Ingeniería del software

Índice

Qué es la ingeniería del software

Introducción

Según Fritz Bauer, “la ingeniería del software es el establecimiento y uso de principios de ingeniería robustos, orientados a obtener económicamente software que sea fiable y funcione eficientemente sobre máquinas reales.”

Es decir, la ingeniería del software es el proceso de desarrollo que seguimos para construir un sistema informático y posteriormente mantenerlo, ajustándonos siempre a diferentes factores: recursos, costo, duración, calidad, etc.

Piensa en cualquier otro proyecto o actividad que se te ocurra: cambiar una rueda, construir una casa o incluso subir una montaña.

En ellas, podemos encontrar factores comunes, como la necesidad de contar con productos específicos para llevarlos a cabo: el gato, planos, una botella de agua…

Pero hay algo que destaca principalmente, y es que debemos seguir unos pasos muy concretos, y hacerlos en orden para poder realizar el objetivo.

No podemos quitar la rueda de un coche si no lo hemos elevado primero con el gato, ni tampoco es habitual comenzar la casa por el tejado.

La botella de agua podemos olvidárnosla y tendremos que darnos la vuelta sin haber logrado nuestro objetivo.

Todo proyecto, y en particular un proyecto software, necesita de una planificación; es decir, debe definirse previamente paso a paso.

La Ingeniería del Software nos va a ayudar en todo este proceso de construcción, aportándonos los métodos, elementos, técnicas y procedimientos adecuados para poder concluir con éxito un proyecto software.

Esto también implica que todos los participantes, tanto usuarios y clientes como desarrolladores, obtengan un beneficio del proyecto y sus resultados.

Por lo tanto, la ingeniería del software cubre la necesidad de los ingenieros de software de contar con una disciplina que nos ayude y nos proporcione normas y herramientas sobre las que nos podamos basar para desarrollar nuestros proyectos.

¿Por qué la ingeniería del software como disciplina?

Cuando el software empezó a desarrollarse, allá por los años 50, se comienza con programas pequeños e intuitivos y esencialmente artesanales. El desarrollador no tenía ni metodologías ni herramientas ni nada que le ayudara en su proceso ni tampoco en la gestión o control del proyecto.

Esto, unido a que la demanda era cada vez mayor, el hardware cada vez más potente y los sistemas más complejos, tuvo como consecuencia una productividad muy baja y una calidad del sistema que se entregaba muy pobre.

Debido a esta situación, la OTAN decidió celebrar una conferencia en 1968 para poner sobre la mesa todos estos problemas, a los que denomina en su conjunto Crisis del software.

Para poner solución a esta Crisis del software propone precisamente una nueva disciplina a la que llama Ingeniería del software. Y así es como nace.

A partir de aquí se empiezan a:

- definir metodologías y procedimientos
- implementar técnicas
- a construir herramientas sobre las que fundamentar la Ingeniería del Software

Por supuesto, hoy en día sigue en evolución y en desarrollo.

La ingeniería del software tiene, por tanto, como objetivo proporcionar métodos, herramientas, procedimientos y técnicas con el fin de reducir el costo de los proyectos y mejorar la calidad del software.

En otras palabras, ayuda al ingeniero de software a desarrollar y mantener su sistema informático asegurando

- su calidad,
- su fiabilidad
- su productividad
- otras características.

De esta forma, el humano siempre dirigirá al ordenador y no al revés.

Al igual que las demás ingenierías, la ingeniería del software ayuda a construir elementos que imitan, aumentan, ayudan, facilitan o sustituyan capacidades del ser humano.

La diferencia es que las demás ingenierías imitan, aumentan, ayudan, facilitan o sustituyen capacidades físicas mientras que la ingeniería del software lo hace con capacidades psíquicas. Estas son, por ejemplo, la capacidad de memoria, la resolución de problemas, la rapidez decisiones o la velocidad de cálculo, entre otras muchas.

Además, según el estándar de ingeniería IEEE, “la ingeniería del software es la aproximación sistemática al desarrollo, operación y mantenimiento del software”.

La palabra **sistemática** es clave ya que implica la utilización de métodos y procedimientos

Conceptos importantes

Una vez vistos los objetivos y el propósito de la Ingeniería del Software, por qué existe y para qué nos sirve, vamos a abordar los conceptos principales para que, a partir de aquí, entendamos la terminología y puedas seguir el curso con facilidad.

Hay dos diferencias clave entre conceptos que debemos tener muy claras:

- PROYECTO: esfuerzo temporal que se lleva a cabo para crear un producto, servicio o resultado único.
- TRABAJO OPERATIVO: efectuar permanentemente actividades que generan un mismo producto o proveen un servicio repetitivo.

Como puedes ver, la definición de **proyecto** no depende de la complejidad o magnitud del mismo, sino de las características de **único** y **temporal**. A lo largo del curso nos vamos a centramos en este concepto.

Proceso

El **proceso** es un **conjunto de actividades para obtener un fin**. En nuestro caso, el fin será el producto software final resultante de nuestro proyecto. Pero ¡ojo!, que puede haber productos intermedios.

Así, cada fase también se puede considerar un proceso, puesto que es un conjunto de actividades para conseguir un fin que sería el producto resultado de esta fase. Por ejemplo, en la fase de análisis, todas las actividades que hay que realizar en esta etapa constituyen un proceso y el resultado de la fase que, en este caso, sería el documento de especificación de requisitos software, es el producto.

Figura 1: Cada una de las fases presentadas es un proceso

Por tanto, tenemos, por una parte, el **producto final**, que es el sistema software que resulta de la ejecución de todo el proyecto y los **productos intermedios** que son los resultados o salidas de cada una de las fases. Y la ingeniería del software actúa sobre todos ellos.

El objetivo de la Ingeniería del software es conseguir un producto software fiable, de alta calidad y bajo costo y conducir un proceso de desarrollo y mantenimiento software de manera eficiente y con éxito.

Evolución de un proyecto

Aquí ves la evolución de un ser humano desde que nace hasta que fallece. Si lo piensas, esta evolución se asemeja mucho a la de un proyecto, ¡hasta incluye chequeos y controles para prevenir o sanar enfermedades!

La discusión es larga, pero no me voy a meter acá.

Software

Definición de software

El software incluye los **productos que generamos durante nuestro proyecto informático**.

Piensa si evoluciona a lo largo de la vida del proyecto, si hay que controlar esa evolución, si tiene que ser mantenido, si lleva gestión de versiones, etc. **Si la respuesta es sí, entonces estamos hablando de software**.

En cuanto a datos, se refiere a estructura de los mismos

En cuanto a la documentación, se refiere al Manual de Usuario, Plan del Proyecto y Documento de diseño.

Peculiaridades del software

Ya hemos mencionado que la ingeniería del software tiene fases similares a las demás ingenierías,pero métodos y técnicas muy distintos.

Y… ¿cuál es el motivo? La respuesta está en la naturaleza del software.

Imagina que queremos construir una silla. Para su construcción, podríamos definir previamente las medidas, el color, el material, la forma, etc.

Puedo dibujar un esquema que será fiel a la realidad, puedo construirla, tocarla, probarla sentarme encima, intentando romperla, etc.

Pero fíjate en que nada de esto lo puedo hacer con un software. No veo ni toco lo que estoy desarrollando. Y, lo que es más complicado, tampoco lo ve ni lo toca el usuario final, por lo que es difícil saber, ya no solo si le va a gustar o no, sino si es lo que quería o no lo es.

Es más, ni siquiera al inicio del proyecto el cliente tiene claro lo que quiere y cómo lo quiere.

Por lo tanto, ¿qué hace que el software sea distinto?

Pues varias características:

- Es un elemento lógico y no físico, por lo que no se puede ver ni tocar.
- Es inmaterial e invisible.
- Es desarrollado, no fabricado. No tenemos una cadena de producción donde se van incorporando piezas y los operarios se turnan. Esto quiere decir que no hay "piezas de repuesto".
- Se deteriora, no se rompe. Se va degradando poco a poco.
- Y se construye a medida.
- El software está "vivo", es decir, siempre está cambiando mientras se está utilizando.
- Tampoco es fácil comprender ni predecir el comportamiento que va a tener.
- Además, es complejo. Los sistemas actuales están formados por miles de funciones con interfaces complejas entre ellas.
- Finalmente, el comprador no lo puede evaluar hasta que no haya sido desarrollado.
- Salvo en el caso de las herramientas, no puede ver ningún sistema funcionando que sea idéntico al que se va a desarrollar para él.

Es por estas características intrínsecas del software y diferentes a cualquier otro material, que la ingeniería del software define

- metodologías,
- procedimientos
- técnicas

específicos para el desarrollo y gestión del software.

Actividades del ingeniero del software

El proceso de desarrollo software se asemeja en sus fases al de construcción de una casa o de cualquier proyecto ingenieril.

Pero, además de estas actividades de desarrollo, el ingeniero de software debe realizar tareas de gestión, de control y de operación, que iremos viendo a lo largo del curso.

- En particular, las **tareas de gestión** engloban estimación y planificación, seguimiento de los proyectos, dirección técnica, gestión de recursos, administración de proyectosy coordinación del equipo de trabajo, entre otras.
- Las **tareas de control** aseguran la buena marcha del proyecto comprobando mediante métricas que las estimaciones, planificación y calidad se cumplen. También plantean cómo

realizar una buena gestión de la configuración del software o cómo asegurar, en la medida de lo posible, la calidad.

- Las **tareas de operación** englobamos la formación a usuarios, la entrega del producto, la puesta en marcha y las actividades de soporte.

Un sistema en operación es un sistema en uso, por lo que habrá que realizar todas las tareas necesarias para asegurar la correcta utilización del sistema software que hayamos implementado.

Algo importante: si un proyecto está perfectamente desarrollado desde el punto de vista técnico, los desarrolladores están contentos con su trabajo, pero no se usa, es un fracaso.

Si no se utiliza por el usuario, el proyecto es un fracaso.

Con todo lo visto hasta ahora podemos concluir que, en contra de lo que muchas personas piensan, realizar un proyecto software no es ni mucho menos programar.

Programar es simplemente una fase más dentro del proceso de desarrollo. Pero el comenzar

y terminar con éxito un proyecto software implica muchas más actividades, tareas y responsabilidades.

Aspectos del proyecto

Para que un proyecto software **finalice con éxito**, el director del mismo debe tener en cuenta, controlar y asegurar continuamente de manera equilibrada varias características o **requisitos del proyecto**. De la lista que aparece, arrastra a las casillas aquellas que consideres que son cruciales para la satisfactoria finalización del proyecto.

Los aspectos cruciales a tomar en cuenta para que un proyecto acabe con éxito son:

- Satisfacción del cliente
- Alcance
- Tiempo
- costo
- Riesgos
- Calidad

El proceso de desarrollo

Como ya has podido darte cuenta, es muy fácil relacionar la construcción de una casa con el desarrollo de un proyecto software.

Fíjate que también en la fase de mantenimiento podemos encontrar actividades de tipo correctivo (solucionar problemas), preventivo (prevenir problemas), adaptativo (adaptarse a nuevas situaciones del entorno) y perfectivo (realizar mejoras).

Ciclo de vida y metodología

Algunos proyectos, y especialmente los más complejos, como construir una casa, requieren de un proceso de desarrollo sistemático.

Nosotros, como ingenieros del software, ¿de qué manera aplicamos la ingeniería del softwareen nuestro trabajo?

Existe un concepto fundamental: la **metodología**.

"La metodología es el conjunto de métodos que se utilizan en una determinada actividad

con el fin de formalizarla y optimizarla."

Es decir, la metodología determina los pasos a seguir y cómo realizarlos para poder finalizar con éxito el proyecto. En nuestro caso, esta actividad será nuestro proyecto software.

Cuando aplicamos metodologías a la ingeniería del software, lo que optimizamos es tanto el proceso como el propio producto software.

Además, nos proporciona métodos que guían al ingeniero de software en la gestión, el desarrollo y el control del proyecto.

En definitiva, define qué hacer, cuándo y cómo hacerlo durante todo el desarrollo y mantenimiento de un proyecto.

Pero… ¿cómo definimos el esquema a seguir en este desarrollo y mantenimiento? Pues con lo que denominamos **"ciclo de vida"**.

"El ciclo de vida es el conjunto de fases por las que pasa el sistema que se está desarrollando, desde que la idea inicial nace hasta que el sistema software es retirado o reemplazado."

Por lo tanto, **un ciclo de vida determina el orden de las fases del proceso software** y también define las entradas y salidas de cada una de estas fases. En un principio, la salida de una fase será la entrada a la siguiente. Y, por último, establece los criterios de transición para pasar de una fase a otra.

Vamos a recapitular los tres conceptos clave: proceso, ciclo de vida y metodología.

- El **proceso** únicamente nos da un conjunto de actividades que tenemos que realizar para conseguir un fin, que sería nuestro producto software.
- El **ciclo de vida** va un poco más allá, y además de identificar estas actividades o fases, define el orden en que deben ejecutarse y los resultados intermedios y finales.
- Las **metodologías** nos dicen, además de lo anterior, el cómo llevar a cabo el proyecto. Para ello, nos proporciona un conjunto de métodos, herramientas y técnicas que facilitan la tarea del ingeniero de software y aumentan su productividad.

Definición y aplicación de metodologías a la Ingeniería del Software

Como hemos visto ya, las metodologías son el conjunto de métodos que utilizamos para formalizar y optimizar tanto el proceso como el producto software.

Las metodologías nos definen qué hacer, cómo hacerlo y cuándo hacerlo durante todo el desarrollo y mantenimiento de un proyecto software.

Teniendo esto en cuenta, las metodologías nos proporcionan un marco o estrategia global para enfrentarnos con el proyecto y esto implica tener varios aspectos en cuenta.

Uno de ellos son las fases por las que pasa nuestro proyecto, el orden en que se llevan a cabo y las tareas a realizar en cada fase.

Las metodologías también definen cuáles son los productos tanto intermedios, es decir entradas y salidas de cada fase, como nuestro producto final.

Además, nos proporciona procedimientos y herramientas para apoyarnos en la realización de cada tarea y, finalmente, identifica criterios de evaluación, tanto del proceso como del producto, para saber si hemos logrado nuestros objetivos.

Las metodologías nos sirven para no enfrentarnos a nuestro proyecto en blanco, sino con una estructura que seguir, en lugar de ir haciendo lo que nos va surgiendo en cada momento, situación que llevaría directamente al fracaso del proyecto software.

