

Capítulo 3

El ciclo de vida del *software* de gestión de sistemas

Contenido

1. Introducción
2. Modelos del ciclo de vida del *software*
3. Descripción de las fases en el ciclo de vida del *software*
4. Calidad del *software*
5. Resumen

1. Introducción

En los inicios del desarrollo de *software* existía un inmenso caos, los programadores abordaban los proyectos sin rumbo fijo. Cuando el mundo comenzó a tomar conciencia de que el desarrollo del *software* no se podía tomar a la ligera, comenzaron a surgir los modelos prescriptivos de proceso, modelos que han traído un orden y han proporcionado un camino razonablemente efectivo a seguir.

Sin embargo, estos modelos prescriptivos no alcanzan el resultado óptimo cuando existen cambios en el *software*. Y, aunque intentan reemplazar a los convencionales no son la panacea del *software*, pero pueden significar la clave para encontrarla.

Los procesos del *software* son complejos, ya que dependen de la toma de decisiones. Por ello y porque cada sistema que se desarrolla es distinto, no existe el proceso ideal.

2. Modelos del ciclo de vida del *software*

Los modelos de procesos intentan dar una descripción particular y no definitiva de un proceso *software*. Son abstracciones que se pueden utilizar para abordar el desarrollo *software* y que a lo largo del tiempo han demostrado su efectividad en muchos proyectos.

Se puede pensar en ellos como los marcos de trabajo del proceso que pueden ser ampliados con mecanismos y que son adaptados a las necesidades específicas de cada sistema.



Sabía que...

Muchas organizaciones desarrollan su propio modelo o enfoque para el desarrollo *software*.

A continuación, se van a explicar algunos de los modelos más conocidos que se han llegado a utilizar en el desarrollo de *software* y que han intervenido en los modelos que se utilizan actualmente.

2.1. Modelo en cascada

Este modelo se denomina también modelo lineal o ciclo de vida clásico. Es el modelo más antiguo que intenta dar un enfoque metodológico y riguroso al desarrollo del *software*, definiendo para ello distintas etapas involucradas en todo proceso.

Las etapas que definen este modelo son: preanálisis, análisis, diseño, desarrollo, pruebas, implantación y mantenimiento. Cada etapa está compuesta por un conjunto de actividades que la caracterizan y que son claves para la siguiente etapa.

El modelo establece un enfoque secuencial para el desarrollo del *software*.

Así, en la fase de **pre análisis** existe un estudio de viabilidad, alcance del sistema, y estudio de las necesidades. Es en esta etapa donde nace la idea de lo que se pretende desarrollar, resolviendo incógnitas como la factibilidad desde el punto de vista económico, técnico y operativo del proyecto. Es decir, es en este momento donde se decide si vale la pena o no desarrollar el sistema. Si el resultado es satisfactorio, el siguiente paso sería realizar la planificación general de las actividades a llevar a cabo para el desarrollo.

Tras esta fase, se pasa a la etapa de **análisis**, donde se estudian y elaboran los requisitos del sistema, el conjunto de todos estos requisitos compone una descripción completa del comportamiento del sistema a desarrollar. Se suele elaborar en un lenguaje informal donde el cliente y el desarrollador pueden llegar a un entendimiento sobre el funcionamiento previsto para el sistema.

Una vez elaborada esta descripción, se pasa a la siguiente etapa, que se encarga de clasificar los requisitos y obtener una arquitectura del sistema. El resultado de esta etapa debe ser un conjunto de diagramas que especifican completamente el sistema tanto desde el punto de vista estructural como funcional.

La etapa de **desarrollo** es la primera etapa técnica e involucra a la mayor cantidad de personal disponible para el proyecto. Se alternan las tareas de codificación y documentación del *software*. Suele ser la parte más mecánica del desarrollo si la planificación y el diseño son correctos. Si, por el contrario, estas etapas no están correctamente desarrolladas, los problemas que se encuentren en la etapa de desarrollo pueden ser inmanejables y en muchísimos casos provocarán severos retrasos en el desarrollo, aumentando el coste del proyecto.

