistema propuesto cumplirá con los requisitos del cliente.
- **Diseñar la estructura básica del *software*** a través de la división de su funcionalidad en componentes y las interfaces que los interconectan, además de documentar todas las decisiones de diseño que han sido adoptadas.
- **Analizar la consistencia, disponibilidad y completitud** de la estructura del *software* propuesta para implementar la solución.
- Implementar el producto *software* en base a un conjunto de programas bien estructurados y documentados.
- De ser necesario, **integrar la nueva solución** con aplicaciones ya existentes o comerciales.
- **Llevar a cabo las pruebas** necesarias para poder validar y verificar el *software.*
- **Revisar y mejorar los sistemas *software*** sin perder la integridad conceptual y manteniendo actualizada toda la documentación asociada al proyecto.

Por último, hay que comentar que al igual que ocurre con otras disciplinas, la ingeniería de *software* ha de estar dentro de un marco legal, normativo y guiado por **buenas prácticas** que deben limitar la libertad de los ingenieros a la hora de desarrollar *software,* para que puedan ser respetados profesionalmente (Sommerville, 2005, p. 12-13).

Ideas clave

Tal y como afirma Génova *et al.* (2007, p. 3-4): «Sin una sólida educación ética específica de la profesión, el ingeniero se convierte en un mero instrumento técnico y despersonalizado en las manos de otros». Lo que ocurre es que, muchas veces, por las características propias del *software* desarrollado, puede ser complicado determinar las **consecuencias morales** y los **daños a personas** que pueden generar. En cualquier caso, el ingeniero de *software* siempre ha de partir de la premisa de construir *software* que **respete y valore** a las personas de acuerdo a una conducta ética (Génova *et al*., 2007).

En cualquier caso, independientemente de la tendencia ética con la que uno se pueda sentir identificado, existe una necesidad de establecer principios morales que guíen el comportamiento ético en el desarrollo de sistemas *software.* Esta necesidad de establecer unos principios morales en el campo de la ingeniería de *software* ha sido recogida por el Código de Ética y Práctica Profesional de Ingeniería de *Software* de la ACM / IEEE Computer Society, para su adopción como estándar en la enseñanza y práctica general de esta disciplina (ACM & IEEE Computer Society, 1997). En este código se definen ocho principios básicos que deben seguir todos los profesionales de la ingeniería de *software* a la hora de ejercer su profesión:

- **Público.** Los ingenieros de *software* deberán actuar de manera consistente en bien del interés general.
- **Cliente y contratista.** Los ingenieros de *software* deberán actuar de tal modo que sea del mejor interés para sus clientes y contratistas, en coherencia con el interés general.
- **Producto.** Los ingenieros de *software* deberán garantizar que sus productos, y las modificaciones asociadas a ellos, cumplan con el mayor número posible de los mejores estándares profesionales.
- **Juicio.** Los ingenieros de *software* deberán mantener la integridad e independencia en su valoración profesional.

Ideas clave

- **Gestión.** Los gestores y líderes en ingeniería de *software* suscribirán y promoverán un enfoque ético en toda gestión de desarrollo y mantenimiento de *software.*

- **Profesión.** Los ingenieros de *software* deberán realizar avances en lo que se refiere a integridad y reputación de la profesión, en coherencia con el interés general.

- **Compañeros.** Los ingenieros de *software* serán justos y apoyarán a sus compañeros de profesión.

- **Persona.** Los ingenieros de *software* deberán participar en el aprendizaje continuo de la práctica de su profesión y promoverán un comportamiento ético en la práctica de la misma.

1.3. La crisis del software

Durante las últimas décadas la disciplina conocida como ingeniería de *software* se ha enfrentado a diferentes problemas relacionados con el desarrollo de *software,* englobados en lo que se ha denominado **«crisis del *software*»**:

- **Planificaciones imprecisas** que difícilmente se llegan a cumplir, superando los plazos de entrega en la mayoría de los proyectos *software.*
- Los **costes** del proyecto suelen superar con creces el presupuesto inicial.
- Índices de **productividad** muy bajos.
- **Clientes insatisfechos** con las soluciones implementadas, ya que no satisfacen sus necesidades.
- La **calidad** del *software* es inaceptable.
- El producto *software* desarrollado resulta muy **difícil de mantener.**
- El producto *software* **no está integrado** o ni siquiera alineado con el negocio de la compañía.
- El departamento de las Tecnologías de la Información **(TI)** suele verse como un **freno al negocio.**

Como consecuencia de la denominada «crisis del *software*», la ingeniería de *software* surge como concepto en 1968, en un congreso organizado por la Organización del Tratado del Atlántico Norte, OTAN (del inglés *North Atlantic Treaty Organization*, NATO) donde se intentaba explicar el porqué de la bajísima calidad del *software* (NATO, 1968). El *software* era visto como un **añadido** y su programación era totalmente 'artesanal', dependiente exclusivamente del programador, para la que no existía ningún tipo de metodología, ni se realizaba ningún tipo de planificación.

Como conclusión en el congreso, se dedujo que el *software* era un producto tecnológico y que, por tanto, había que tratarlo como **una ingeniería más,** aplicando sus principios, pero trasladados al *software,* acuñándose así el término de 'ingeniería de *software'*.

De esta manera, se pretendía que el ingeniero de *software* aplicara los principios ingenieriles a la hora de construir los productos *software*, en base a un método más sistemático, disciplinado, cuantitativo y menos artesanal.

Con lo cual, se puede decir que la ingeniería de *software* es una **disciplina relativamente joven.** La ingeniería de sistemas, que no hace referencia al mismo dominio que la ingeniería de *software*, es anterior y comprende todos los aspectos del desarrollo y la evolución de sistemas complejos, no solo del *software,* aunque bien es cierto que el *software* representa un papel principal en todo el conjunto.

A lo largo del tiempo, el porcentaje de *software* en los sistemas se ha visto incrementado de manera muy significativa (por ejemplo, los sistemas intensivos de *software* son sistemas en los que aproximadamente el 90 % está constituido por *software*), y las técnicas empleadas en la ingeniería de *software* se están aplicando de manera equivalente en el proceso de ingeniería de sistemas (por ejemplo, el modelado de requisitos con casos de uso y la gestión de la configuración). Precisamente la **evolución del *hardware*** y su caída de precios es lo que está causando que los sistemas de *software* se compliquen cada vez más, y se haga más difícil llegar al final de la mencionada «crisis del software».

Pero **¿por qué es tan difícil desarrollar software?** Si uno se pone a indagar, son muy pocos los proyectos *software* que pueden atribuirse el haber terminado el producto *software* en los plazos estipulados, dentro del presupuesto establecido inicialmente, y además cumpliendo con los requisitos del cliente. En Internet se

pueden encontrar diversas historias de proyectos que han fracasado de manera estrepitosa, con errores que han generado un coste demasiado alto, y no solo desde un punto de vista económico. Por ejemplo, entre los errores informáticos más costosos de la historia se encuentran los que se muestran en la Tabla 2.