Por ello, uno de las funciones principales de las metodologías es definir el ciclo de vida más adecuado a las condiciones y características del proyecto, que será el esquema que vamos a ir siguiendo durante nuestro desarrollo.

Teniendo en cuenta cómo son estos ciclos de vida, las metodologías se dividen en varios tipos.

Los dos principales son metodologías pesadas

1. Tradicionales: Las metodologías tradicionales nos proporcionan fases bien definidas, requisitos bien identificados y entregas planificadas.
2. Ágiles: se basan en una continua interacción con el cliente, muchas entregas parciales y ciclos iterativos cortos.

Aunque haya épocas en las que algunas metodologías estén más de moda que otras, realmente la mejor metodología para un proyecto es la que mejor se adecúe a este, dependiendo de sus características y necesidades.

Aun así, es importante tener en cuenta que las metodologías habitualmente se adaptan y no se adoptan con el fin de que sean realmente eficaces para nuestro proyecto.

Metodologías tradicionales: principales ciclos de vida

Vamos a ver los ciclos de vida más habituales de las metodologías tradicionales:

- **cascada simple**: se presenta como una sucesión de etapas que producen productos intermedios.

  Para que el proyecto tenga éxito deben desarrollarse todas las fases y en el orden establecido, que es secuencial. Este sencillo ciclo de vida tiene una serie de limitaciones que hacen que sea muy improbable poder utilizarlo.

  Una de ellas es que no se permiten las iteraciones, por lo que los requisitos se tienen que congelar al principio del proyecto. Esto supone que hay que tener desde el inicio los requisitos totalmente claros y completos sin permitir ninguna modificación durante el desarrollo, lo que es bastante irreal.

- **cascada de refinamiento por pasos**: Para solventar este aspecto se propone el modelo de cascada de refinamiento por pasos. En este caso, los productos de las diferentes etapas se van mejorando y refinando de tal manera que desde cada fase puedas volver a cualquiera anterior.

Este modelo de ciclo de vida es muy utilizado y lo que hay que tener en cuenta es que debes llevar un control exhaustivo.

Si se permiten iteraciones y refinamientos continuamente el proyecto puede no acabar nunca. Es por eso que hay que ir negociando qué cambios se van a aceptar y cuáles no.

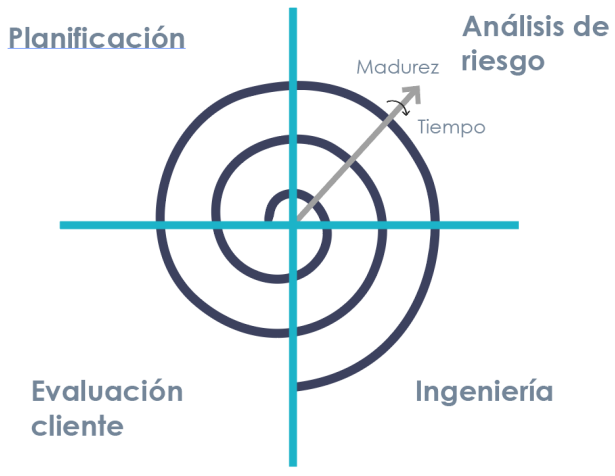
- **cascada iterativa:** es muy similar al de refinamiento por pasos pero, normalmente, se iteran todas las fases cada vez, es decir, en cada ciclo o iteración se revisa y mejora la calidad del producto.
- **cascada incremental:** Aquí se analiza primero el problema y se divide en subproblemas o subsistemas que se van desarrollando en cada iteración.

  A su vez, **al principio del proyecto** se define la división de subsistemas y requisitos que se implementarán en cada iteración.

  En este modelo, el producto software se desarrolla por partes que se van integrando a medida que se completan los subsistemas.

  De esta forma, en cada incremento hay que finalizar con un producto que sea operativo y, a su vez, en cada incremento se agrega más funcionalidad al sistema.
- **ciclo de vida incremental-iterativo:** combina el ciclo de vida incremental y el iterativo. El producto final se va desarrollando mediante incrementos y dentro de cada incremento, el ciclo de vida es iterativo. Esto tiene una gran potencia, pero hay que definir al inicio del proyecto bien qué se va a desarrollar en cada incremento, por una parte y, por otra, controlar las iteraciones dentro de cada incremento.
- **espiral:** Cada etapa se divide en cuatro cuadrantes. En el primero se definen en detalle las actividades y los objetivos de la fase. En el segundo se analizan los riesgos de la misma, intentando poner soluciones, incluyendo la creación de prototipos para minimizarlos. En el tercer cuadrante se lleva a cabo el desarrollo de la fase en sí y su validación. Finalmente, en el cuarto cuadrante, se planifica la siguiente fase.

  Dado que es una metodología que consume muchos recursos, se usa únicamente en proyectos donde se prevén riesgos importantes. De hecho, presenta un enfoque dirigido al riesgo. Tiene como ventajas que utiliza las fases de modelos tradicionales, se centra en la eliminación de errores y alternativas poco atractivas y que se orienta a detectar y prevenir el



riesgo evita muchas dificultades. Sin embargo, las fases y las entradas y salidas de las mismas quedan poco definidas.

Metodologías Ágiles

Las llamadas metodologías ágiles proporcionan una visión más dinámica que las metodologías tradicionales.

Su característica principal es que permiten cambios en los requisitos en cualquier fase del desarrollo; es más, estos cambios son bienvenidos.

Puesto que la máxima prioridad es la satisfacción del cliente, se le van entregando resultados evaluables de forma continua con los que se va midiendo el progreso del proyecto.

Es importante que tanto el cliente y usuario como los desarrolladores trabajen juntos durante el proyecto, por lo que el cliente debe tener disponibilidad para ello.

Finalmente, los equipos de las metodologías ágiles deben auto-organizarse entre ellos, se reúnen diariamente y tienen que reflexionar frecuentemente sobre cómo ser más eficaces y mejorar la calidad técnica.

En este sentido, el compromiso del equipo de proyecto debe ser fuerte. Dentro de las metodologías ágiles más usadas podemos encontrar XP (eXtreme Programming), KANBAN o SCRUM.

SCRUM

Comenzamos conociendo al equipo, que se divide en roles principales y roles secundarios.

Los roles secundarios son todas aquellas personas adicionales que participan de alguna manera en el proyecto: proveedores, vendedores, subcontratados, etc.

Entre los roles principales está

- **Product Owner**, que representa al cliente, por lo que conoce las prioridades del proyecto y prioriza las tareas a realizar.
- **Scrum Master**, que, ¡ojo!, no es un jefe de proyecto tradicional, ya que en metodologías ágiles no existe este rol. El Scrum Master se encarga de asegurar el seguimiento de la metodología guiando a los demás para que el equipo alcance el objetivo de cada sprint y organiza las reuniones pertinentes.
- Finalmente está **SCRUM Team** que es el resto del equipo encargado de implementar las funcionalidades asignadas por el Product Owner, integrar y entregar el producto, que no debe ser muy grande ya que es descentralizado y autodirigido. El tamaño ideal es entre cuatro y ocho personas.

Proceso de desarrollo.

El desarrollo del proyecto comienza creando el **Product Backlog**, que es un conjunto de requisitos y características de alto nivel priorizados que definen todo el trabajo a realizar.

Habiendo definido todos los requisitos y características, se decide cuáles de ellos se llevarán a cabo en el primer sprint.

Pero ¿Qué es un sprint?

Los sprints son las iteraciones de desarrollo. Cada una finaliza con un incremento del software potencialmente utilizable, que el cliente evaluará.

Su duración recomendada es entre dos y cuatro semanas. Por lo tanto, estamos hablando de un desarrollo iterativo incremental.

En el **Sprint Planning** se decide qué elementos del Product Backlog formarán parte de ese sprint en particular.

Durante esta reunión, el Product Owner, que recordemos que es el representante del cliente identifica los elementos del Product Backlog que quiere ver completados en este sprint.

A continuación, el equipo determina la cantidad de ese trabajo que puede comprometerse a completar durante el sprint.

Con esto se crea el **Sprint Backlog**, donde figuran los requisitos y las tareas que se van a realizar durante el sprint y que durante el mismo NO se podrán cambiar.

Durante el Sprint, el equipo tiene diariamente una reunión muy corta, llamada Daily Meeting, con una duración de unos 10 minutos y que siempre se realiza en el mismo sitio y a la misma hora con el fin de optimizar el tiempo.

En esta reunión cada miembro del equipo responde a tres preguntas:

1. si ha finalizado el trabajo que se había auto-asignado.
2. qué problemas ha tenido,
3. qué trabajo va a realizar hasta el siguiente Daily Meeting.

Todo esto se refleja en un tablero de tareas que va, lógicamente, evolucionando según avanza el proyecto.

Una vez finalizado el sprint, se llevan a cabo las reuniones de revisión y retrospectiva, donde el equipo de desarrollo presenta al cliente el entregable que ha desarrollado.

A partir de ahí, se reflexiona sobre la calidad y completitud del mismo y comienza una nueva iteración con una nueva Sprint Planning donde se seleccionarán los requisitos del Product Backlog para el siguiente sprint.

Y así se continua hasta la finalización del proyecto.

Product Backlog

Los requisitos de alto nivel en el Product Backlog se formulan en forma de Historia de Usuarios y se formulan de la siguiente forma:

- Como [Rol], quiero hacer [descripción de la funcionalidad], con el objetivo de [Razón o Resultado].

- Donde [Rol] es el cargo de la persona que hace la petición o, por defecto, el cliente, [descripción de la funcionalidad] es la descripción breve y concisa de la funcionalidad o requisito que el Product Owner quiere que tenga el sistema y [Razón o Resultado] es el beneficio que se pretende obtener de la implementación de dicha funcionalidad.
- Todas las historias de usuario deben llevar un identificador de tal manera que sean únicas.
- La prioridad se establece en función del valor para el cliente y el esfuerzo estimado.
- En la columna de Iteración o Sprint se escribe el número del Sprint en el que se va a desarrollar esta funcionalidad o Historia de Usuario dependiendo de su prioridad.

Identificador (ID) de la Historia	Enunciado de la Historia	Valor para el cliente	Esfuerzo en p-d	Iteración (*sprint*)	Prioridad
XXX-XXXX	Como [Rol], quiero hacer [descripción de la funcionalidad], con el objetivo de [Razón o Resultado]				
XXX-XXXX	Como [Rol], quiero hacer [descripción de la funcionalidad], con el objetivo de [Razón o Resultado]				

La gestión de un proyecto

Tareas de gestión

¿Cómo influye la gestión de un proyecto software en el éxito o fracaso del mismo?

La gestión del proyecto es una actividad continua, pues se realiza durante todo su ciclo de vida. Comienza con la idea, necesidad o demanda y finaliza cuando el proyecto se retira.

La gestión y dirección de proyectos tiene una gran influencia en el éxito o fracaso de los mismos.

Generalizando, cuando un proyecto está bien gestionado pero tiene problemas técnicos, tiene más posibilidades de recuperarse que un proyecto mal gestionado. De hecho, se podría decir que un proyecto mal gestionado está casi, casi, abocado al fracaso.

¿Y, qué implica esa gestión deficiente, que lleva casi directamente al fracaso de un proyecto?

Pues aspectos tales como:

- una incorrecta estimación y planificación de los recursos o la duración del proyecto
- una pobre gestión del riesgo
- un insuficiente seguimiento durante el desarrollo,
- el hecho de no aplicar procedimientos de monitorización y control
- una escasa gestión de la calidad.

El papel del jefe o director del proyecto es, por tanto, fundamental ya que será el responsable final de su éxito o de su fracaso.

Según el estándar IEEE, la gestión de proyectos es el proceso de planificar, organizar, proveer de personal, monitorizar, controlar y liderar un proyecto software. Si tuviésemos que resumir el objetivo principal en dos palabras, diría: resolver problemas.

Por otra parte, la gestión eficaz de un proyecto software se centra en las cuatro P's: personal, producto, proceso y proyecto.

Para gestionar eficazmente tendremos que realizar tareas de gestión sobre cada una de estas cuatro P's.

Tareas principales de la gestión de proyectos

Las tareas principales englobadas en la gestión y dirección de proyectos son:

- **Estimación**: La estimación es la predicción cuantitativa de aspectos del proyecto tales como duración, esfuerzo y costos requeridos para realizar todas las actividades y constituir todos los productos asociados al proyecto.
- **Planificación**: La planificación es la organización de recursos humanos y tareas para finalizar con éxito el proyecto.
- **Coordinación del equipo de trabajo**: La coordinación y motivación del equipo de trabajo es esencial para tener un equipo efectivo y eficiente. El entorno de trabajo influye muchísimo en la productividad, con lo que es importante ocuparse de crear un buen ambiente en el equipo.
- **Negociación:** La negociación, especialmente con el cliente, es crucial porque va a implicar el poder acotar y acordar qué vamos hacer en nuestro proyecto.
- **Seguimiento y control**: Necesitamos hacer un exhaustivo seguimiento y control de los avances, tanto del proceso como de los productos para asegurar que el proyecto va por buen camino y no se desvía de la planificación realizada. Pero, ¡ojo!, no solo debemos controlar la calidad de los productos que estamos generando, sino también que no nos pasemos de costos, esfuerzo y tiempo.
- **Gestión:** La gestión propiamente dicha engloba todas las actividades administrativas, financieras, de planificación y de definición que influyen en el buen término del proyecto software. No siempre es una tarea que hace personalmente el propio jefe de proyecto, pero él es el responsable final, en cualquier caso.
- **Dirección técnica:** La dirección técnica también puede delegarla el jefe de proyecto, especialmente si el proyecto es grande. El director técnico se encarga de colaborar y supervisar las actividades técnicas en las distintas fases del desarrollo, resolviendo los problemas que surjan.

Coordinación y motivación del equipo de trabajo

A la hora de crear el equipo de trabajo, lo primero que hay que hacer es definir una estructura y una organización del mismo.

También asignar roles y responsabilidades a todos los participantes y definir los canales de comunicación.

Es importante que cada persona sepa:

- qué tiene que hacer
- para cuando lo tiene que hacer,
- de qué herramientas dispone
- a quien tiene quién recurrir si tiene algún problema
- de quien depende su trabajo
- cómo se van a evaluar sus resultados.