Tras la fase de codificación se realizan las pruebas con las que se somete el nuevo sistema creado a rigurosos test de calidad y funcionamiento. En esta fase, se llevan a cabo tanto pruebas de integridad del *software* como pruebas de funcionalidad.

Tras superar la fase de **pruebas** y después de haber certificado que el *software* está completamente disponible para su producción, se pasa a la etapa de **implantación**. En esta etapa, se instala el *software* y se entrena a los usuarios que van a manejarlo.

Por último, existe una fase de **mantenimiento** que se puede asumir como una etapa fuera del contexto del propio desarrollo del proyecto. Se compone de tareas como corrección de errores, mejoras del *software* y la posibilidad de añadir funcionalidad es según el análisis posterior.



Actividades

1. Según el modelo clásico de desarrollo del *software*, cada etapa es clave en el ciclo de vida del sistema. Pero ¿cuál es la etapa que requiere más recursos en el proceso de desarrollo? Justifique la respuesta.

2.1. Modelo en cascada

Este modelo se denomina también modelo lineal o ciclo de vida clásico. Es el modelo más antiguo que intenta dar un enfoque metodológico y riguroso al desarrollo del *software*, definiendo para ello distintas etapas involucradas en todo proceso.

Las etapas que definen este modelo son: preanálisis, análisis, diseño, desarrollo, pruebas, implantación y mantenimiento. Cada etapa está compuesta por un conjunto de actividades que la caracterizan y que son claves para la siguiente etapa.

El modelo establece un enfoque secuencial para el desarrollo del *software*.

Así, en la fase de **pre análisis** existe un estudio de viabilidad, alcance del sistema, y estudio de las necesidades. Es en esta etapa donde nace la idea de lo que se pretende desarrollar, resolviendo incógnitas como la factibilidad desde el punto de vista económico, técnico y operativo del proyecto. Es decir, es en este momento donde se decide si vale la pena o no desarrollar el sistema. Si el resultado es satisfactorio, el siguiente paso sería realizar la planificación general de las actividades a llevar a cabo para el desarrollo.

Tras esta fase, se pasa a la etapa de **análisis**, donde se estudian y elaboran los requisitos del sistema, el conjunto de todos estos requisitos compone una descripción completa del comportamiento del sistema a desarrollar. Se suele elaborar en un lenguaje informal donde el cliente y el desarrollador pueden llegar a un entendimiento sobre el funcionamiento previsto para el sistema.

Una vez elaborada esta descripción, se pasa a la siguiente etapa, que se encarga de clasificar los requisitos y obtener una arquitectura del sistema. El resultado de esta etapa debe ser un conjunto de diagramas que especifican completamente el sistema tanto desde el punto de vista estructural como funcional.

La etapa de **desarrollo** es la primera etapa técnica e involucra a la mayor cantidad de personal disponible para el proyecto. Se alternan las tareas de codificación y documentación del *software*. Suele ser la parte más mecánica del desarrollo si la planificación y el diseño son correctos. Si, por el contrario, estas etapas no están correctamente desarrolladas, los problemas que se encuentren en la etapa de desarrollo pueden ser inmanejables y en muchísimos casos provocarán severos retrasos en el desarrollo, aumentando el coste del proyecto.

Tras la fase de codificación se realizan las pruebas con las que se somete el nuevo sistema creado a rigurosos test de calidad y funcionamiento. En esta fase, se llevan a cabo tanto pruebas de integridad del *software* como pruebas de funcionalidad.

Tras superar la fase de **pruebas** y después de haber certificado que el *software* está completamente disponible para su producción, se pasa a la etapa de **implantación**. En esta etapa, se instala el *software* y se entrena a los usuarios que van a manejarlo.

Por último, existe una fase de **mantenimiento** que se puede asumir como una etapa fuera del contexto del propio desarrollo del proyecto. Se compone de tareas como corrección de errores, mejoras del *software* y la posibilidad de añadir funcionalidad es según el análisis posterior.