AÑO	ENTIDAD	DESCRIPCIÓN
2012	Knight Capital	Un error informático ocasionó la pérdida de 500 millones de dólares en menos de una hora y casi provoca la quiebra de la empresa de inversión Knight Capital. El error fue debido a que sus computadoras comenzaron a comprar y vender millones de acciones sin ningún tipo de control humano.
2010	Toyota	Toyota tuvo que retirar más de 400 000 de sus vehículos híbridos, debido a un problema *software* que provocaba un retraso en el sistema antibloqueo de frenos. Se estima que, entre sustituciones y demandas, el error le costó a Toyota unos 2,8 billones de euros.
2000	Cambio de siglo	Se esperaba que el bug Y2K paralizase al mundo al acabar el año 1999, ya que mucho *software* no había sido previsto para trabajar con el año 2000. El mundo no se acabó, pero se estima que se gastaron unos 300 billones de dólares para mitigar los daños.
1999	NASA	Los ingenieros de la NASA perdieron el contacto con la sonda *Mars Climate Orbiter* en un intento que orbitase en Marte. La causa fue debida a un programa que calculaba la distancia en unidades inglesas, mientras que otro programa utilizaba unidades métricas.
1996	Agencia Espacial Europea (ESA)	El cohete Ariane 5 explotó debido a que un número real de 64 bits en coma flotante, relacionado con la velocidad, se convirtió en un entero de 16 bits. Las pérdidas se estiman en 400 millones de euros.
1994	Intel	Un profesor de Matemáticas descubrió e informó acerca de un fallo en el procesador Pentium de Intel. La sustitución de chips costó a Intel más de 400 millones de euros.
1988	Internet	Un estudiante de posgrado, Robert Tappan Morris, fue condenado por el primer ataque con 'gusanos' a gran escala en Internet. El coste de limpiar el desastre ocasionado por Robert se cifra en unos 100 millones de dólares.
1985	Atomic Energy of Canada Limited	Un error de programación de la máquina de radioterapia Therac-25 causó, entre 1985 y 1987, al menos seis accidentes en los que los pacientes recibieron sobredosis masivas de radiación. Los expertos creen que el fallo fue causado por un error en el código que obligó al programa a realizar la misma acción varias veces.
1978	Hartford Coliseum	Apenas unas horas después de que miles de aficionados abandonaran el estadio Hartford Coliseum (Estados Unidos), el techo se derrumbó por el peso de la nieve. La causa fue debida al cálculo incorrecto que se había introducido en el *software* CAD utilizado para diseñar el estadio.
1962	NASA	La sonda espacial Mariner I se desvió de la trayectoria de vuelo prevista con destino a Venus poco después de su lanzamiento. Desde control se destruyó la sonda a los 293 segundos del despegue. La causa de la desviación fue una fórmula manuscrita que se programó de manera incorrecta. Las pérdidas económicas se estimaron en unos 15 millones de euros.

Tabla 2. Errores informáticos más costosos de la historia. Fuente: Harley, 2018; Garzas, 2013.

La dificultad para satisfacer las necesidades del cliente (¿problemas de incomunicación?), unido a la complejidad del *software* y a sus constantes cambios y evolución, tanto en fase de desarrollo como una vez entregados, hacen de la tarea de desarrollar productos *software* un

proceso arduo y muy costoso (una gran parte del presupuesto en informática se debe a las modificaciones en el *software* ya implementado y desplegado).

El intento de **sistematizar** la ingeniería de *software* para fabricar productos *software* de la misma manera que se construyen **productos físicos** en las ingenierías clásicas ya se ha comprobado que no es la solución ni el camino por seguir, como ya se comentó en el apartado anterior, por las particularidades características del *software,* pero el desarrollo de *software* tampoco es ni mucho menos algo artesanal como en sus orígenes. Es por ello por lo que hay que intentar encontrar el equilibrio, y la ingeniería de *software* es la disciplina que se esfuerza en conseguirlo.

Tal y como afirma Ince (1993), los sistemas *software* son el puro reflejo de la rapidez a la que se mueve el mundo y la **aceleración en los cambios** que se producen: sistemas contables que han de actualizarse porque cambian las leyes fiscales, sistemas de defensa que han de actualizarse a las últimas tecnologías en materia de seguridad, ampliaciones de los sistemas de información como consecuencia de fusiones entre compañías que modifican sus operaciones, etc.

De cara a poder adaptarse a este cambio tan acelerado que se da en la sociedad, Ince (1993) enumera **tres requisitos** que se han de satisfacer en el desarrollo de productos *software* para que sean viables y de calidad:

- Utilización de diferentes técnicas de desarrollo que permitan minimizar la complejidad de un sistema *software.*
- Utilización de métodos y conceptos que permitan a las compañías desarrolladoras de *software* y a sus clientes (ya sean externos o internos de la organización) poder valorar la naturaleza y validez de la solución *software* lo antes posible una vez iniciado el proyecto.

- Utilización de técnicas de desarrollo que permitan minimizar los efectos colaterales que se producen como consecuencia de las modificaciones y evolución del *software*.

El enfoque de Ince (1993) claramente defiende el **enfoque ingenieril** del *software* a través del uso de herramientas y metodologías que permitan obtener un *software* de calidad, pero sin dejar de tener en cuenta las características propias inherentes al *software* a través de métodos que permitan **gestionar los cambios y evolución** del *software* a lo largo del tiempo.

1.4. Proceso de desarrollo de software

> Un proceso, de manera genérica, define «quién está haciendo qué, cuándo, y cómo para alcanzar un determinado objetivo» (Jacobson *et al.*, 2000, prefacio).

Para la ingeniería de *software* dicho objetivo sería el **desarrollo de un producto *software*,** o bien, la modificación o ampliación de uno ya existente. La definición y aplicación de un proceso de desarrollo cuando se inicia un proyecto *software* se debe a ese intento de mejorar la forma de trabajar (más disciplinada y sistemática), favoreciendo la prevención y resolución de posibles problemas que puedan surgir en dicho proceso (Humphrey, 1990, p. 255).

En la literatura se pueden encontrar varias definiciones para el proceso de desarrollo de *software.* **Jacobson, Booch y Rumbaugh** (2000, p. 4), definen el proceso de desarrollo de *software* como «el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema software». Otra definición similar para el proceso de desarrollo de *software* es la proporcionada por **Sommerville** (2005, p. 60), que lo define como «un conjunto de actividades que conducen a la creación de un producto software».

En la misma línea, otra definición más elaborada se encuentra en **Pressman** (2010, p. 27), que describe el proceso de desarrollo como un **conjunto de actividades estructurales,** donde cada una de estas actividades estará compuesta por un conjunto de acciones de ingeniería de *software,* que a su vez vendrán definidas por un conjunto de tareas que identificarán el trabajo a realizar, los productos que se generan en cada trabajo y los puntos de aseguramiento de la calidad que son requeridos, así como los puntos de referencia que se van a utilizar para evaluar el avance del producto *software.*

Por último, bajo otra perspectiva, **Humphrey** (1990, p. 3) define el proceso de desarrollo de software como «el conjunto de herramientas, métodos y prácticas que utilizamos para crear un producto software». Independientemente del enfoque de la definición, los ingenieros de *software* serán los responsables de llevar a cabo gran parte de ese conjunto de actividades y de seleccionar las herramientas, métodos y prácticas más adecuados para desarrollar el *software* requerido.

La ejecución de cualquier proceso de desarrollo de *software* comprenderá un conjunto de actividades, que van a generar artefactos de dos tipos diferentes:

- Artefactos internos del proceso de desarrollo.
- Artefactos entregables al cliente.

Un artefacto se puede definir como toda pieza de información que se utiliza o se produce en un proceso de desarrollo de *software.*

El motivo por el que se siguen definiendo procesos de desarrollo de *software* se debe a un intento de mejorar la forma de trabajar cuando se construye *software.* De este modo, si se piensa en el proceso de desarrollo de una manera ordenada, resultará más sencillo **anticiparse a ciertos problemas** y elaborar formas de **prevenir y resolver** riesgos potenciales. Tal y como se destaca en Humphrey (1990), algunos de los aspectos de mayor interés en los que se centra el proceso *software* tienen que ver con la calidad, la tecnología del producto, la inestabilidad de los requisitos y la complejidad de los sistemas.