Una vez que el equipo de trabajo está creado y en marcha, es importante darles formación, información y motivación.

Estas son las tres claves para que un equipo funcione bien.

A cada miembro hay que hacerle sentirse importante, darle responsabilidad en lo que haga. La gente prefiere trabajar sabiendo que es responsable de lo que hace a estar desarrollando algo de lo que luego la responsabilidad la va a tener otro, le pone menos interés.

A las personas también nos gusta estar informados, por tanto, dales información del proyecto, hacer que se sientan parte de él. Por ejemplo, mantener reuniones periódicas donde se exponga la situación del proyecto, los problemas existentes y se propongan soluciones.

En cuanto a la motivación, es uno de los aspectos más importantes para que un equipo de trabajo funcione. Para ello, hay que conocer las expectativas y las necesidades de cada uno y motivarles según estas sean estas.

Por ejemplo, si a alguien le gusta mucho el trabajo técnico, recompensarle con algún curso de formación de tipo técnico. Si alguien tiene problemas de horario, adáptaselo en la medida de lo posible, etc.

Lo importante es crear un ambiente de equipo agradable. Nada que ver entre trabajar en un entorno cómodo y amigable y un entorno donde no se esté a gusto. Si se consigue un buen ambiente de trabajo, es mucho más fácil que las capacidades individuales y grupales de los miembros del equipo se puedan desarrollar al máximo.

Otro aspecto que suele funcionar bien es fijar metas alcanzables y utilizar un sistema de incentivos para premiar con comportamientos positivos. Pero no que exista un único premio para un único ganador, porque puede perjudicar la cohesión del equipo.

Suele funcionar mejor dar un premio a cualquiera que supere un objetivo planteado.

Para coordinar al equipo de trabajo y sacar lo mejor de cada persona de manera individual y como equipo, uno de los principales puntos es ser un líder y no un jefe.

Esto conlleva saber delegar, estimular y guiar, más que ordenar y dirigir.

Algunos trucos para ser un buen líder son:

- hacer preguntas en vez de dar órdenes,
- elogiar los progresos de los demás,
- hablar de tus propios errores,
- no poner en ridículo a nadie en público, o hacer que sus errores parezcan fáciles de corregir
- hacer sugerencias
- y, sobre todo, ser un ejemplo.

Como dijo Robert Dils, “liderar es crear un mundo al que las personas deseen pertenecer”. En nuestro caso ese mundo es nuestro proyecto.

Estimación

La estimación es la predicción cuantitativa de aspectos del proyecto, tales como duración, esfuerzo y costo.

Es decir, se trata de asignar una medida o un número a cada uno de estos elementos con la que podamos acabar el proyecto satisfactoriamente para todos.

Cada una de estas medidas tienen sus unidades correspondientes:

- La duración la mediremos en unidades de tiempo, ya sean días, semanas, meses o años.
- El costo, en la moneda que corresponda dólares, euros, etc.
- Y el esfuerzo en la unidad de esfuerzo que consideremos, ya sea persona-día, persona-semana, persona-mes o persona-año.

La unidad de esfuerzo, por ejemplo, persona-semana, por ejemplo, hace referencia al trabajo resultante que, de media, puede sacar adelante una persona a tiempo completo durante una semana.

Por eso es una unidad de esfuerzo o productividad. Cuidado porque aquí personas y tiempo no son intercambiables. No es lo mismo cinco personas trabajando un mes que una persona trabajando cinco meses.

Las tareas tienen dependencias por lo que estos números no son intercambiables. A partir del esfuerzo estimado, obtenemos el costo.

¿Cómo?

En todas las organizaciones existe un coeficiente interno que es el dinero que tiene que ingresar una persona que trabaja en un proyecto en una semana, mes o la unidad en la que estemos midiendo, para poder abordar los gastos de la empresa y que el balance final salga positivo.

A esta partida de Recursos Humanos, que habitualmente es la más grande, habría que sumarle viajes, hardware o software que haya que comprar para el proyecto y cualquier otro gasto necesario y, con ello, configurar el costo total del proyecto.

Por lo tanto, resumiendo, la estimación predice el tiempo, esfuerzo y costo que necesito para finalizar con éxito mi proyecto.

Técnicas de descompocisión:

Para la estimación del esfuerzo es muy útil descomponer el problema en subproblemas o subsistemas siguiendo el enfoque conocido como “divide y vencerás”. Consiste en dividir el sistema completo en distintos módulos y, a su vez, descomponer estos en tareas o funciones lo más elementales posibles.

La estimación de cada una de estas tareas unitarias es mucho más fácil, al ser más pequeñas y sencillas. Simplemente quedaría ir sumando hasta tener el total de todos los módulos, que será la estimación del proyecto completo.

Una manera útil de hacer esta descomposición es siguiendo el modelo matricial, es decir, con una tabla. En las filas ponemos los distintos módulos y en las columnas las distintas fases de cada uno de estos módulos; por ejemplo, análisis, diseño, calificación y pruebas.

Así, en cada celda tenemos que estimar cuántos recursos necesitaríamos para realizar la tarea que sea (análisis, diseño, codificación o pruebas) en cada uno de los módulos y sumando filas y columnas obtendremos la estimación total.

Vemos un ejemplo en la siguiente tabla. Supongamos un sistema software que tendrá los siguiente módulos o subsistemas: comunicaciones, razonamiento, interfaz de usuario y base de datos. Estimamos los recursos que necesitaremos, por ejemplo, en persona-semana para cada una de las tareas de estos módulos y lo ponemos en la celda correspondiente:

	Análisis	Diseño	Codificación	Pruebas	Total (p-s)
Comunicaciones	3	2	1,5	2	**8,5**
Razonamiento	7	6	5	5	**23**
Interfaz usuario	5	4	4	4	**17**
Base de datos	2	2	1,5	3	**8,5**
Total	**17**	**14**	**11**	**14**	**57 p-s**
Tarifa (euros)	**1.800**	**1.600**	**1.400**	**1.400**	
Coste total	**30.600**	**22.400**	**15.400**	**19.600**	**88.000 €**

Sumando las filas podemos saber la estimación para cada módulo y sumando las columnas conoceremos la estimación para cada tarea. Por ello, este modelo resulta simple y completo a la vez.

Si, además ponemos la tarifa o ratio de cada personal y multiplicamos, podemos saber el costo de cada perfil, y, lo que es más importante, como el costo total del proyecto.

Esta ratio, evidentemente, no es el salario, sino cuánto hay que ingresar por perfil para que el ingreso de los proyectos a clientes cubra todos los gastos de la empresa

Técnicas empíricas:

Por su parte, las técnicas empíricas usan fórmulas derivadas empíricamente para predecir el esfuerzo y otros atributos del software terminado en función de líneas de código, puntos de función, etc.

Los modelos más utilizados son: COCOMO II, modelo de puntos de función, modelo de puntos objeto, modelo de casos de uso, etc.

Planificación

En la planificación de un proyecto distribuimos entre las distintas tareas los recursos que ya hemos estimado: duración, costo y recursos humanos.

Previamente debemos haber desglosado el proyecto en subproyectos y estos en tareas. De esta forma, podemos definir las relaciones y las dependencias entre tareas y distribuir entre ellas los recursos, de manera que el conjunto complete el producto final.

Es decir, a cada tarea le asignamos:

- una duración
- unas dependencias
- unos recursos humanos.

Además de esta planificación de recursos y tiempo, hay que planificar hitos.

Los hitos son momentos del ciclo de vida donde pararemos a evaluar lo que hemos hecho y comprobar si esto coincide con nuestra planificación o no. Evaluaremos tanto los resultados obtenidos como el costo, el tiempo y los recursos consumidos hasta ese momento.

Si todo está acorde con lo planificado y la calidad se considera adecuada, continuaremos.

Si ésto no se cumple, lo primero es diagnosticar qué es lo que ha pasado, qué ha ocurrido para que nos desviemos del plan previsto. Lo segundo será aplicar los planes de contingencia necesarios para reconducir el proyecto. Por último, tendremos que actualizar el plan y proponer uno viable y acorde con la situación actual del proyecto.

Si el plan de proyecto no se va actualizando, no servirá para nada haberlo hecho.

Por lo tanto, la planificación de un proyecto consiste en la:

- organización temporal de las tareas
- y en la asignación y distribución de recursos a estas tareas.

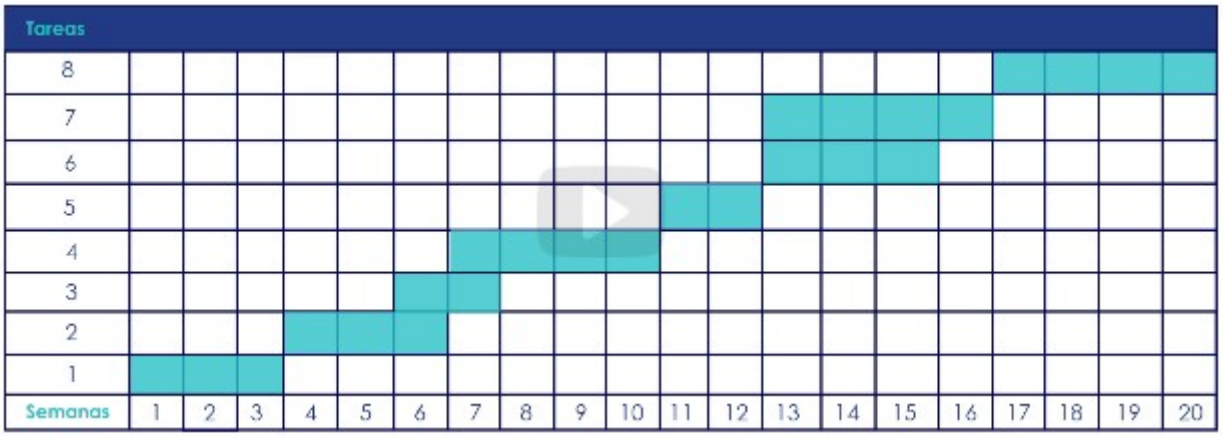
El definir hitos a lo largo del ciclo de vida es muy importante ya que nos da la posibilidad de hacer el seguimiento y control del proyecto durante todo el ciclo de vida.

Técnicas de ayuda: GANTT y PERT

El diagrama de GANTT es un diagrama de barras donde el eje de las equis (x) representa el tiempo y el eje de las íes (y), las tareas.

Por lo tanto, cada tarea se dibuja como una barra horizontal que muestra la duración de la misma.

En este ejemplo, la duración total del proyecto es de 20 semanas, y el número de tareas es 8:

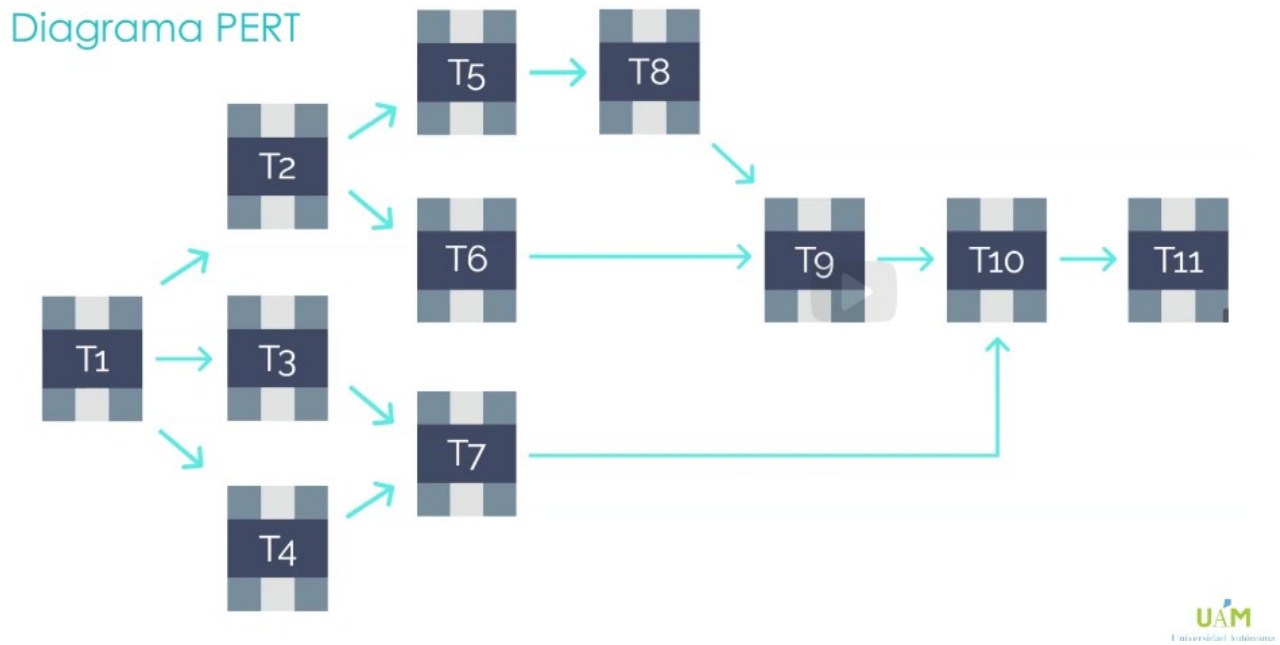


La tarea 1 dura tres semanas; en particular, de la semana 1 a la 3. La tarea 2 comienza cuando la primera ha acabado y dura también 3 semanas. La tercera dura 2 semanas y debe comenzar una semana antes de finalizar la tarea 2, etc.

Fíjate en que, a pesar de ser simple, el diagrama GANTT tiene una gran potencia porque, de un vistazo, nos da una visión global del proyecto. Además, nos permite saber en cualquier momento por dónde deberíamos ir, para compararlo con la ejecución real de nuestro proyecto.

Por su parte, el diagrama PERT está formado por una red de tareas, con sus relaciones, dependencias y duraciones.

En este ejemplo encontramos 11 tareas organizadas en estas cajas que vemos:



Para realizar el PERT comenzamos haciendo una pasada de izquierda a derecha para el cálculo del comienzo y el fin más pronto posible de cada tarea. Así, suponiendo que estamos trabajando con días, la casilla central de arriba indica el número de días que dura cada tarea.