Actividades

1. Según el modelo clásico de desarrollo del *software*, cada etapa es clave en el ciclo de vida del sistema. Pero ¿cuál es la etapa que requiere más recursos en el proceso de desarrollo? Justifique la respuesta.

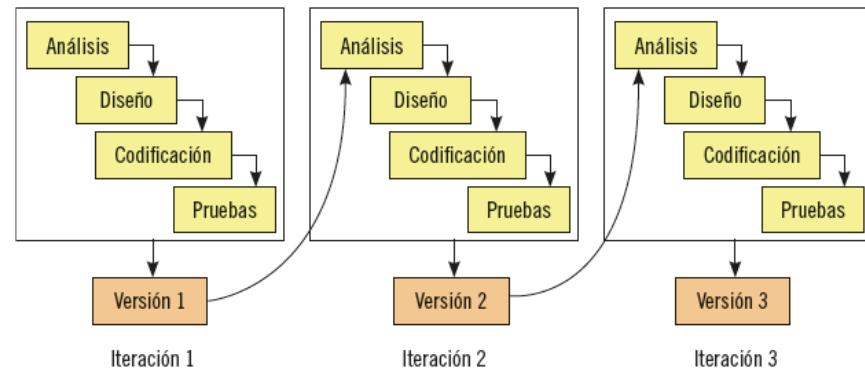
2.2. Modelo iterativo

El modelo anterior es la primera aproximación que se da para acometer el proceso de desarrollo de un proyecto. Sin embargo, pronto aparecen los primeros problemas que derivan de este modelo. En concreto, en el modelo en cascada se llega a un acuerdo con el cliente en la primera fase del proyecto sobre la funcionalidad del sistema y no existe ningún vínculo con este hasta la finalización del sistema. Es decir, no existe retroalimentación por parte del cliente.

Surge entonces el modelo iterativo, para minimizar el riesgo que supone no contar con la opinión del cliente durante todo el desarrollo del sistema.

Este modelo se compone de iteraciones, donde en cada iteración se produce la secuencia de etapas de un modelo en cascada clásico. Una iteración es, por tanto, un conjunto de períodos de tiempo donde se produce una versión ejecutable del producto y la documentación necesaria. Cada iteración posee una fase de análisis para determinar cuál es la mejora que se va a realizar sobre el sistema en esa iteración y existe una fase de entrega del módulo elaborado.

Secuencia de etapas del proceso iterativo



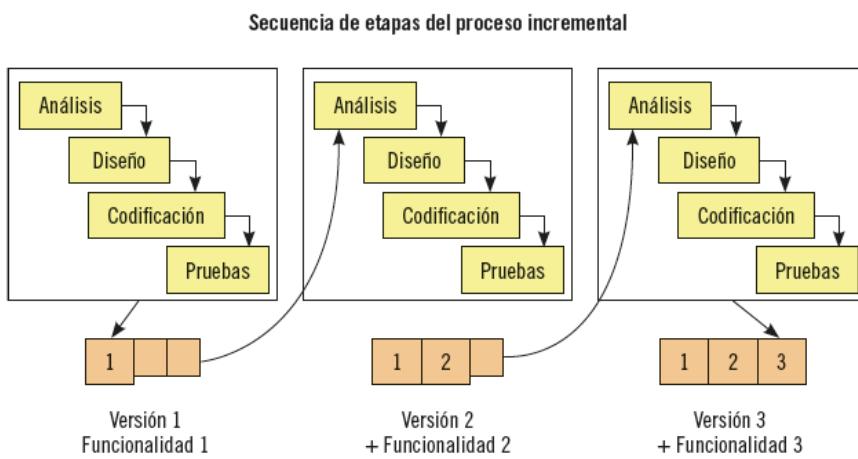
De esta manera, los objetivos de cada iteración se establecen en función de la evaluación por parte del cliente de las anteriores, corrigiéndolas o proponiendo mejoras.