Todo proceso de desarrollo de *software* debe **definir el problema** para favorecer su comprensión y diseñar su solución. Para ello, tal y como se muestra en la Figura 1, se partirá de un conjunto de requisitos que se ha de ser capaz de transformar en un producto *software* que los satisfaga.

Figura 1. Transformación de los requisitos de usuario en un producto *software.* Fuente: elaboración propia.

Según Sommerville (2005, p. 7), existen **cuatro actividades** fundamentales que son comunes para todos los procesos de desarrollo de *software:*

- **Especificación del *software.*** Clientes e ingenieros definen el *software* a construir junto con sus restricciones.
- **Desarrollo del *software.*** El *software* se diseña y se implementa, a la vez que se verifica que se construye de manera correcta.
- **Validación del *software.*** El *software* se valida, para comprobar que el software es correcto, es decir, hace lo que el cliente había solicitado. La validación se llevará a cabo teniendo en cuenta los requisitos de usuario.
- **Evolución del *software.*** El *software* se modifica para adaptarse a los cambios requeridos por el cliente o por el mercado.

Cada actividad comprenderá a su vez un **conjunto de tareas** que deberán ser realizadas por recursos concretos, no siempre necesariamente humanos. Estas tareas se podrán organizar de distinta manera en el tiempo, y con diferente nivel de detalle según las características y la complejidad del *software* a desarrollar.

Estas actividades se suelen **complementar** con otras tareas o procesos que se aplican a lo largo de todo el desarrollo: seguimiento y control, gestión del riesgo, aseguramiento de la calidad, revisiones técnicas, medición, gestión de la configuración, gestión de la reutilización, etc.

A continuación, vamos a ver de forma general los **procesos generales** definidos por el cuerpo de conocimiento:

El proceso de toma de requisitos

El proceso de requisitos se refiere a la obtención, análisis, especificación y validación de los requisitos de *software,* así como a la gestión de los requisitos durante todo el ciclo de vida del producto de *software*. Está ampliamente reconocido entre los investigadores y profesionales de la industria que los proyectos de *software* son **extremadamente vulnerables** cuando las actividades relacionadas con los requisitos se realizan de manera deficiente.

Los requisitos de *software* expresan las necesidades y restricciones/limitaciones que se imponen a un producto de *software* que se quiere construir para dar solución de algún problema del mundo real.

El proceso de requisitos para que tenga éxito no debe considerarse como un proceso aislado y discreto realizado al inicio. Los requisitos están **estrechamente relacionados** con el diseño, las pruebas, el mantenimiento...

El proceso de diseño

El diseño se puede definir como el proceso de **definir** la arquitectura, los componentes, las interfaces y otras características de un sistema o componente. El diseño de *software* es la actividad del ciclo de vida de la ingeniería de *software* en la que se analizan los requisitos para producir una descripción de la estructura interna del *software* que servirá de base para su construcción.

Ideas clave

El diseño del *software* juega un papel muy importante en el desarrollo de *software,* durante el diseño, los ingenieros de *software* producen varios **modelos** que forman una especie de **borrador de la solución** que se implementará. Estos modelos se pueden analizar y evaluar para determinar si nos permitirán o no cumplir con los requisitos del proyecto.

El proceso de construcción

> El término 'construcción de *software'* se refiere a la creación detallada del *software* de trabajo mediante una combinación de codificación, verificación, prueba de unidad, prueba de integración y depuración.

El proceso de construcción de *software* está fuertemente vinculado al proceso de **diseño de software** y al de **pruebas de *software,*** porque el proceso de construcción de *software* implica un diseño y unas pruebas importantes del *software.*

La construcción de *software* generalmente produce la mayor cantidad de **elementos de configuración** que deben administrarse en un proyecto de *software* (archivos de origen, documentación, casos de prueba, etc.). Por lo tanto, el proceso de construcción también está estrechamente vinculado al **proceso de configuración.**

El proceso de pruebas

Las pruebas de *software* consisten en la **verificación dinámica** de que un programa proporciona el comportamiento esperado en un conjunto finito de casos de prueba, adecuadamente seleccionados del dominio de ejecución generalmente bastante grande como para poder ser probado en su totalidad.

En los últimos años, la visión de las pruebas de *software* ha **madurado.** Las pruebas ya no se ven como una actividad que comienza solo después de que la fase de codificación se completa y con el propósito limitado de detectar fallos.

Las pruebas de *software* están, o deberían estar, generalizadas a lo largo de todo el ciclo de vida de desarrollo y mantenimiento.

De hecho, la planificación de pruebas debe comenzar con las etapas iniciales del proceso de requisitos de *software* y, los planes y procedimientos de prueba, deben desarrollarse sistemática y continuamente, y posiblemente refinarse a medida que avanza el desarrollo del *software.* Estas actividades de planificación de pruebas y diseño de pruebas proporcionan **información útil** para los diseñadores de *software* y ayudan a resaltar posibles puntos débiles, como descuidos, incoherencias de diseño, omisiones y ambigüedades en la documentación.

El proceso de mantenimiento

El desarrollo de *software* da como resultado la entrega de un producto de *software* que satisface los requisitos del usuario. Seguramente el producto de *software* debe cambiar o evolucionar. Una vez en funcionamiento, se descubrirán defectos, los entornos operativos cambiarán y surgirán nuevos requisitos para el usuario. La **fase de mantenimiento** comienza después del período de arranque o la entrega después de la implementación, pero las actividades de mantenimiento se comienzan a realizar desde mucho antes.

El mantenimiento del *software* se define como la totalidad de las actividades requeridas para proporcionar un soporte rentable al *software.* Las actividades se realizan durante la etapa previa a la entrega, así como durante la etapa posterior a la entrega. Las **actividades previas** a la entrega incluyen la planificación de las operaciones posteriores a la entrega, la capacidad de mantenimiento y la determinación logística para las actividades de transición. Las **actividades posteriores** a la entrega incluyen la modificación del *software,* la capacitación y el funcionamiento o la interconexión con un servicio de asistencia.

Proceso de gestión de la calidad

La gestión de la calidad del *software* se forma por medio de un conjunto de procesos que garantizan que los productos de *software,* los servicios y las implementaciones del proceso del ciclo de vida **cumplan con los objetivos de calidad** del *software* de la organización y logren la satisfacción de los interesados.

La **garantía de calidad** define procesos, requisitos para los procesos, mediciones de los procesos y sus resultados, y canales de retroalimentación a lo largo de todo el ciclo de vida del *software.*

1.5. Modelo de proceso de desarrollo de software

Un modelo de proceso de desarrollo de *software* es una descripción simplificada de un proceso de desarrollo de *software* real.

Aunque se trata de una representación simplificada del proceso *software,* precisamente esta característica constituye una de sus principales ventajas: un **modelo** de proceso *software* debería ser **fácil de entender y de seguir** por todos los desarrolladores que participan en un proyecto determinado.

Ante un proyecto *software,* las organizaciones adoptarán el **modelo de proceso** *software* que mejor se adapte, lo que dependerá de la complejidad y las características del sistema a construir. Hay que tener en cuenta que la elección inadecuada de un modelo de proceso puede tener un gran impacto en la calidad o utilidad del sistema *software* construido.