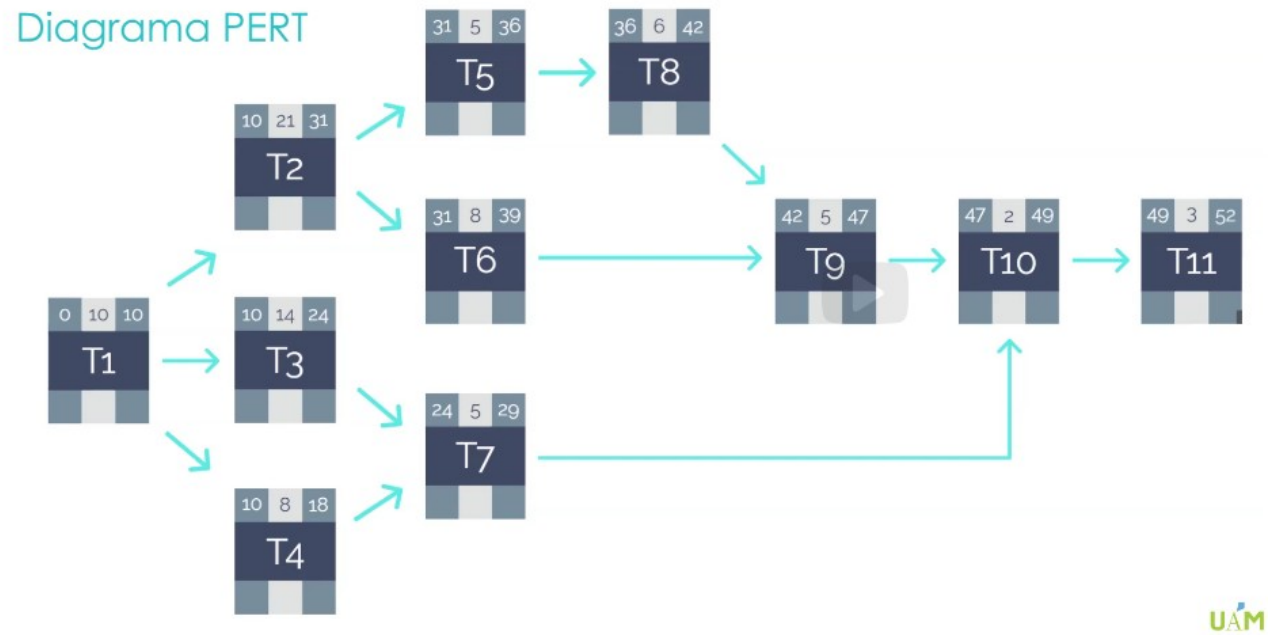
En la casilla de arriba a la izquierda pondremos el día que antes puede comenzar esa tarea y en la de arriba a la derecha, el día que antes puede finalizar.

Así, empezamos con la tarea 1 que tiene una duración de 10 días por lo qué contando que empieza el día 0, lo más pronto puede terminar es el día 10, puesto que dura 10 días.

Las tareas 2, 3 y 4 no pueden comenzar hasta que finalice la 1, por tanto, lo más pronto que pueden comenzar es el día 10 y, sumando la duración de cada una, tenemos lo más pronto que pueda finalizar cada tarea. Y así seguiremos sucesivamente.

Un caso especial es cuando una tarea depende de varias, es decir, no puede comenzar hasta que dos o más hayan finalizado.

Por ejemplo, la tarea 7 no puede comenzar hasta que finaliza tanto la 3 como la 4. Como la tres finaliza, como muy pronto, el día 24 y la 4 el día 18, la tarea 7 no podrá comenzar hasta que finalice la última de las dos, en este caso la 3, por lo tanto no puede empezar hasta el día 24



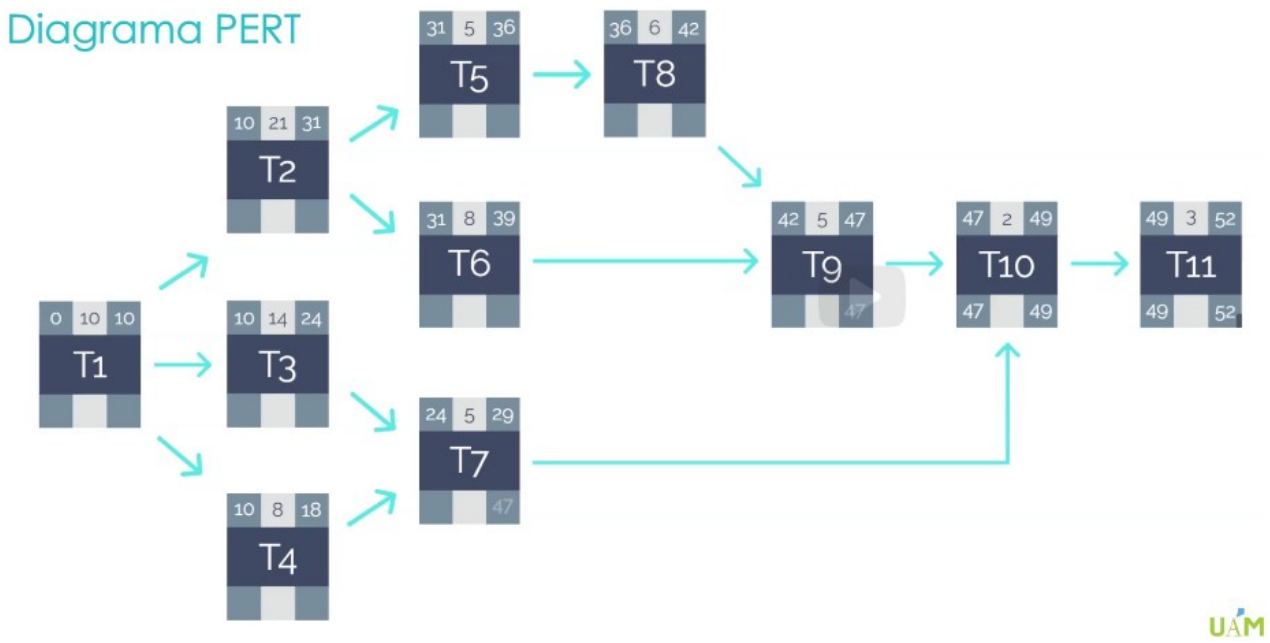
.

De esta forma, llegamos hasta la última tarea, la 11 y obtenemos en la casilla de la derecha la duración del proyecto, en este caso 52 días.

A continuación, hacemos una pasada de derecha a izquierda para calcular el inicio y el final más tardío posible de cada tarea sin que se vea afectada la duración total.

Empezamos por la tarea 11 y copiamos el 52 en la casilla de abajo a la derecha, porque es la duración total, y no queremos que se desplace porque se retrasaría el proyecto. Como la tarea 11 dura 3 días, lo más tarde que puede comenzar sería en el día 49, que lo escribimos en la casilla de abajo a la izquierda.

Trasladamos este día a su predecesora, la tarea 10, como el día que más tarde puede finalizar.

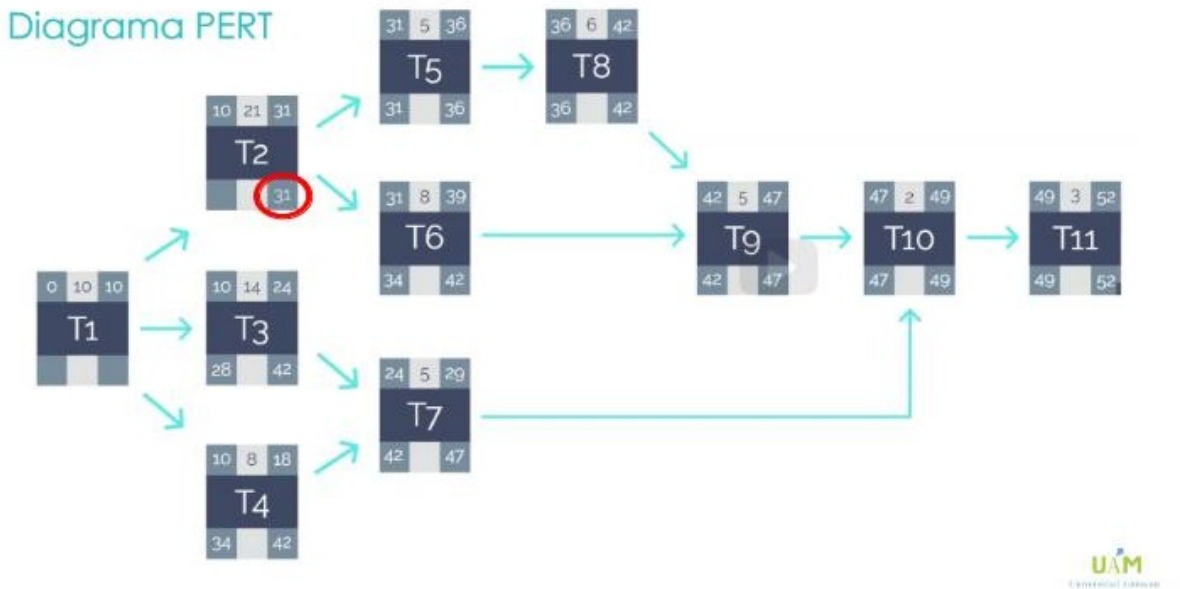


Así continuamos hacia atrás.

Cuando tenemos dos tareas con una única predecesora, elegiremos el día que más tarde puede empezar como finalización de la anterior.

Por ejemplo, la tarea 2, como muy tarde, tiene que acabar a tiempo para que tanto la 5 como a la 6 puedan empezar sin retrasos.

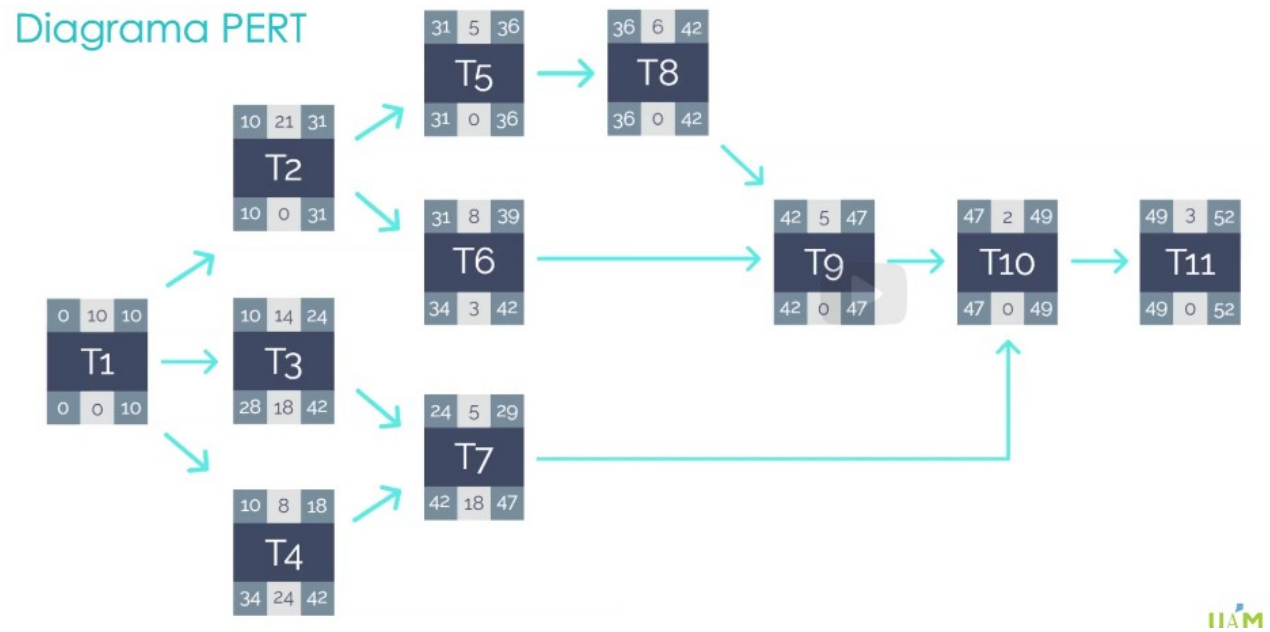
En este caso, como la tarea 5 como muy tarde puedo empezarla el día 31 y la seis, como muy tarde, el día 34, entonces tomamos el día 31 como última fecha posible de la tarea 2.



Así continuamos hasta la tarea inicial.

Una vez que tenemos en cada tarea la duración y las fechas que más pronto y más tarde pueden comenzar y terminar, sin que la duración del proyecto se vea afectada, hallaremos la holgura.

La holgura nos indica la ventana o margen que tiene cada tarea de poder retrasare sin que afecte al proyecto.



Se calcula como la resta entre lo más tarde que puede finalizar una tarea (casilla izquierda inferior) y lo más pronto que puede comenzar (casilla derecha superior). Y se coloca en la casilla central de abajo.

Por ejemplo, en la tarea 6, que tengo una holgura de 3 días, significa que, si se retrasa 3 días respecto a la planificación prevista, no ocurre nada porque la duración del proyecto se mantiene. Aunque, obviamente, hay que estar pendiente.

Aquellas tareas que tienen holgura cero son las que no tienen ningún posible margen de movimiento. Si estas tareas se retrasan lo más mínimo, la duración total del proyecto también se retrasa.

Los caminos de tareas que tiene holgura cero, se llama caminos críticos y es en las tareas de estos caminos donde tendremos que llevar un especial control para asegurar que se cumplen estrictamente los plazos.

Así, en este ejemplo, el camino crítico de este proyecto es el formado por las tareas T1-T2-T5-T8-T9-T10-T11.

Para realizar el GANTT de este segundo ejemplo, se eligen las fechas que más pronto puede comenzar y terminar cada tarea. De esta forma, si se retrasa, tendremos la holgura de margen sin afectar a la duración total. Si utilizáramos las fechas más tardías, no tendríamos ningún margen.

Estas técnicas de ayuda son muy útiles para la planificación de un proyecto. Normalmente se utilizan herramientas software para realizar los diagramas que acabamos de ver.

Documentación:

La salida de la etapa de planificación es el **plan de proyecto**. El plan de proyecto es un documento que principalmente determina el alcance del proyecto, define los recursos y el costo y detalla el plan temporal.

Este sería un posible ejemplo del contenido del plan del proyecto. Fíjate en que recogemos todos los aspectos que hemos ido mencionando: recursos, costo, planificación, etc.

1. Alcance
    - Objetivos del proyecto
    - Desglose en subsistemas
    - Funcionalidades principales
    - Otras características
    - Escenario de desarrollo
    - Restricciones técnicas y de gestión
2. Estimación de recursos
    - Recursos humanos
    - Recursos hardware y software
    - Ventanas de disponibilidad
3. Estimación de costo
4. Plan temporal
    - Red de tareas PERT/CPM
    - Diagrama de Gantt
5. Equipo de desarrollo: roles y responsabilidades
6. Requisitos de disponibilidad del cliente
    - Información/Datos
    - Reuniones
    - Otros

Errores típicos durante la planificación

- La estimación como la planificación de un proyecto la hace una persona con experiencia. Es una tarea que necesita de alguien que haya trabajado en distintos proyectos y experimentado los problemas que suelen surgir.

- Error: ser demasiado optimista en la estimación. Nadie es capaz de trabajar y producir todo al día durante todos los días laborables de la semana. Hay vacaciones, la gente se pone enferma, se tiene que excusar por asuntos personales, se utiliza tiempo para hablar por teléfono, tomarse un café, etc. Así que seamos realistas y planifiquemos pensando en todos estos tiempos adicionales.
- Error: No realizar un seguimiento del proyecto y no ir actualizando el plan. Pues, un plan que no se usa y no se va actualizando, no es de utilidad.
- Debemos identificar los riesgos probables del proyecto durante esta etapa de planificación.
- Definir y repartir las responsabilidades adecuadamente entre los miembros del equipo.
- Definir los criterios de evaluación de nuestro producto final.

Con todo esto tendremos una importante parte del éxito del proyecto conseguida.

Notas:

- Para realizar la planificación de un Proyecto distribuimos el número de recursos humanos hallado en la estimación entre el costo total.
- El objetivo de realizar un diagrama PERT/CPM es obtener el o los caminos críticos.

Análisis

Propósito e importancia de hacer un análisis

El análisis de requisitos es la primera fase del ciclo de vida, en la que empezamos a desarrollar nuestro proyecto software.

En esta fase tenemos que llevar a cabo dos aspectos muy importantes:

- analizar el problema
- y definir el producto.

En esta etapa participan tanto los clientes y usuarios como los ingenieros de software.

Los clientes y usuarios plantean el problema actual o la necesidad que tienen y el resultado que esperan obtener bajo las condiciones que necesitan.

Los ingenieros de software, por su parte, preguntan, analizan, asimilan y presentan la solución adecuada.

Ambos roles deben reunirse tantas veces como haga falta para conseguir una descripción completa y detallada del sistema software final.

Pero, en realidad, partimos de dos fuentes:

- estas entrevistas o reuniones en las que trataremos de comprender las necesidades de los usuarios
- un análisis del contexto para aprender lo más posible sobre el problema o la necesidad.

Tenemos que empaparnos bien del entorno y las condiciones en las que trabaja el futuro usuario, así como especificar claramente las tareas y las restricciones del sistema que vamos a desarrollar.

Por lo tanto, estos son los principios que se deben cumplir al realizar esta fase de análisis:

- Se debe comprender el problema y su entorno. No podemos definir aquello que no comprendemos.
- Además, las funcionalidades y los requisitos se determinan siguiendo una aproximación descendente. Es decir, primero se analiza el problema de forma global y después vamos detallando cada vez con más profundidad.
- También, la especificación de requisitos debe poder ser ampliable, pues es habitual que los requisitos software iniciales tengan que sufrir alguna modificación controlada a lo largo del ciclo de vida.
- No nos podemos dejar influir por el sistema operativo, el lenguaje de programación o por la técnica que vamos a utilizar a la hora de definir qué es lo que va a hacer el sistema final.

Todo esto es igualmente válido para las metodologías ágiles y los ciclos de vida incrementales, aunque no se definen todos los requisitos al inicio del proyecto.

Ten en cuenta que, en cualquier caso, por muy cortos que sean los ciclos de vida, siempre hay que seguir las mismas fases de desarrollo:

- análisis (qué)
- diseño (cómo)
- codificación (ejecución)
- pruebas.

Finalmente, concluiremos esta fase de análisis con un documento llamado especificación de requisitos software donde recogemos todas las funcionalidades y requisitos que tiene que cumplir nuestro sistema.

Esta fase es de extrema importancia porque es aquí donde se define qué va a hacer nuestro sistema exactamente y bajo qué condiciones y restricciones. Y cuando esté desarrollado, lo validaremos contra este documento de requisitos para comprobar si los cumple o no.

Es por esto que, si no realizamos el análisis de requisitos de manera correcta, cualquier error se propagará a lo largo del ciclo de vida, resultando en un producto software final que no es lo que el cliente quiere.

¿Cómo empiezo el desarrollo de mi proyecto?

Lo más difícil de comenzar algo nuevo es dar el primer paso.

Veamos cómo me enfrento a un proyecto, cómo comienzo a analizar el problema y a definir el producto.

Hagámoslo con un ejemplo.

Un restaurante desea mejorar su servicio de carta y ofrecer menús para comidas contratadas con antelación.

Para ello, te encarga una aplicación informática que le proporcione sugerencias de menús, así como una lista adicional de alimentos que necesitan y que no tienen en el almacén.

El menú sugerido dependerá del tipo de comida (desayuno, almuerzo o cena), el número de comensales, el gasto aproximado que el cliente desea desembolsar, el ambiente requerido (como formal, de trabajo o de fiesta, etc.) y se permitirá introducir alguna condición adicional por ejemplo, como vegetariano o bajo en calorías, etc.

Además, los usuarios tienen muy poca formación informática, por lo que debe ser sencillo de utilizar y tampoco va a existir un ordenador dedicado solo para este sistema.

Además, quieren conectar una impresora.

Bien, para empezar a analizarlo, pensamos en el sistema a desarrollar como una caja negra y estudiamos las salidas que deberá tener.

Las salidas serán, como mínimo, los menús con sus precios correspondientes y la lista de alimentos que es necesario comprar en cada caso.

Después, pasamos identificar qué entradas necesitamos para generar estas salidas. Pues necesitaremos una interfaz interactiva donde, por pantalla, se pueda introducir el tipo de comida, el número de comensales, el gasto aproximado, el ambiente y las condiciones adicionales.

También tendremos que tener una base de datos con platos ya pensados que se puedan combinar, para configurar así el menú que más se ajuste a las peticiones del cliente.

Por ejemplo, los platos pueden estar catalogados por estaciones y cada plato tiene que figurar con los alimentos de los que está compuesto.

A su vez, cada alimento debe contener atributos tales como precio, calorías, tipo (carne pescado verdura, etc.) condiciones adicionales, etc.

Pero, para poder sacar una lista de alimentos que hay que comprar, también necesitaremos tener una base de datos actualizada con el Stock de alimentos que tiene el restaurante cada día para poder saber de qué disponemos y de qué no.

Por tanto, estas serán las tres entradas que necesitaremos:

- la interfaz interactiva
- una base de datos con los menus (y sus cualidades, como ser precio, calorías y alimentos que utiliza)
- una base de datos con el stock.

Una vez tengamos claras las entradas y salidas de nuestro sistema, del que también sabemos el objetivo general, lo dividiremos en subsistemas que representen las funcionalidades principales.

Por ejemplo, en los subsistemas de: lectura de datos, gestión del stock de alimentos, búsqueda de la solución óptima, generación de la lista de alimentos y presentación de resultados.

A continuación, para cada uno de estos subsistemas identificaremos todos los requisitos funcionales. Y, para el sistema global, los requisitos no funcionales.

Vamos a poner algún ejemplo de estos últimos.

Un requisito de documentación, por ejemplo, puede incluir la realización y entrega del manual de usuario, pero también proporcionar ayuda por voz,

En cuanto a recursos, podemos pensar que van a manejar el sistema cuatro usuarios diferentes en dos terminales distintos.

Respecto a la fiabilidad, podemos requerir que, cuando no encuentre alguna sugerencia, el sistema explique por qué y en este caso y en los que se produzca algún error, se permita el paso al manual.

De esta forma iremos identificando y formulando todos y cada uno de los requisitos: los funcionales por subsistemas y los no funcionales para todo el sistema.

Al finalizar obtendremos la definición completa de nuestro producto.

Actividades de análisis

En la fase de análisis encontramos cinco tareas principales para llevar a cabo una buena extracción de requisitos.

- En primer lugar, identificaremos los distintos usuarios que utilizarán la aplicación, su rol, su jerarquía y las tareas que realizará cada uno.
- A continuación, extraeremos los requisitos, que obtendremos de los usuarios y clientes y del estudio del entorno.
  Una vez tengamos la lista de requisitos, debemos razonar sobre la posibilidad de combinar aquellos requisitos relacionados, dividir los requisitos complejos, determinar su viabilidad o establecer prioridades. Con esto concluimos con otra lista de requisitos ya analizada y razonada.

- De esta forma ya estaremos preparados para representar los requisitos, lo cual es habitual hacer mediante diagramas.
    1. Si estamos haciendo un análisis estructurado, haremos principalmente un diagrama de flujo de datos.
    2. Si estamos siguiendo una aproximación orientada a objetos, utilizaremos diagramas UML.

  Representar los requisitos en forma de diagramas nos permite obtener más información sobre la aplicación.
  Y es muy habitual durante este proceso el darse cuenta de que faltan algunos requisitos o que alguno es incorrecto, por lo que podremos actualizar nuestra lista.
- Finalmente, debemos validar todos los requisitos, tanto individualmente como en su conjunto. Por una parte, el cliente debe revisarlos y asegurar que está de acuerdo con ellos. Por otra, nosotros debemos comprobar que cumplen atributos como los siguientes:
    1. Que sean completos; es decir que todo lo que el software tiene que hacer esté recogido en este conjunto de requisitos.
    2. Que no sean ambiguos, con lo que cada requisito debe tener una sola interpretación.
    3. Que sean trazables, de tal forma que posteriormente cada acción de diseño se corresponda con algún requisito.
    4. Que sean correctos, es decir, que reflejen las necesidades reales.
    5. Que sean consistentes, es decir, que no haya requisitos contradictorios.
- Y siempre, antes de dar por finalizado el conjunto de requisitos, debemos enseñárselos al cliente para que los valide y ratifique que, efectivamente, esas son las funcionalidades que quiere que haga la aplicación, su aplicación.
  En esta tarea de validación debemos ser muy estrictos y pensar que los requisitos no están escritos para que los entendamos nosotros, sino para que el diseñador los pueda representar sin ninguna duda al respecto.

Identificación de requisitos

Ya hemos visto los pasos a seguir en el análisis de un proyecto software. La **extracción e identificación de requisitos** es un proceso a través del cual los clientes, compradores o usuarios de un sistema software exponen y articulan sus requisitos, mientras que los ingenieros de software los comprenden y formulan. Esta tarea se realiza mediante reuniones, entrevistas, análisis de las tareas, lectura de documentos o manuales, etc.
Vamos a centrarnos en el elemento estrella: los tipos de requisitos que podemos identificar.

Aunque ya hemos introducido el término requisito, vamos a estudiarlo un poco más.

Los requisitos son las condiciones o capacidades necesarias que debe tener un sistema informático para que pueda resolver un problema o alcanzar un objetivo propuesto.

La principal división que hacemos de requisitos es distinguir entre requisitos funcionales y requisitos no funcionales.

- requisitos funcionales: representan acciones fundamentales que tienen que tener lugar en la ejecución del software. Son acciones elementales necesarias para el correcto comportamiento de nuestro sistema final.

Como su nombre indica, representan una funcionalidad básica y, por tanto, se deben formular como una acción, con un verbo.
Por ejemplo, “El usuario podrá dar de alta un elemento”. Esto es un requisito funcional.

- requisitos no funcionales son también necesarios para el correcto comportamiento de nuestro sistema software. Sin embargo, no representan funcionalidades ni acciones, sino características o cualidades generales que se esperan del software para conseguir su propósito.
  Normalmente hacen referencia a atributos como la eficiencia, seguridad o usabilidad del sistema. Algunos son orientados al usuario y otros al desarrollador.

El estándar de la IEEE (Std 830 – 1993) establece 13 tipos de requisitos no funcionales que deberían tenerse en cuenta en toda especificación de software. Además de los anteriores otros pueden ser los requisitos de interfaz de usuario, de usabilidad, de seguridad o los de documentación. Por otra parte, también podemos diferenciar entre requisitos de usuario y requisitos software.

- Los requisitos de usuario son declaraciones en lenguaje natural de las funciones o acciones que los distintos usuarios pueden realizar con la aplicación y bajo qué restricciones. Estos requisitos conforman el Documento de Requisitos de Usuario o DRU.
- Los requisitos software, derivados de los anteriores, especifican de una manera completa, consistente y detallada qué debe hacer y cómo debe comportarse el software para cumplir con los objetivos de la aplicación. Estos requisitos sirven de base a los desarrolladores para diseñar el sistema y se recogen en el documento Especificación de requisitos Software o ERS.
  Los requisitos software deben responder a la pregunta: ¿qué características necesita cumplir el sistema software para permitir alcanzar los requisitos expuestos en el DRU?

Por ejemplo, el requisito de usuario: “Un estudiante podrá matricularse de todas las asignaturas pendientes de un curso” Para pasarlo a requisito software, debemos definir varios requisitos en los que el sistema software consulte los datos del expediente, realice las validaciones necesarias, muestre el importe dependiendo del número de matrícula que sea, etc.

Es decir, todas aquellas acciones que el sistema software debe realizar para permitir que el estudiante se matricule de las asignaturas pendientes.
La salida de la fase de análisis será el documento de especificación de requisitos software y, en el caso de las metodologías ágiles, el conjunto de historias de usuario.

Modelos de desarrollo de productos

Independientemente de los modelos de ciclo de vida, podemos utilizar modelos de desarrollo de productos software, en particular **maquetas** y **prototipos**.

Ambos se utilizan cuando los requisitos no están claros, pero en el caso de la **maqueta** los que no están claros son los **requisitos de usuario**. Lo que hacemos es implementar la interfaz de usuario, lo que sería la fachada del sistema, pero sin funcionalidad detrás, para que el usuario la vea y juegue con ella reportando el feedback. Así sacaremos requisitos de usuario que no habían salido hasta el

momento ya que, normalmente, los usuarios cuando ven algo hecho se les ocurren más funcionalidades o cambios.