Existen una serie de configuraciones canónicas a las que podemos denominar **modelos de proceso prescriptivos,** son las más antiguas históricamente y trataron de poner orden en los procesos de desarrollo de *software* adoptando un enfoque muy formal. Alrededor de estos modelos se desarrollaron un número de técnicas tradicionales, entre las que podemos citar: ***Rational Unified Process*** (Kruchten, 2003), ***Microsoft Solutions Framework*** (Turner, 2006) o **Métrica v.3** (Consejo Superior de Informática, 2001).

Más recientemente, han ido apareciendo **nuevos modelos** que relajan las imposiciones y simplificaciones de otros modelos tradicionales, con el fin de permitir que los procesos de desarrollo se adapten con mayor velocidad y menor costo a los posibles cambios e imprevistos de un proyecto.

Estos **modelos de procesos ágiles** siguen una serie de principios generales, pero

dejan gran libertad en cuanto a las particularidades de su implementación en cada proyecto concreto. Por este motivo, muchos autores no consideran las técnicas ágiles como auténticas metodologías, pues dejan muchos aspectos sin concretar y no siempre responden a las preguntas: ¿quién?, ¿cuándo? y ¿cómo? Pero este es, precisamente, su fundamento y origen, tratar de ofrecer herramientas para **facilitar la adaptación al cambio** en los proyectos de desarrollo (ver Figura 2).

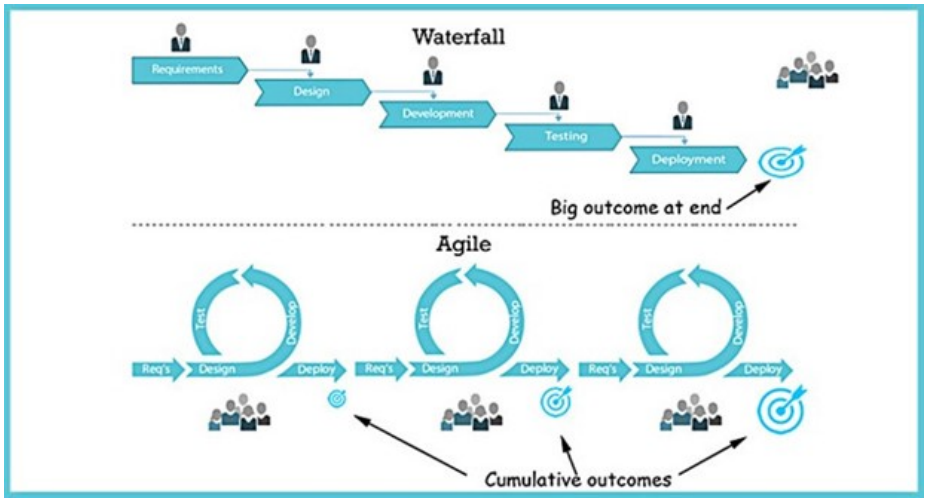


Figura 2. Comparativa entre el enfoque tradicional (waterfall o en cascada) y el enfoque ágil. Fuente: Linkedin, 2019.

Modelos de proceso prescriptivos

Estos modelos, y las técnicas que en ellos se apoyan, se preocupan por definir de manera precisa y detallada las distintas actividades y tareas, con el objetivo de desarrollar *software* de calidad estableciendo fechas y presupuestos concretos. En todo momento se busca **minimizar la incertidumbre** asociada al proyecto.

Modelo en cascada

El modelo en cascada (*waterfall model*) fue el **primer paradigma** de proceso de desarrollo de *software* reconocido (ver Figura 3), y se deriva de otros procesos de ingeniería utilizados para la construcción de productos físicos (Royce, 1970).

Este modelo toma las actividades comunes para todos los procesos de desarrollo de *software* (especificación, desarrollo, validación y evolución) y las representa como fases individuales secuenciales: especificación de requisitos, diseño de *software,* implementación, pruebas, etc. Una fase no puede comenzar hasta no haber finalizado (aprobado) la anterior.

Desgraciadamente, este modelo de proceso resulta **poco realista,** ya que difícilmente se van a encontrar proyectos *software* reales que se ajusten a este tipo de modelo de proceso de manera tan rigurosa.

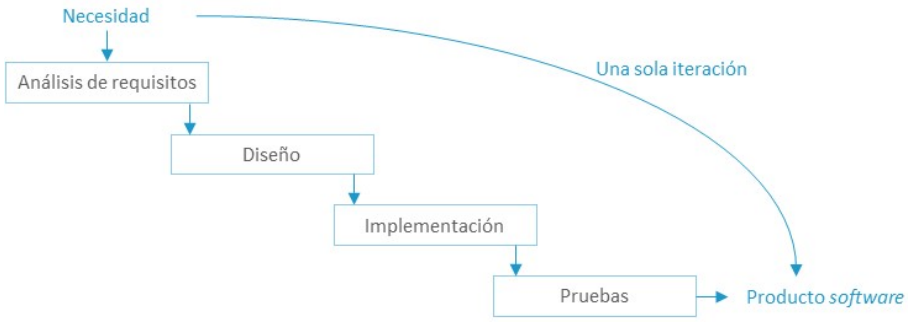


Figura 3. Ciclo de vida del desarrollo de software en cascada. Fuente: elaboración propia.

Este modelo solo debería aplicarse en procesos donde los requisitos están claramente definidos desde un comienzo y es poco probable que cambien. Su **poca flexibilidad** hace que plantee muchos problemas en determinados escenarios (Pressman, 2010):

- Los proyectos reales rara vez siguen un flujo puramente lineal, siendo necesarias las iteraciones y revisiones de fases previas, lo cual es difícil de gestionar según este modelo.
- Muchos proyectos no están claramente definidos desde un comienzo, y solo se dispone de una especificación de requisitos muy genérica y parcial.

Ideas clave

- Se debe esperar hasta la finalización del proyecto para que el cliente disponga de una versión funcional del producto.

Modelo incremental

Para resolver algunos de los problemas que plantea el modelo en cascada, surge este modelo. Está orientado a construir **versiones incrementales del producto,** de manera que el *software* funcional llegue cuanto antes al cliente y este pueda dar una retroalimentación.

El primer incremento puede contener solo un conjunto de funcionalidades básicas, mientras que a medida que progresa el proyecto se va aumentando la funcionalidad y complejidad del producto. Dentro de cada incremento se puede seguir un flujo puramente lineal, iterativo o incluso paralelo, pero el objetivo fundamental es liberar versiones del producto que **resulten útiles.**

Este modelo tiene una serie de ventajas con respecto al modelo en cascada (Sommerville, 2011) y también presenta algunos problemas como se puede ver en la Tabla 3.

VENTAJAS E INCONVENIENTES DEL MODELO INCREMENTAL

VENTAJAS	INCONVENIENTES
▸ Facilita la retroalimentación del cliente. ▸ Reduce el coste de adaptación de requisitos. ▸ Reduce el esfuerzo de análisis y documentación o, al menos, lo distribuye. ▸ Se mejora el TTM (*Time To Market*) y el ROI (*Return Of Investment*), puesto que el cliente dispone antes de versiones funcionales de las que puede comenzar a obtener un rendimiento.	▸ Se pierde visibilidad y control sobre el proceso global. ▸ Las continuas modificaciones sobre el sistema pueden hacer que la estructura general del *software* tienda a degradarse si no se aplican técnicas específicas de ingeniería, como la refactorización, lo cual implica tiempo y recursos adicionales. ▸ Ciertas organizaciones de gran tamaño requieren de procedimientos burocráticos más estrictos y en ocasiones están sujetas a regulaciones externas.

Tabla 3. Ventajas e inconvenientes del modelo incremental. Fuente: elaboración propia.

Modelo evolutivo

Este modelo es similar al incremental, pero se centra más en la **evolución del sistema,** tanto en su implementación como en la elaboración de los requisitos cuando en un comienzo no resultan del todo claros.

Se trata de ir mejorando el *software* de manera progresiva e iterativa, al mismo tiempo que se profundiza en su comprensión empleando herramientas como la construcción de prototipos. Un modelo evolutivo es el modelo en espiral.

El modelo en espiral constituye una técnica propuesta por **Boehm** (1988) y representa el proceso *software* como una **espiral** (ver Figura 4), que comprende a la vez el modelo iterativo y el modelo incremental (ver Figura 5).

De este modo, este modelo de proceso está diseñado explícitamente para soportar iteraciones en el proceso, donde en cada iteración se va añadiendo más funcionalidad al sistema (es decir, un incremento) hasta llegar a una iteración que contiene el **sistema completo.**

Además, en cada **bucle** de la espiral se representa **una fase** del proceso *software.* El bucle más interno trata la viabilidad del sistema, el siguiente bucle la definición de los requisitos del sistema, el siguiente bucle el diseño del sistema, y así sucesivamente.

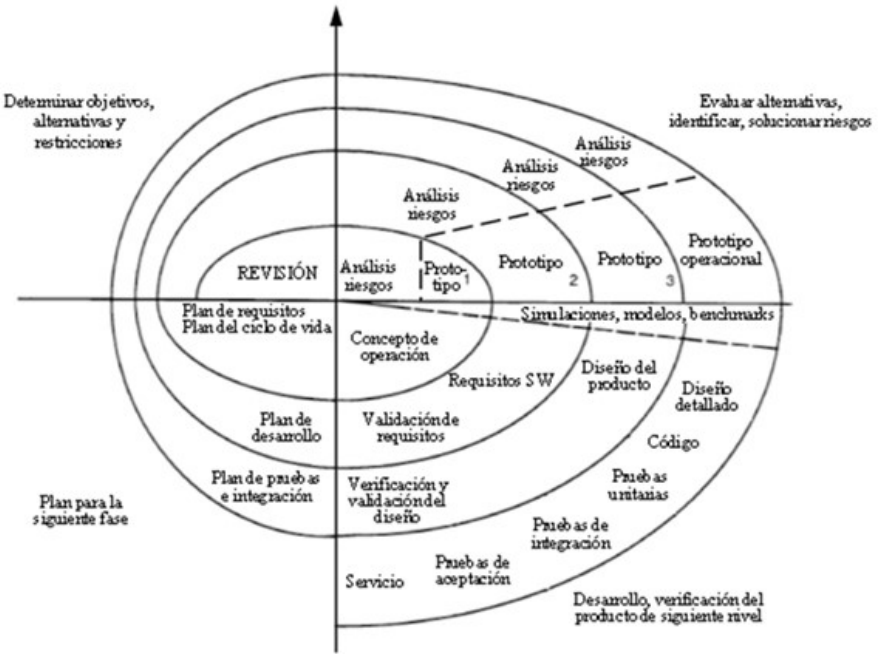


Figura 4. Modelo en espiral (iterativo + incremental) Fuente: Boehm, 1988.

Por otro lado, cada bucle de la espiral se divide en cuatro sectores:

- **Establecimiento de los objetivos,** donde se definen los objetivos específicos para cada fase del proyecto.
- **Evaluación y reducción de riesgos,** donde se lleva a cabo un análisis detallado de cada uno de los riesgos del proyecto que se hayan identificado.
- **Desarrollo y validación,** donde una vez que se ha llevado a cabo la evaluación de riesgos se selecciona una técnica de desarrollo para el sistema (por ejemplo, el modelo en cascada podría ser el desarrollo más apropiado si el riesgo identificado más relevante es la integración de subsistemas).
- **Planificación,** donde se revisa el proyecto y se toma la decisión de si se ha de continuar con un nuevo bucle de la espiral. Si se decide continuar, entonces se planifica la siguiente fase del proyecto.

La distinción más importante entre el desarrollo en espiral y otras técnicas de

procesos de *software* es la consideración explícita de riesgos en el proceso en espiral.

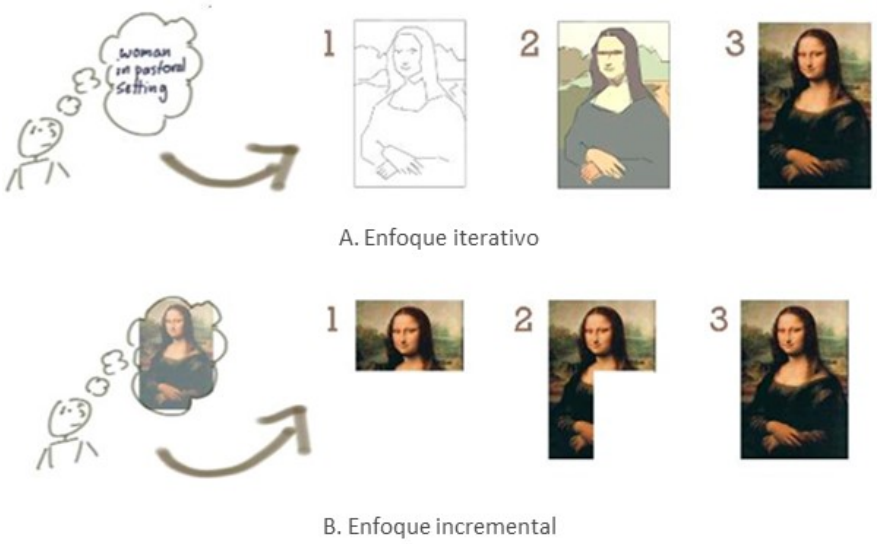


Figura 5. Iterativo vs Incremental. Fuente: Jan, 2008.

Modelos de proceso especializados

Estos modelos comparten muchas características con los modelos anteriores, o combinaciones de estos, pero se orientan a enfoques del proceso de ingeniería mucho más específicos.

Desarrollo basado en componentes

En todo proceso de desarrollo se realiza algún tipo de reutilización, aunque sea de manera informal. Hoy en día, cada vez es más necesaria la aplicación de **técnicas de reutilización** —y posible, dada la gran variedad de código disponible—, sobre todo cuando pensamos en sistemas grandes y complejos. Es posible reducir el tiempo de desarrollo y el coste del proyecto y, en general, aumentar la fiabilidad y seguridad al emplear componentes ya probados.

Aunque sea de **manera informal,** en la mayoría de los proyectos *software* siempre se va a dar algo de reutilización, ya que es muy raro que las organizaciones construyan sistemas completamente nuevos, es decir, que partan de cero (Sommerville, 2005, p. 64-66).

Bajo esta perspectiva, el modelo de ingeniería de *software* basada en componentes (*Component-Based Software Engineering*, CBSE), orientado a la reutilización, se basa en la existencia de un número significativo de componentes reutilizables. CBSE define una **arquitectura de colaboración** e implementa el código necesario para que los componentes funcionen unos con otros.

En este contexto, un componente se define como:

> «[…] un elemento software que se adecúa a un modelo de componentes y que puede ser distribuido de manera independiente e integrado sin ningún tipo de modificación siguiendo un estándar de composición» (Heineman y Council, 2001).

Un componente *software* es un fragmento de código que encapsula un conjunto de funcionalidades específicas y relacionadas entre sí, que se exponen a través de interfaces estandarizadas, de manera que se facilita su reutilización.