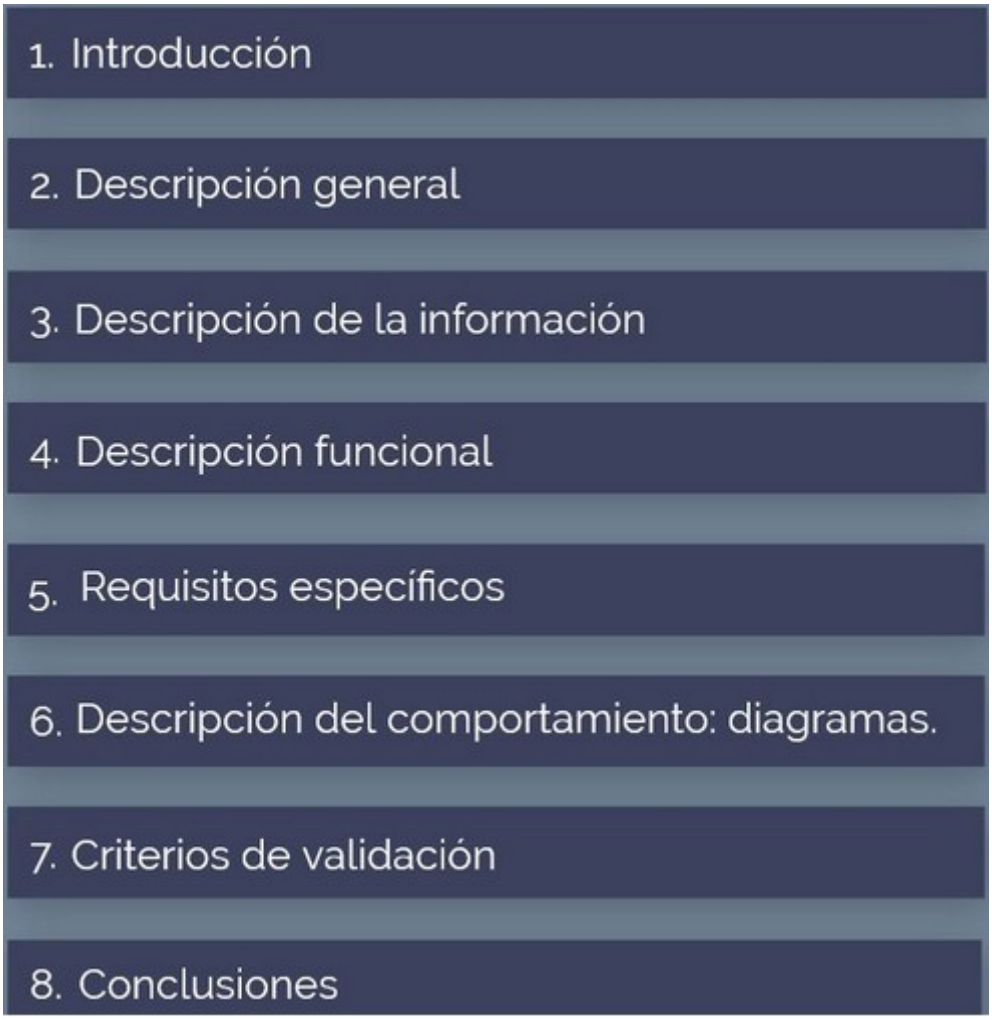
El uso de maquetas tiene una gran ventaja y es que la posible no aceptación del sistema final, que es uno de los grandes problemas de los proyectos software, queda prácticamente solventada. El usuario ve cómo quedará el sistema final, lo va comentando y se va negociando hasta finalizar con una interfaz que contenta tanto al desarrollador como al cliente.

El **prototipo** se realiza cuando el desarrollador **no tiene claro los requisitos software**, en particular, cuando no tiene muy clara la solución informática más adecuada o no se conoce la complejidad total del desarrollo.

En este caso, se construye un prototipo de una parte del sistema con total funcionalidad, normalmente la parte más compleja. Una vez verificada la viabilidad de este prototipo se continúa iterativamente implementando funcionalidades sobre el mismo.

Documentación: especificación de requisitos software

Los siguientes puntos identifican los principales contenidos de un ejemplo de una **especificación de requisitos software**, documento generado como salida de la fase de análisis.



Diseño

Definición y niveles del diseño

El diseño de software parte de las especificaciones software y las transforma en una solución software informática, cambiando el foco del qué al cómo.

Es decir, pasamos del "qué hay que hacer", que hemos definido en la especificación de requisitos, al "cómo hay que hacerlo", que lo determinaremos en el diseño.

Es muy importante dejar claro desde el principio que no hay una única solución de diseño válido. Cada persona o equipo de trabajo realizará un diseño diferente y muchos de ellos pueden ser válidos.

Cuando tengamos varias soluciones posibles, siempre que todas sean correctas, elegiremos la más simple, lo que nos va a facilitar la vida en las próximas etapas.

En la fase de diseño definiremos:

- la arquitectura del sistema
- sus componentes o módulos
- las interfaces internas y las externas hacia otros sistemas o dispositivos

- Diseño de datos.

Todo ello para satisfacer los requisitos especificados en análisis. Por eso era tan importante la etapa previa de análisis. Si los requisitos están mal, el diseño no será correcto.

Tenemos dos niveles de diseño:

- diseño de arquitectura o de alto nivel:es el proceso de, una vez determinado el criterio de descomposición que vamos a seguir, definir todos los componentes de mi sistema, las interfaces entre ellos y determinar las estructuras de datos

  Nota: un diagrama de GANTT es parte del diseño de arquitectura.

- diseño detallado o de bajo nivel: describe de forma pormenorizada la lógica y la estructura de cada uno de estos componentes o módulos, las estructuras de datos que utilizan y los procedimientos de acceso a estas estructuras físicas de datos.

Realmente estamos partiendo del diseño de arquitectura para meternos dentro de cada uno de los módulos y diseñarlos más en detalle.

Volviendo a la casa que nos planteábamos construir al inicio de este curso, el diseño de arquitectura lo formarían los planos de la casa. Por su parte, el diseño detallado sería el plano específico de una habitación donde figuran todos los puntos de luz, enchufes, cableado, etc.

Finalmente, al concluir el diseño es el momento de definir la estrategia de pruebas que vamos a seguir más adelante en nuestro proyecto. En especial, las pruebas de integración, ya que son las que se realizan contra el diseño, como veremos en el tema de pruebas.

Principios básicos y métricas

En el diseño encontramos varios principios básicos que deben cumplirse en nuestras tareas:

1. Refinamiento, que consiste en seguir una estrategia descendente a la hora de diseñar.
2. Modularidad, puesto que es necesario dividir el software en niveles hasta llegar a unidades pequeñas pero con entidad propia.
3. Abstracción, es decir, el manejo de conceptos generales y no de instancias particulares.
4. Ocultación de la información, es decir, los detalles internos de cada módulo no se muestran a los demás.

De esta forma nos centramos en minimizar la complejidad. Al ir descomponiendo el problema en piezas cada vez más pequeñas, reducimos la complejidad. Pero ¡cuidado! Cada una de estas piezas debe contener una única función y ser lo más independiente posible en relación a las demás.

Justo en este punto entran las métricas de diseño.

Un buen diseño se mide a través de dos métricas:

- Acoplamiento: es una medida de la interconexión entre los módulos de un programa.

En este sentido, cuanto más bajo sea el acoplamiento, mejor, ya que la dependencia entre módulos es menor y minimizamos el riesgo de propagación de errores, además de simplificar la comprensión.

- Cohesión: es una medida de la relación de los elementos de un módulo. Por ello, querremos conseguir un alto grado de cohesión, de tal forma que los elementos de un mismo módulo estén muy relacionados entre sí. Esto supondrá un menor esfuerzo en las etapas de programación y pruebas y una mayor calidad del sistema.

Por tanto, podemos resumir que un módulo con un alto grado de cohesión hace idealmente una sola cosa.

En resumen, dividiremos nuestro sistema en módulos. Estos serán tan independientes como sea posible para conseguir un mínimo acoplamiento. Y cada módulo llevará a cabo una sola función para conseguir la máxima cohesión.

Representación: Diagramas

Dependiendo de si vamos a desarrollar el producto con de manera estructurada u orientada a objetos, se utilizan unos u otros diagramas tanto en el análisis como en el diseño.

Estructurado

En el análisis estructurado es habitual utilizar diagramas de flujo de datos y diccionarios de datos. Por su parte, en el diseño estructurado se trabajan los diagramas de estructura de cuadros.

Orientado a objetos

En el análisis y diseño orientado a objetos priman los diagramas UML, pero también se trabajan:

- Para el análisis, diagramas de casos de uso, casos de uso detallados u otros diagramas conceptuales y del sistema.
- Para el diseño, diagramas de clases y objetos, de componentes, de comunicación, de secuencia o de estados y actividad

Documentación

Un ejemplo de documento de diseño puede ser el que se muestra a continuación:

1. Introducción

- Propósito del documento
- Entorno hardware y software
- Principales funciones del software
- Bases de datos externas
- Restricciones y limitaciones
- Referencias

2. Descripción del diseño
    - Diagramas
    - Descripción de datos
    - Descripción de interfaces
    - Descripción de comunicaciones

3. Descripción de los módulos (para cada módulo)
    - Descripción
    - Descripción de la interfaz
    - Módulos relacionados
    - Organización de los datos

4. Descripción de archivos externos y datos globales
    - Descripción
    - Métodos de acceso
5. Especificaciones de programas
6. Referencias cruzadas con los requisitos
7. Plan de pruebas.
    - Estrategia de integración
    - Estrategia de pruebas
    - Consideraciones especiales
8. Conclusiones

Diferencias y similitudes entre análisis y diseño estructurado y orientado a objetos

En general, los principios de análisis y diseño que hemos visto son comunes tanto para técnicas estructuradas como para orientadas a objetos.

Sin embargo,hay elementos que difieren.

1. En una estrategia estructurada, la frontera entre análisis y diseño es más clara. Sin embargo, la frontera en una estrategia orientada a objetos es más difusa.

   Por ejemplo, comenzaremos un diagrama de clases en la fase de análisis y continuaremos refinándolo en la de diseño.

2. Y lo que también difiere son los diagramas utilizados en cada caso. En una estrategia estructurada tendremos, principalmente diagramas de flujo de datos, diagramas de flujo de

control y diagramas de transición de estados. Y en diseño, diagramas de estructura de cuadros.

En una estrategia orientada a objetos disponemos de diagramas de casos de uso, diagramas de despliegue, diagramas de objetos, de componentes, de clases, diagramas de secuencia, diagramas de comunicación, diagramas de estados, de actividad, etc.

Por último, es importante ser consistente a lo largo del proyecto. Por ejemplo, si realizo un análisis y un diseño orientado a objetos, después tendré que utilizar un lenguaje de programación orientado a objetos y nunca estructurado.

Codificación

Objetivo y salidas de la codificación

Pasemos ahora a la fase de codificación, donde tiene lugar la programación en sí.

Igual que dijimos que el objetivo del diseño era transformar la definición del problema en la solución software, en este caso el objetivo de la codificación es traducir las especificaciones de diseño en un lenguaje de programación.

Por ello, ¡cuidado! si hemos hecho mal el análisis o el diseño, arrastraremos los errores al código.

No es el objetivo de este curso enseñar a programar, con lo que pasaremos esta fase muy por encima. Pero sí es importante comentar, en cualquier caso, a la hora de codificar hay que seguir una guía de estilo, en especial si estamos trabajando en equipo. Además, el código tiene que estar bien estructurado y ser fácil de entender.

Como salidas de esta fase de codificación tenemos:

- los programas en sí
- el manual técnico:es un documento interno, es decir, para los desarrolladores, recogerálas funciones del código con el detalle de lo que hacen.
- el manual de usuario.es donde se describe cómo manejar el sistema y lo entregamos al usuario junto con el software ejecutable.

También se puede hacer una guía de instalación o cualquier otra documentación que hayamos acordado con el cliente.

Durante la programación es bastante habitual tener que modificar algo del diseño realizado anteriormente.

En tales casos, será necesario actualizar el documento correspondiente para que continúe siendo coherente con el código.

Pruebas

¿Por qué y para qué hacer pruebas?

Un error software existe cuando el software no hace lo que el usuario espera que haga.

Estos resultados que el usuario espera del sistema, deben estar previamente acordados en la especificación de requisitos.

Muy habitualmente se definen las pruebas software diciendo que probar es:

- demostrar que no hay errores del programa
- mostrar que el programa funciona correctamente.

Pero ambas cosas son incorrectas. Porque siempre hay errores, siempre. Y si no los detectamos nosotros, los encontrará el usuario.

Así que es mejor para todos que seamos nosotros, los desarrolladores nos encontremos la mayor cantidad posible de errores que haya, antes de entregar vuestro software.

Sin embargo, realizar pruebas de un código de forma completamente exhaustiva es imposible.

Para que te hagas una idea, probar exhaustivamente unos pocos miles de líneas de código llevaría millones de años, lo que obviamente es inviable.

Por tanto, nuestro desafío es diseñar casos de prueba y seguir una estrategia de pruebas que nos permita identificar el mayor número de errores con el menor número de casos posibles.

Por lo tanto, un buen caso de prueba es aquel que tiene una alta probabilidad de detectar un error no descubierto todavía.

Con esto quedan contestadas dos preguntas clave:

1. ¿Para qué hay que hacer pruebas? Para encontrar los errores que siempre existen en el código realizado.
2. ¿Por qué hay que hacer pruebas? Porque debemos encontrarlos nosotros, como ingenieros de software, el mayor número posible de errores antes de entregar el sistema informático al cliente.

Verificación y validación

- **Verificación** responde a la pregunta: ¿se ha construido el sistema correctamente? Es decir, comprueba el **funcionamiento correcto** del software desde un punto de vista técnico.
- Por su parte, **validación** contesta a la pregunta: ¿se ha construido el sistema correcto? Así, comprueba si los **resultados de usuario se cumplen**.

Un software puede estar técnicamente muy bien construido, pero los resultados obtenidos no ser los previstos, con lo que la verificación se cumpliría, pero no así la validación.

Técnicas de pruebas de código

Hay varios principios básicos que debemos respetar a la hora de realizar pruebas de código.

Un principio esencial es que el resultado esperado del programa es una parte integrante y necesaria de un caso de prueba.

Esto lo que nos quiere decir es que un caso de prueba está siempre compuesto por dos partes:

- las entradas que vamos a probar
- el resultado esperado.