Para CBSE, el proceso de desarrollo del sistema se centra, por tanto, en **integrar**

estos componentes de manera sistemática en vez de desarrollar el sistema partiendo de cero. El modelo de proceso genérico para CBSE se muestra en la Figura 6.

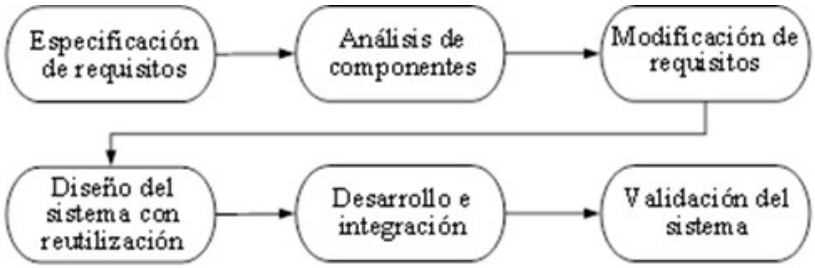


Figura 6. Ingeniería de software basada en componentes. Fuente: Sommerville, 2005.

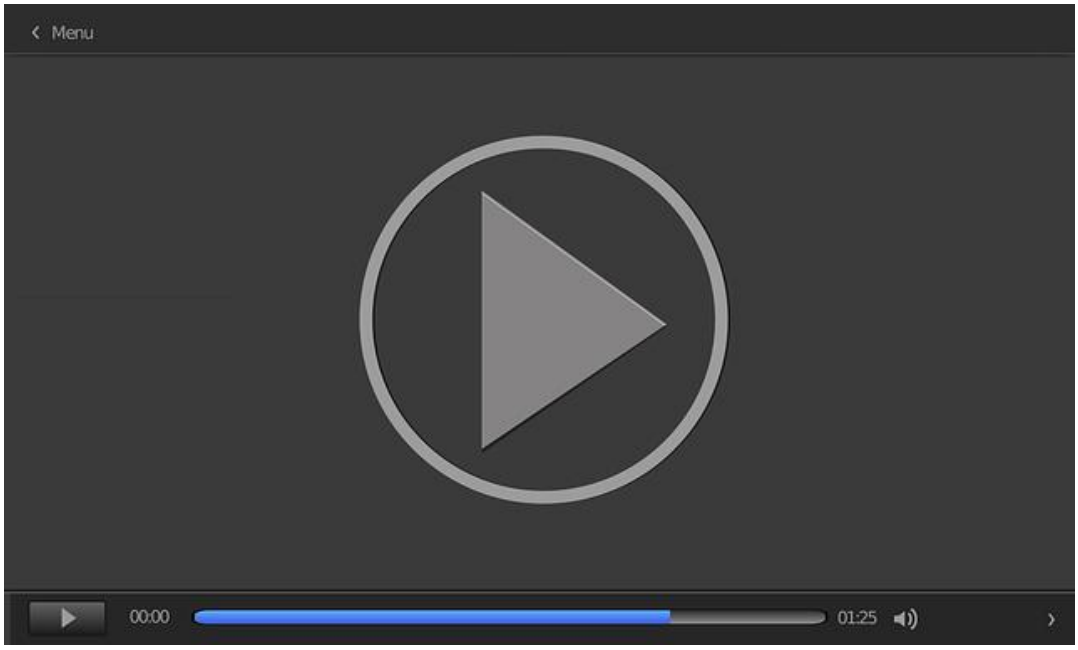
Modelo de métodos formales

Se trata de una variación del modelo en cascada que parte de una representación y modelado muy precisos, en lenguaje matemático, del sistema que se va a desarrollar.

Según Sommerville (2011): «son apropiados en sistemas con fuertes requisitos de seguridad, fiabilidad o protección» (p. 32). El objetivo es lograr un producto **libre de defectos,** precisando para ello de un gran consumo de tiempo y recursos, y unas capacidades específicas de los desarrolladores. Además, esta especificación formal dificulta la comunicación con el cliente (Pressman, 2010).

A continuación, en la lección magistral *Modelos de proceso en metodologías ágiles,* se realiza una introducción a las principales características de los modelos de proceso habituales en técnicas ágiles. En un primer momento, se revisa el modelo tradicional de desarrollo de *software* en cascada, seguido se presenta el manifiesto ágil que veremos más adelante y, finalmente, se analizan las principales características de los modelos de proceso ágiles.

Ideas clave



Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=aeb65e2e-c35a-43fd-a55a-ae0101129226

1.6. Los stakeholders

Un *stakeholder* es cualquier persona o colectivo que tiene algún tipo de interés o participación en el desarrollo de un nuevo producto o servicio, o en el propio proyecto.

Más formalmente:

> «Un stakeholder es una persona, grupo u organización que está involucrada en el proyecto, resulta afectada por el proceso o su resultado, o puede influir en el proceso o su resultado» (Wiegers y Beatty, 2013, p. 27).

Clasificación de los *stakeholders*

Pueden ser figuras internas o externas al equipo de desarrollo. En la Figura 7 se muestran muchos de los *stakeholders* que podemos encontrar en un proyecto genérico, aunque en la mayoría solo aparecerá un subconjunto de ellos. Una de las primeras tareas del analista de requisitos será la **identificación de estos agentes,** intentando no olvidar a ninguna comunidad importante.

En una primera aproximación podemos distinguir entre dos tipos de *stakeholders*: los clientes y «otras partes interesadas».

- El **cliente** es «la organización o el individuo que ha solicitado la construcción de un producto y lo recibirá finalmente» (NASA, 2017, p. 179). Es la figura que solicita el desarrollo, paga por él, selecciona sus características, recibirá o utilizará el producto generado. No siempre coincide con el patrocinador, que es el encargado de financiar el desarrollo. Se suele incluir dentro de este grupo a usuarios directos e indirectos, patrocinadores ejecutivos y compradores.

Figura 7. Potenciales *stakeholders* dentro del equipo, dentro de la empresa desarrolladora y fuera de esta.

Fuente: adaptada a partir de Wiegers y Beatty, 2013.

- Las **otras partes interesadas** son:

> «[…] grupos o individuos que no son clientes directos, pero pueden ser afectados de algún modo por el producto resultante, por la manera en que el producto es realizado o utilizado, o tienen alguna responsabilidad proporcionando servicios durante el ciclo de vida del producto» (NASA, 2017, p. 186).

Una figura importante es la del sponsor o **patrocinador ejecutivo.** Es el agente encargado de financiar el proyecto o de encontrar la financiación para el proyecto (PMI, 2014). En muchas compañías comerciales y servicios públicos los recursos financieros son a veces escasos. Diferentes departamentos compiten por estos recursos, y es posible que proyectos financiados sean vistos con recelo por otros departamentos dentro de la misma organización e intenten buscar debilidades con el objetivo de que sean cerrados y se liberen sus recursos.

Los **usuarios** (o usuarios finales) son un tipo importante de cliente, puesto que serán quienes utilicen finalmente el producto de manera directa o indirecta. Son una figura fundamental porque deben expresar qué tipo de tareas necesitan realizar con el producto, qué resultados esperan y qué características de calidad esperan del sistema.

- Los **usuarios directos** trabajarán directamente con el producto, interactuando con él y manipulando sus capacidades.
- Los **usuarios indirectos** se beneficiarán indirectamente del producto, consumiendo informes u otros productos generados a partir del sistema o utilizándolo a través de otras aplicaciones.

Clases de usuarios

En general cualquier producto será utilizado por una amplia variedad de usuarios. No podemos hablar de «el usuario», sino que el primer paso será identificar qué **categorías de usuarios** debe analizarse. Los diferentes tipos de usuarios finales que podemos encontrar en un sistema son casos particulares de los usuarios del producto, que a su vez particularizan la categoría de clientes del producto, que son un tipo especial de *stakeholder,* como se muestra en la Figura 8. Al mismo tiempo, una persona puede pertenecer simultáneamente a **varias clases de usuario.**

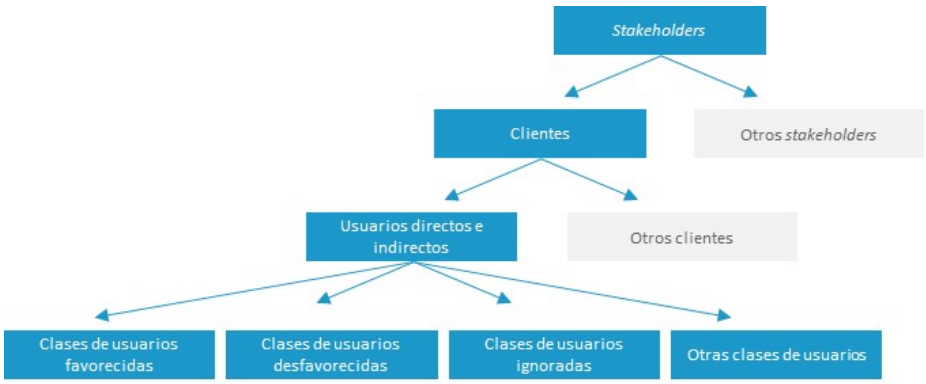


Figura 8. Representación jerárquica de stakeholders, clientes, usuarios, y tipos de usuario. Fuente: adaptada a partir de Wiegers y Beatty, 2013.

Una manera de clasificar los usuarios pasa por identificar sus **diferencias** en estos aspectos:

- El nivel de acceso y privilegios en el sistema (usuarios ordinarios, usuarios invitados, administradores).
- El tipo de tareas que realizan con el sistema y las características que utilizan.
- La frecuencia con que utilizan el producto.
- Su nivel de competencia tecnológica y habilidades para interactuar con el sistema.
- La plataforma en la que utilizan el producto (PC de escritorio, ordenador portátil, Smartphone y otros dispositivos).

Otra manera de enfocar el problema es atendiendo al **beneficio esperado** por las diferentes clases de usuarios. Así podemos identificar:

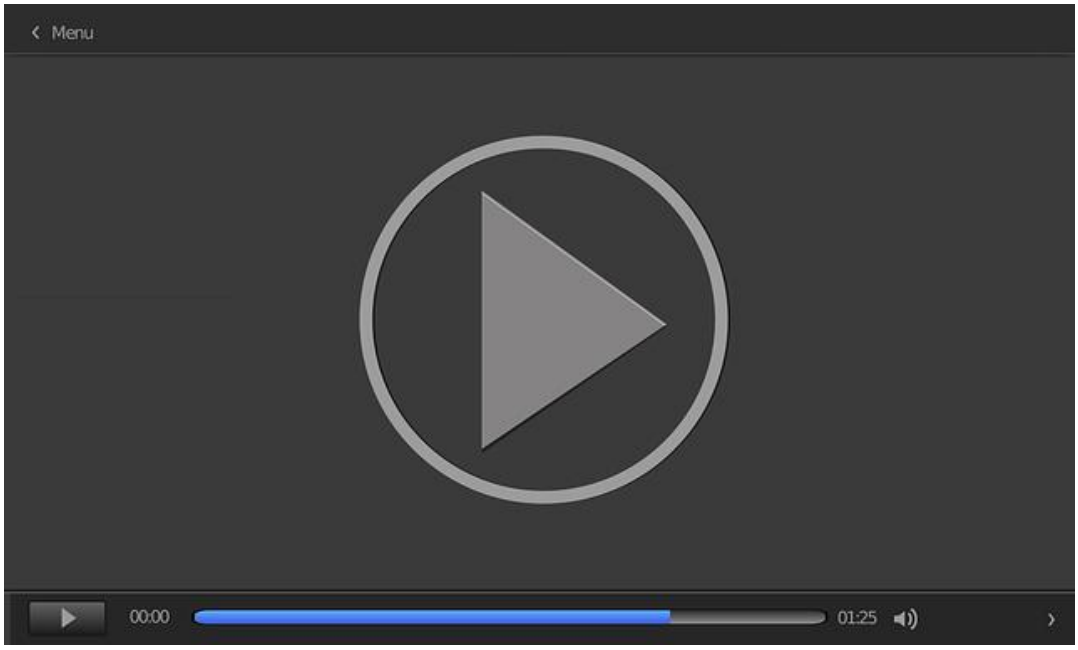
- **Usuarios favorecidos** por el sistema, cuya satisfacción se verá mejorada en la medida que sus intereses están alineados con los objetivos de negocio de la organización. En caso de conflicto con otros intereses, la opinión de estos usuarios se debe ver favorecida y tenida en cuenta.
- **Usuarios desfavorecidos** por el sistema. Son aquellos que no se espera que lo utilicen por razones legales o de seguridad. Se trata de incluir características que inhabiliten o dificulten el acceso de estos usuarios al producto, mediante controles de acceso y otras protecciones de seguridad.
- **Usuarios ignorados.** Se puede elegir no tener en consideración los deseos y necesidades de otros grupos de usuarios. Tal vez porque resulta imposible predecir la totalidad de perfiles que utilizarán el sistema o porque lo harán de manera muy marginal o limitada en el tiempo.

- **Otras clases de usuarios.** Además de todo lo anterior, es posible identificar grupos de usuarios relevantes que no serán favorecidos ni desfavorecidos por el sistema, pero que deben tenerse en consideración. Por ejemplo, el equipo de mantenimiento del sistema no pertenece a ninguna de estas dos categorías, pero tienen puntos de vista muy relevantes a la hora de construir el producto.

Los usuarios no siempre son seres humanos, pueden ser **otros sistemas** o agentes que acceden al servicio en representación de un humano, como los *bots.* Pueden ser sistemas informáticos que inspeccionan un sitio web buscando información sobre productos, servicios o personas para indexarla, o simplemente para detectar vulnerabilidades.

Ideas clave

A continuación, en la lección magistral *Clases de usuarios,* se profundiza en los tipos de usuarios que podemos encontrar.



Accede al vídeo:

https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=eb1b8e73-eecd-4a85-b219-ae0100fe5da9

1.7. Referencias bibliográficas

ACM & IEEE Computer Society (1997). *Software Engineering Code of Ethics and Professional Practice.* Committee on Professional Ethics. https://ethics.acm.org/code-of-ethics/software-engineering-code/.

Boehm, B. (1988). A Spiral Model of Software Development and Enhancement. *Journal of IEEE Computer, 21*(5), pp. 61-72.

Consejo Superior de Informática (2001). *Métrica versión 3*. PAe. https://administracionelectronica.gob.es/pae_Home/pae_Documentacion/pae_Metodolog/pae_Metrica_v3.html#.W1IuUNgzbUI.

Garzas, J. (2013). Top 8 de errores informáticos más costosos de la historia. Javier Gaarzas https://www.javiergarzas.com/2013/05/top-7-de-errores-informaticos.html.

Génova, G., González, M. R. y Fraga, A. (2007). Ethical education in software engineering: responsibility in the production of complex systems. *Journal on Science and Engineering Ethics*.

Heineman, G. y Council, W. (2001). *Component-Based Software Engineering - Putting the Pieces Together.* Addison-Wesley.

Humphrey, W. S. (1990). *Managing the Software Process*. Addison-Wesley.