Te asombrarías de la cantidad de veces que se hace una prueba, se ve que el resultado es más o menos razonable y no se compara con el esperado, con lo que no podemos sabersi la prueba ha resultado correcta o no.

Por lo tanto, es importante revisar cuidadosamente el resultado de cada prueba.

Otro principio relevante es que los casos de prueba deben ser escritos tanto para condiciones de entrada válidas como para condiciones inválidas.

En el caso de que haya errores, debe informar al usuario de cuál ha sido el problema.

Por otra parte, un programador no debe ser el único que pruebe su propio programa.

Esto no quiere decir que no tengamos que probarlo, por supuesto que sí, sino que tiene que haber más personas que lo prueben.

Uno mismo tiende a no encontrar errores en su propio código. Al fin y al cabo, a nadie le gusta decirse, aunque sea implícitamente, que lo que ha hecho está mal.

Así que deben probar nuestro código otros miembros del equipo de desarrollo e, idealmente, alguien con un perfil similar al usuario final. Aunque esto último hay veces que no es fácil de conseguir.

Y, finalmente, es necesario documentar los casos de prueba:

- el diseño de los casos que vamos a probar
- el resultado que ha producido cada prueba y, en el caso de ser negativo, el procedimiento seguido hasta que ha resultado correcto. No dejemos nada sin documentar.

Diseño de casos de prueba

Aunque estemos hablando de pruebas de código, no debemos olvidar que debemos hacer pruebas a lo largo de todo el ciclo de vida para asegurar la buena marcha del proyecto.

Después del análisis de requisitos tenemos que comprobar, por ejemplo, que todos los requisitos están identificados, que no hay requisitos contradictorios ni inconsistentes ni redundantes.

Después del diseño tendremos que asegurar que éste cubre todos los requisitos descritos en la especificación y que cumple las métricas de acoplamiento y cohesión.

En codificación podemos hacer inspecciones de código que consisten en revisarlo únicamente de manera visual, es decir, leyéndolo.

Es sorprendente la cantidad de errores que se detectan solo con inspecciones de código.

Y también tenemos dos técnicas de pruebas de código en las que éste debe ser ejecutado:

- las pruebas de caja blanca: comprueban la lógica interna del programa.

Se centran en el comportamiento y la estructura interna del código. Para ello hay técnicas que nos permiten asegurar la ejecución de, al menos una vez, todas las sentencias, recorrer todos los caminos independientes y comprobar todas las decisiones lógicas y todos los bucles.

- las pruebas de caja negra: Aquí nos fijaremos en los datos de entrada y los datos de salida.

  Consideraremos el software como una caja negra, sin tener en cuenta los detalles internos.

  Introduciremos los datos de entrada que hayamos seleccionado y comprobaremos que generan las salidas especificadas en los requisitos.

  Las técnicas que se utilizan son

  - la partición en clases de equivalencia, principalmente,
  - y el análisis de valores límite.

  Con estas dos técnicas lo que conseguimos es diseñar un conjunto de casos de prueba que tenga la más alta probabilidad de detectar el mayor número posible de errores.

Usaremos principalmente técnicas de caja negra, pues son mucho más fáciles de realizar. Y para módulos u objetos complejos crearemos casos adicionales que examinen la lógica del programa con técnicas de caja blanca.

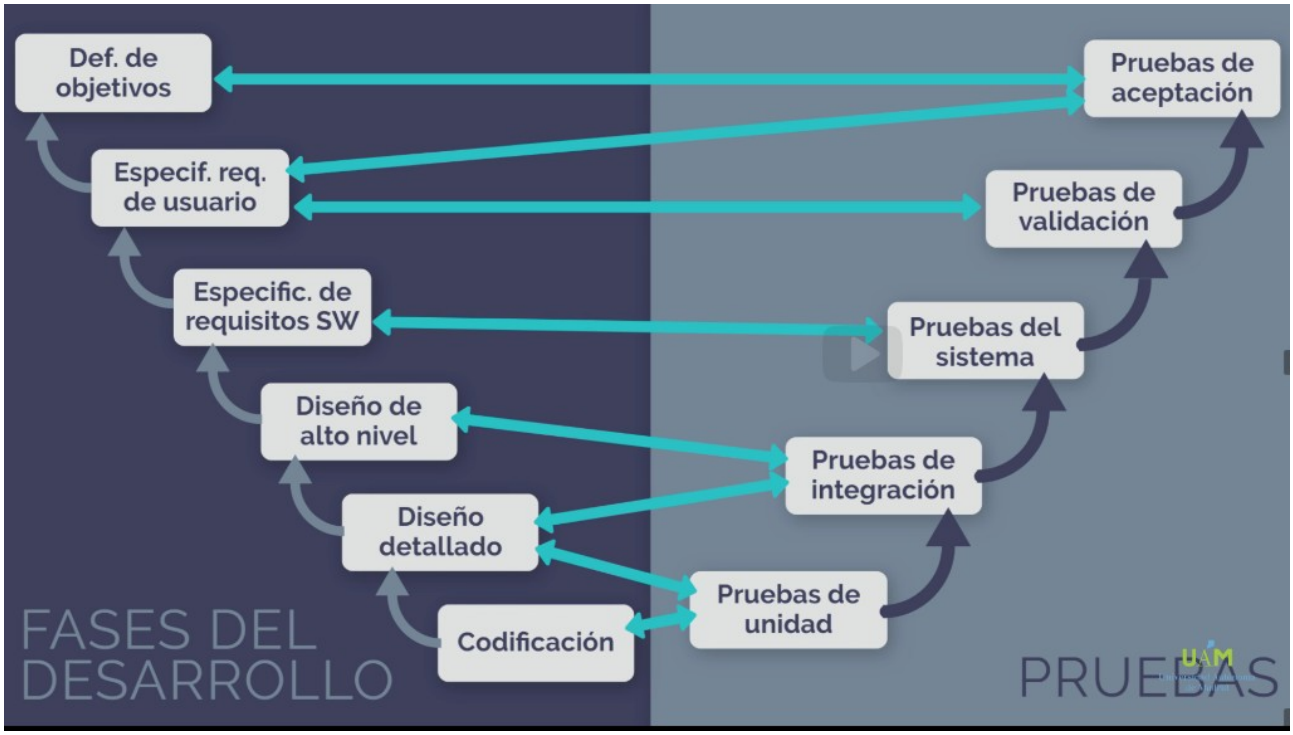
Recordemos que los casos de pruebas se forman por las entradas seleccionadas y la salida esperada.

Estrategia de pruebas

En un sistema informático, la estrategia de pruebas a seguir siempre será desde dentro hacia fuera, es decir, empezando con los módulos más pequeños y finalizando con el sistema completo.

Por eso las primeras pruebas serán las pruebas unitarias donde se prueba cada módulo o función por separado. Hay que decir que en muchas ocasiones estas pruebas se solapan con la codificación.

A continuación, realizaremos las pruebas de integración, donde vamos agrupando los módulos ya probados para comprobar las interfaces entre ellos. Lo que hacemos es integrar los distintos módulos uno a uno al grupo de módulos ya probado. No debemos integrar varios módulos a la vez porque si da error no sabemos de dónde procede.



Después, llevaremos a cabo:

- las pruebas de validación, donde comprobamos si las salidas están en concordancia con los requisitos de usuario especificados.
- las pruebas del sistema en las que integraremos nuestro sistema informático en el entorno hardware y software donde funcionará.

  Ningún sistema software está aislado. Se conecta con otros sistemas informáticos, bases de datos, dispositivos, etc.
- las pruebas de aceptación donde el cliente comprueba que el producto final se ajusta a los requisitos de usuario, es decir, lo que ellos pidieron.

Si estas pruebas son satisfactorias, el cliente dará el proyecto por aceptado.

En las pruebas de sistemas orientados a objetos se procede de forma similar, comenzando con lo más pequeño para finalizar con el sistema completo. En este caso:

- probaremos en primer lugar los métodos de las clases.
- A continuación, las clases individuales.
- Después la agrupación de objetos.
- Y, finalmente, el sistema entero.

Mantenimiento

Definición y motivación

Una vez tenemos nuestro software utilizándose por los usuarios, comenzamos un nuevo proceso, el de mantenimiento.

Este comprende todas las actividades que se realicen sobre el software una vez que esté ya en operación.

Actividades tales como corregir errores, mejorar su rendimiento u otros atributos, añadir funcionalidades, adaptarlo a un cambio en su entorno, etc.
El mantenimiento tiene una serie de problemas que el desarrollo no tiene y que dificultan su realización y aumentan su costo.

Por ejemplo, la dificultad en seguir la evolución del software a través de las versiones o de los cambios, la complicación en comprender un programa ajeno o el hecho de que muchas veces la documentación existente es bastante mala.
Todo esto hace que el trabajo de mantenimiento resulte poco atractivo, ya que no resulta fácil enfrentarse a un software que o bien no está hecho por ti o lo hiciste hace tiempo.

Y, ¡cuidado! El mantenimiento en duración es mucho más largo que el desarrollo, pero los recursos que se utilizan son mucho menores y más difíciles de estimar.
Esto implica que, si no llevamos controlado y planificado nuestro proceso de mantenimiento, se nos puede acabar yendo el proyecto de costo.
Muchas veces damos poca importancia al mantenimiento.

Sin embargo, también los costos intangibles derivados de una inadecuada estrategia de mantenimiento nos pueden perjudicar mucho.
Por ejemplo, la disminución de la calidad global del software o de la satisfacción del cliente, o incluso la pérdida de oportunidad.

Por ello, cuanto más nos hayamos preocupado por la mantenibilidad durante el proceso de desarrollo, más fácil será hacer el mantenimiento ahora.
Entendemos por **mantenibilidad** la facilidad de mantenimiento, es decir, la facilidad de corregir, adaptar o mejorar el software, y es uno de los atributos de la calidad del software.
Por lo tanto, nos tenemos que preocupar de ella durante todo el ciclo de vida del proyecto.
Eso implica principalmente ir realizando las distintas fases del ciclo de vida siguiendo los principios de ingeniería del software y revisando los resultados al final de cada etapa.

Por ejemplo, en análisis hay que asegurarse de que el conjunto de requisitos es completo, considerar requisitos de futuras mejoras o de extinción de la capacidad, etc.
En diseño hay que cumplir los principios básicos: modularidad, abstracción, ocultamiento de datos.
Y, cumplir las métricas de bajo acoplamiento y alta cohesión, de tal forma que tengamos un diseño claro y fiable.
En la codificación, debemos hacer y seguir una guía de estilo que nos proporcione un código legible y estructurado, así como documentar el código.

Y, en general, usar metodologías y estándares y generar una buena documentación que nos va a seguir a servir de mucha ayuda durante el mantenimiento.

En resumen, debemos hacer de la mantenibilidad uno de los objetivos del proceso de desarrollo.

Tipos de mantenimiento:

- **Correctivo:** Corrige errores
- **Adaptativo:** Acomoda el software a un nuevo entorno o a cambios en el mismo
- **Preventivo:** Previene errores
- **Estructural:** Modifica la arquitectura interna (reingeniería)
- **Perfectivo:** Mejora y añade requisitos o amplía los ya implementados

Estrategias de mantenimiento

Tenemos tres estrategias de mantenimiento con las que abordar este proceso:

1. **Estructurado:** se van acumulando las peticiones de cambio aceptadas y se realizan todas a la vez cada cierto tiempo.
   Por ejemplo, mi empresa ha decidido que el mantenimiento de un determinado sistema software se llevará a cabo por dos personas los meses de julio y diciembre de cada año. Según llegan las peticiones de mantenimiento, se evalúan y analizan. Las aprobadas se ponen en una cola y se llevarán a cabo en el siguiente periodo que corresponda, julio o diciembre en este ejemplo.
2. **No estructurado:** es bajo demanda. Según llegan las peticiones de mantenimiento se van resolviendo e implementando.
3. **Combinado:** es una mezcla de ambas. Lo primero que se hace es evaluar la petición de cambio. Si es viable se analiza su severidad, ¿cómo de grave es para el sistema informático y su entorno?
   Si no es muy severa se encola y se realizará en las fechas de nuestro mantenimiento planificado.
   Si es muy grave, se atenderá inmediatamente resolviendo la incidencia.
   En este caso, normalmente, las peticiones que se tratan bajo demanda, en el momento, son las de tipo correctivo y cuando el error es grave.
   Los demás tipos de mantenimiento, adaptación o mejora, habitualmente pueden esperar a cuando esté planificada su realización.

No hay una estrategia mejor que otra. Tenemos que seguir aquella que hayamos acordado con el cliente y que sea más adecuada para nuestro proyecto.
Es cierto que, normalmente, la más eficiente suele ser la combinada, ya que permite atender de forma inmediata las peticiones prioritarias del cliente.
Pero a la vez, puedo planificar los recursos humanos que se van a encargar del mantenimiento y cuándo lo van a hacer, de tal forma que sea posible optimizar el personal que trabaja en los distintos proyectos de mi empresa.
En cualquier caso, sea cual sea la estrategia de mantenimiento seguida, es muy importante ir actualizando la documentación a medida que se realizan cambios, controlar las versiones y asegurar la calidad del producto que estamos manteniendo.

Documentos relativos al mantenimiento
En el proceso de mantenimiento tenemos que realizar varios documentos.

- Antes de empezar cualquier actividad, debemos firmar con el cliente el **contrato de mantenimiento** donde se recogen todas las condiciones bajo las que se realizará este servicio.
- Justo al inicio deberemos llevar a cabo el **plan de mantenimiento** donde establecemos qué vamos hacer, quién, cuándo y cómo.
  Al igual que el plan de desarrollo debe irse actualizando a lo largo del proyecto.
- Por otra parte, debemos crear un **formulario de petición de mantenimiento**, de tal forma que el cliente solicite su petición siempre a través de este formulario y del medio que hayamos acordado con él.
- Finalmente, el **informe de cambios** recoge los detalles del cambio ya realizado, incluyendo objetivos, fecha, esfuerzo, persona encargada del cambio, etc.
- Y, muy importante, no nos olvidemos tampoco de documentar los programas que cambiemos para que queden registradas las modificaciones que vayamos realizando.

Formulario de petición de mantenimiento:
Mantenimiento correctivo

- Identificador
- Fecha
- Nombre de la persona que reporta el error
- Error encontrado, condiciones de entorno (circunstancias) y versión
- Consecuencias
- Soporte (datos de entrada, listados, etc.)