Ince, D. C. (1993). *Ingeniería de software*. Addison-Wesley Iberoamericana.

Ismailwala, I. (2019). *Transforming from Waterfall to Agile.* Linkedin https://www.linkedin.com/pulse/transforming-from-waterfall-agile-iqbal-ismailwala-/.

Jacobson, I., Booch, G. y Rumbaugh, J. (2000). *El Proceso Unificado de Desarrollo de Software.* Addison Wesley.

Jan. (2008). *Difference between iterative and incremental development*. Blogger. http://jan-so.blogspot.com.es/2008/01/difference-between-iterative-and.html.

Kruchten, P. (2003). *The Rational Unified Process: An Introduction* (3ª ed.). Addison-Wesley.

Lewis, G. (1994). What is Software Engineering? *DataPro* (4015), 1-10.

NASA. (2017). *NASA Systems Engineering Handbook. Nasa.* https://www.nasa.gov/connect/ebooks/nasa-systems-engineering-handbook.

Martin, D. (2022). 11 of the most costly software errors in history. Raygun. https://raygun.com/blog/costly-software-errors-history/.

Naur. P. y Randell, B. (Eds.). (1968). *SOFTWARE ENGINEERING, Report on a conference sponsored by the NATO SCIENCE COMMITTEE Garmisch, Germany, 7th to 11th October 1968.* NATO Science Committee.

Naur, P. y Randell, B. (Eds.). (enero de 1969). Software Engineering. Report of the NATO Software Engineering Conference 1968 sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October 1968. Scientific Affairs Division, NATO.

Página de ABET.

Parnas, D. L. (1999). Software Engineering Programs Are Not Computer Science Programs. *Journal IEEE Software, 16*(6), pp. 19-30.

PMI. (2014). *La participación de los patrocinadores ejecutivos: principal piloto del éxito de proyectos y programas*. Project Management Institute. https://www.pmi.org/-/media/pmi/documents/public/pdf/learning/thought-leadership/pulse/executive-sponsor-engagement.pdf?sc_lang_temp=es-ES.

Pressman, R. S. (2010). *Ingeniería del software. Un enfoque práctico*. (7ª ed.).

McGraw-Hill.

Royce, W. W. (1970). Managing the development of large software systems. En *Proceedings* (pp. 328-338). IEE WESCON.

Sommerville, I. (2005). *Ingeniería del Software* (7ª ed.). Pearson Addison-Wesley.

Sommerville, I. (2011). *Ingeniería de software* (9ª ed.). Pearson Educación de México.

Turner, M. S. (2006). *Microsoft Solutions Framework Essentials*. Microsoft Press.

Wiegers, K.E. y Beatty, J. (2013). *Software requirements* (3ª ed.). Microsoft Press.

Ingeniería de software

Sommerville, I. (2011). *Ingeniería de software* (9ª ed.). Pearson Educación de México.

Libro orientado a estudiantes de ingeniería de *software,* así como a los profesionales del *software,* donde se da una visión general de los aspectos más relevantes vinculados a la ingeniería del *software.*

Guía del cuerpo de conocimiento sobre ingeniería de software SWEBOK

Página de Swebok.

Cada profesión tiene un cuerpo de conocimiento específico. Esta guía pretende promover una visión consistente del área de ingeniería de *software*.

Software engineering

Bauer, F.L., Bolliet, L. y Helms, H.J. (1969). Software Engineering. *Report on a conference sponsored by the NATO Science Committee. Germany*, 7-11. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

En este informe se recogen las actas de la NATO *Software Engineering Conference* de 1968, en la que el desarrollo de *software* pasó a clasificarse como ingeniería en un intento de sistematizar el proceso de desarrollo de *software* y acabar con la denominada «crisis del *software*».

¿Cuándo usar metodologías ágiles? Ágiles vs Tradicionales

Garzas, J. (2020) *¿Cuándo usar «»metodologías»» ágiles? Ágiles vs Tradicionales.* JavierGarzas https://www.javiergarzas.com/2020/01/metodologias-agiles-vs-tradicionales.html.

En este artículo se analiza de una manera sencilla, ¿cuándo sería conveniente el uso de técnicas ágiles y cuándo tradicionales? Hay enlaces directos a vídeos con explicaciones y ejemplos interesantes.

Test

1. ¿Qué es la ingeniería de *software*?

A. Es una metodología para desarrollar *software* de calidad.

B. Es una disciplina que permite desarrollar *software* de calidad.

C. Es una disciplina que solo contempla el desarrollo del *software,* pero no su mantenimiento.

D. Es una metodología que contempla todo el proceso de producción del *software.*

2. A la hora de desarrollar un producto *software:*

A. Hasta que no se termina el diseño jamás se empieza con la codificación.

B. No se hace ninguna estimación.

C. Diseño y codificación pueden solaparse en el tiempo.

D. No se tiene en cuenta el entorno en el que va a funcionar.

3. Señala la respuesta correcta sobre las características de un producto *software:*

A. Es algo tangible.

B. No evoluciona con el tiempo.

C. Los cambios en el *software* pueden tener efectos colaterales.

D. Ninguna de las respuestas anteriores es correcta.

4. ¿Qué afirmación tiene relación con la «crisis del *software*»?

A. Las empresas ya no tienen tanta necesidad de productos *software* y hay crisis en el sector.

B. El *software* tiene una calidad muy baja, pero cumple con las planificaciones realizadas.

C. Los proyectos *software* no cumplen con las planificaciones realizadas.

D. El *software* tiene una alta calidad y está alineado con el negocio.

5. ¿Qué es un artefacto *software?*

A. Pieza de información que se produce o se necesita en un proceso de desarrollo de *software.*

B. Cada una de las etapas que se dan un proceso de desarrollo de *software.*

C. Fecha de referencia en una etapa dentro del proceso de desarrollo de *software.*

D. Ninguna de las definiciones anteriores es correcta.

6. ¿Qué define un proceso?

A. Un proceso define quién está haciendo qué, cuándo y cómo para alcanzar un determinado objetivo.

B. Un proceso define quién está haciendo qué, pero no cuándo ni cómo.

C. Un proceso define quién está haciendo qué, cuándo, pero no cómo.

D. Un proceso no define los pasos a seguir para desarrollar *software.*

7. El objetivo principal del proceso de desarrollo de *software* es:

A. Reducir el coste del proyecto.

B. Gestionar los recursos humanos del proyecto.

C. Mejorar la forma de trabajar a la hora de desarrollar *software,* favoreciendo la prevención y resolución de posibles problemas que puedan surgir.

D. Reducir el tiempo de desarrollo de un proyecto.

8. ¿Qué es un modelo de proceso *software?*

A. Un modelo de proceso de desarrollo de *software* es un prototipo del sistema a desarrollar.

B. Un modelo de proceso de desarrollo de *software* es el proceso de desarrollo específico que se aplica en un proyecto *software* determinado.

C. Un modelo de proceso de desarrollo de *software* es una descripción detallada del sistema a implementar.

D. Un modelo de proceso de desarrollo de *software* es una descripción simplificada de un proceso de desarrollo de *software* real.

9. El modelo en espiral:

A. Es iterativo e incremental.

B. Es equivalente al modelo en cascada.

C. No contempla los riesgos del proyecto.

D. No contempla el desarrollo de prototipos.

10. ¿Cuáles de los siguientes pueden ser *stakeholders* de un producto o proyecto?

A. Un experto en el dominio.

B. Un ministerio que establece regulaciones sobre productos y servicios.

C. Un usuario final.

D. Todas las respuestas anteriores son correctas.