Mantenimiento adaptativo y perfectivo

- Identificador
- Fecha de la petición
- Petición
- Motivo de la petición
- Quién realiza la petición

Gestión de configuración y aseguramiento de calidad

Gestión de configuraciones

Introducción

La gestión de configuraciones del software se trata de una disciplina cuya misión es identificar, controlar y organizar la evolución del sistema software.

Dado que el software es algo vivo y en continua evolución, tanto en desarrollo como en mantenimiento, es necesario controlar formalmente los cambios que vamos a ir haciendo en el software.

Recuerda que entendemos por software no solo el código, sino también documentación y datos. Necesitamos hacer una gestión de la configuración del software dado que los sistemas software tienen una vida larga y van cambiando a lo largo de su vida, con lo que hay que asegurar que los cambios producidos ocasionen el mínimo costo.

Una buena gestión de configuraciones nos debe asegurar la coherencia entre las distintas versiones del producto, la seguridad ante pérdidas de software o de personal y la posibilidad de reutilizar el software. No es raro que se pierda alguna versión o algún cambio hecho y suponga un problema importante en el proyecto. O simplemente que se vaya o se ponga enferma una persona y no haya nadie sepa qué es lo que estaba haciendo y cómo continuar con su trabajo.

Estas situaciones se controlan con una buena gestión de configuraciones que, además, es la encargada de mantener la integridad de los productos generados durante el desarrollo y a lo largo de todo el ciclo de vida.

Elementos de configuración del software y líneas base

En la gestión de configuraciones de software encontramos dos conceptos importantes:

- el elemento de configuración del software: Un elemento de configuración del software es cada uno de los componentes básicos de un producto software que cumplen dos condiciones:
  - evoluciona a lo largo del ciclo de vida.
  - nos interesa controlar esa evolución.

  A cada elemento le asignaremos un identificador y un nombre, ya que cada uno debe ser único.
  Por ejemplo, el contrato no evoluciona, por lo que no sería un elemento de gestión de configuración. Tampoco lo sería una maqueta, porque evoluciona hasta un cierto momento y después ya no hacemos uso de ella.

  Sin embargo, el documento de análisis o de diseño sí serían elementos de configuración, ya que evolucionan durante todo el ciclo de vida y, además, nos interesa controlar los cambios que se hagan sobre ellos.

- la línea base: es una configuración de referencia en el ciclo de vida del software a partir de la cual las revisiones se realizarán de manera formal.
  Es decir, se encarga de controlar los cambios en el software pero sin impedir llevar a cabo aquellos que estén justificados.

  Por ejemplo, cuando finalizo mi etapa de análisis, marco conceptualmente una línea base. De ella van a formar parte todos los productos software que he realizado hasta entonces, que estén en una versión estable y que cumplen las dos condiciones que hemos dicho: que evolucionen y que nos interese controlar esa evolución.

  En este ejemplo serían el plan de proyecto y el documento de especificación de requisitos

Esto significa que, a partir de este momento, en el que estos dos productos ya forman parte de una línea base, los cambios que hagamos sobre ellos deben ser justificados y se hacen de manera controlada.

Normalmente existe una base de datos de proyectos software donde voy introduciendo los elementos de configuración cuando ya forman parte de una línea base.
El hacer un cambio de alguno de ellos de forma controlada significa que:

- Me tengo que asegurar de que el cambio está justificado y autorizado.
- Sacarlo de la base de datos, hacer los cambios necesarios, volver a introducirlo y notificar el cambio a las personas que les afecte.

De esta forma, aseguramos que todos los cambios que se deben hacer, se hacen.

Pero ya no es como cuando estamos en pleno desarrollo del producto, cambiándolo continuamente y por varias personas. Hasta que no esté acabada una primera versión estable, no podemos seguir un procedimiento formal porque ralentizaríamos mucho el desarrollo,

Tanto los elementos de configuración del software como las líneas base los definimos al principio del proyecto, en la fase de planificación. En cada proyecto, podemos definir las líneas base que queramos.

Por defecto, las más habituales son: las más habituales son:

- La línea base funcional, que tiene lugar después de análisis.
- La línea base de diseño, después de la etapa de diseño.
- La línea base de producto, que se establece después de las pruebas del sistema.
- Y, la línea base operativa, que tiene lugar cuando el producto final ya se ha entregado al cliente y empieza a ser utilizado.

Suele ser útil definirnos las líneas base coincidiendo con los hitos del proyecto.

Estos dos conceptos tienen una gran potencia, ya que nos permite ir desarrollando nuestro código o escribiendo nuestros documentos y, cuando ya hay una primera versión estable, controlar los cambios que se realizan. De esta forma, el proyecto no se nos va de las manos y aseguramos la coherencia entre los productos.

Actividades de gestión de configuraciones:

Identificación

- identificación de elementos: consiste en identificar la estructura del producto software y sus elementos, haciéndolos únicos y accesibles.
- La identificación de líneas base consiste en identificar las líneas base del proyecto y asignar a cada una los elementos de gestión de la configuración que va a contener.

Control:

- control de versiones consiste en gestionar las versiones de los elementos de configuración creados durante todas las fases del ciclo de vida.

- control de cambios consiste en controlar los cambios que se producen a lo largo del ciclo de vida de un producto software.

Aseguramiento de calidad

¿Cuándo podemos decir que un software es de calidad?

La calidad de un proyecto software depende tanto de que el producto como su proceso sean de calidad, aun teniendo presente que es imposible asegurar algo al cien por cien.

Cuando hablamos de la calidad de un producto hacemos referencia a que el producto cumpla los requisitos definidos.

- los requisitos funcionales,
- los requisitos no funcionales
- los requisitos implícitos, como que funcione de manera consistente, no tenga errores, etc.

Por su parte, un proceso será de calidad cuando sea eficiente, productivo y cumpla con las estimaciones previstas.

Según el estándar IEEE, la calidad del software es el grado en el que un sistema, componente o proceso cumple los requisitos especificados y las necesidades o expectativas del cliente o usuario.

Pero vamos más allá. En general, un proyecto será de calidad si satisface tanto al cliente como a los desarrolladores, es decir, a todos los participantes.
No sería de calidad si el cliente queda muy contento pero el desarrollador, o la empresa desarrolladora ha perdido mucho dinero en el proyecto, por ejemplo.

Por lo tanto, tenemos que tender a que en un proyecto todos salgan ganadores, lo que se conoce como Win-Win.
Hay factores que determinan la calidad del software. Algunos que pueden ser medidos directamente y otros que no pueden ser medidos directamente.

Entre los factores medibles encontramos:

1. los errores reportados
2. el número de documentos generados
3. el número de peticiones de cambio
4. el número de test de pruebas satisfactorios, etc.

Y entre los que no pueden ser medidos directamente están:

1. la facilidad de uso
2. la facilidad de pruebas
3. la facilidad de mantenimiento, etc.

Métricas de calidad

Vamos a hablar de las medidas de aseguramiento de calidad del software, entendiendo estas como las medidas que podemos aplicar en nuestro proyecto para asegurar la calidad.
O, si nuestro proyecto ya ha finalizado mal, cómo podemos saber qué ha ocurrido y qué medidas aplicar para que lo sucedido no vuelva a pasar y los siguientes proyectos no tengan estos problemas.

Las medidas de aseguramiento de calidad del software se pueden dividir en

1. medidas constructivas
2. medidas analíticas.

Los tipos de **medidas constructivas** son:

- las medidas técnicas, que nos proporcionan las distintas técnicas y procedimientos de ingeniería del software para acometer las distintas tareas con calidad.
- Las medidas organizativas que engloban a realización y seguimiento de planes,
- y las medidas humanas, que establecen la formación que necesitan los desarrolladores para realizar su trabajo con mayor calidad.

Por otra parte, las **medidas analíticas** se dividen en:

- dinámicas requieren la ejecución del objeto que está siendo medido o aprobado, es decir, hace referencia a las pruebas de software. Aplicar medidas dinámicas significa hacer pruebas del código.
- Estáticas. analizan el objeto sin necesidad de ejecutarlo y, principalmente, son revisiones formales y auditorías.

Las **revisiones** son reuniones formales donde se analizan de forma estructurada los resultados, parciales o totales, de un proyecto software. Sirven para detectar desviaciones del producto o proceso respecto a las especificaciones iniciales y se pueden realizar en todas las fases del ciclo de vida, aunque sean especialmente relevantes en las primeras para detectar desviaciones o errores cuanto antes y solventarlas. Los revisores deben ser externos al proyecto.
Pero, hay que decir que en el proceso de revisión no siempre hay reuniones. Por ejemplo, si revisamos un documento, se trata de leer el documento y extraer todo lo que pensemos que está mal y afecta a la calidad de ese producto.

Lo normal es, si se revisa entre varios, reunirse a continuación con el equipo de desarrollo para comunicarles los resultados, pero sí hay un único revisor, en ocasiones, no es necesaria la reunión.

Las **auditorías** tienen como objeto realizar una investigación para determinar, por una parte, el grado de cumplimiento de estándares, requisitos, procedimientos y métodos definidos, y por otras, la efectividad del proceso llevado acabo. Puede ser de producto, cuando se cuantifica el grado de conformidad de un producto con los requisitos y especificaciones definidas, o de proceso, donde se evalúa el proceso de desarrollo o de gestión para determinar qué se puede mejorar.

En ambos casos, revisiones y auditorías se finalizan con un informe donde se describe el producto o proceso revisado o auditado y los resultados obtenidos.

En cuanto a los resultados, es importante, además de decir los defectos encontradoso las áreas problemáticas, emitir recomendaciones sobre posibles mejoras.
En ese sentido, los informes de revisión y de auditoría deben ser constructivos, donde no solo digamos los problemas sino cómo se pueden solucionar.

Actividad no calificable:

Para las siguientes actividades o atributos de productos o procesos de un proyecto software, indica la o las medidas de calidad más apropiadas que debes aplicar en cada caso.

1. Corrección del Plan de Aseguramiento de Calidad

- [x] Revisiones
- [ ] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [x] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

2. Identificación acertada de elementos de configuración del software y de las líneas base

- [x] Revisiones
- [x] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

3. Completitud de los casos de prueba creados

- [x] Revisiones
- [ ] Auditorías
- [x] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

✔

4. Implantación eficiente del plan de aseguramiento de calidad

- [x] Revisiones
- [x] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

✔

5. Corrección del diseño

- [x] Revisiones
- [ ] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [x] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

✔

6. Estrategia de mantenimiento utilizada

- [ ] Revisiones
- [x] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

7. Productividad y rendimiento del equipo de trabajo

- [ ] Revisiones
- [x] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [x] Medidas constructivas humanas

✔

8. Completitud del contrato de mantenimiento

- [x] Revisiones
- [ ] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

9. Validación del producto final

- [ ] Revisiones
- [ ] Auditorías
- [x] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

10. Seguimiento del plan de mantenimiento

- [ ] Revisiones
- [x] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

11. Completitud de la fase de análisis

- [x] Revisiones
- [x] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

12. Eficiencia en la gestión de requisitos a lo largo de todo el ciclo de desarrollo

- [ ] Revisiones
- [x] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

13. Corrección de la asignación de responsabilidades y roles por parte del director de proyecto.

- [ ] Revisiones
- [x] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

14. Efectividad en el proceso de catalogación y priorización de peticiones de mantenimiento

- [ ] Revisiones
- [x] Auditorías
- [ ] Medidas dinámicas (pruebas)
- [ ] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

15. Usabilidad de la base de datos de proyecto

- [ ] Revisiones
- [ ] Auditorías
- [x] Medidas dinámicas (pruebas)
- [x] Medidas constructivas técnicas
- [ ] Medidas constructivas organizativas
- [ ] Medidas constructivas humanas

Usabilidad

ISO 9126 es un estándar internacional para la evaluación de la calidad del software y, aunque posteriormente fue reemplazado por un conjunto de normas, sigue siendo una referenciapara la calidad del software.

Este estándar clasifica la calidad del software mediante una serie de características:

- funcionalidad
- fiabilidad
- usabilidad
- eficiencia
- mantenibilidad
- portabilidad.

Vamos a centrarnos en la usabilidad, ya que es la característica que más recientemente se ha incorporado y a la que actualmente se le está dando una gran relevancia.

De forma muy simple, usabilidad es facilidad de uso, pero dentro de las muchas definiciones que hay sobre usabilidad, vamos a ver dos.

Una establece que la usabilidad se refiere al grado en que un producto puede ser usado por usuarios específicos para conseguir metas específicas con efectividad, eficiencia y satisfacción dado un contexto específico de uso.

Con esta definición vemos que la usabilidad no es una característica genérica. Hace referencia a que un producto sea fácil de utilizar para el usuario para el que está diseñado bajo las condiciones de entorno para las que está especificado.

Por otra parte, la usabilidad también se define como la medida de la calidad de la experiencia que tiene un usuario cuando interactúa con un producto o sistema.

Fijémonos aquí en la palabra experiencia. La usabilidad se mide en gran parte como la sensación que le queda al usuario después de utilizar el producto. Tanto el recuerdo como la respuesta emocional son métricas de la usabilidad de un software.

También el número de errores cometidos por un usuario al hacer una tarea en el sistema software y el tiempo requerido para concluir dicha tarea.

Por ello, para establecer la usabilidad de un sistema software, tendremos primero que identificar a los usuarios potenciales y su perfil: qué saben y qué van a aprender, cuál es su contexto de trabajo y bajo qué circunstancias van a utilizar nuestro sistema software.

En general, podemos decir que para que un sistema se considere usable debe ser:

- fácil de aprender y de utilizar
- flexible
- robusto

- intuitivo
- tener el tiempo de respuesta requerido
- etc.

Como hemos venido diciendo a lo largo de este curso, un producto software puede estar técnicamente muy bien hecho, pero si el usuario no lo utiliza, es un fracaso. Y aquí la usabilidad tiene mucho que ver. Démosle la importancia que tiene.

Documantación

En el **plan de gestión de configuración del software** se definen los procedimientos asociados a la implantación de la gestión de configuraciones dentro de un proyecto software. Incluye, además, organización, responsables, actividades y herramientas.

Por su parte, **el plan de aseguramiento de la calidad del software** recoge cómo se va a asegurar la calidad en el software a lo largo del proyecto y cómo se va a evaluar.

Ambos se realizan al inicio del proyecto, en la etapa de planificación.