Este modelo permite entre otras cosas disminuir los riesgos que supone un cambio en los requisitos del sistema durante el desarrollo, reduciendo costes y proponiendo una participación activa del cliente dentro del proyecto.

2.3. Modelo incremental

El modelo incremental se centra en desarrollar el sistema en partes, de forma que se van entregando a medida que se van completando. Este modelo se adapta mejor a los sistemas

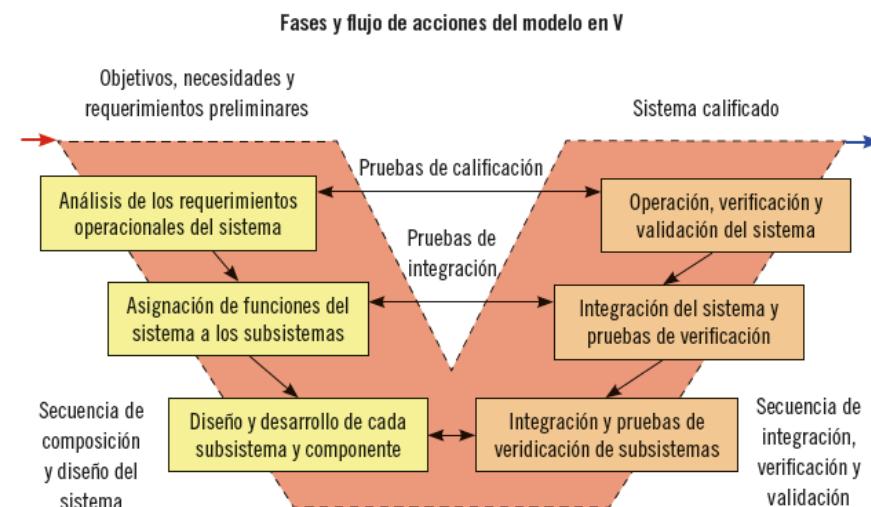
con una gran complejidad funcional. A veces, el modelo incremental se confunde con el iterativo y viceversa. La diferencia principal es que el modelo iterativo produce en cada iteración una versión mejorada de la anterior mientras que el modelo incremental añade en cada iteración una parte de la funcionalidad del sistema. Por lo tanto, en cada iteración se obtiene una versión estable del sistema.



Las dos claves de este modelo son: comenzar con una implementación simple de los requisitos del sistema e ir mejorando y añadiendo nuevas funcionalidad es hasta que el sistema esté completo, y, por otro lado, usar la experiencia y conocimientos adquiridos en anteriores iteraciones.

2.4. Modelo en V

El modelo en V es una evolución del modelo de ciclo de vida en cascada. En él, las etapas se organizan en una estructura en forma de V. En el lado izquierdo se representa la descomposición de las necesidades: requisitos y especificación del sistema, diseño, etc. Mientras que en el lado derecho se muestra la integración de las piezas y la verificación del sistema.





Importante

La idea esencial del modelo en V es que las pruebas comiencen a realizarse lo antes posible.

Este modelo contiene las etapas clásicas del desarrollo: una fase de análisis, una fase de diseño, programación y verificación. Sin embargo, existe una conexión directa entre las etapas de pruebas y las de desarrollo. Cualquier error de diseño detectado en la etapa de pruebas conduce al rediseño y nueva programación del código afectado. Las etapas de desarrollo y diseño deben realizarse de forma paralela con las actividades de pruebas y debiendo existir una comunicación activa entre los técnicos de pruebas, analistas y desarrolladores.

La principal ventaja de este modelo es que la relación entre las etapas de desarrollo y de pruebas facilita la localización de los errores. Es un modelo sencillo, en el que se especifican correctamente los roles de las distintas pruebas a realizar y además involucra de forma activa al usuario en ellas. Entre sus desventajas, destaca que el cliente no forma parte activa del proyecto durante su desarrollo, que el producto final puede no reflejar todos los requisitos del usuario y, además, que las pruebas pueden ser costosas y no lo suficientemente efectivas.



Actividades

2. En el modelo en V del desarrollo de software, la distancia entre los analistas, diseñadores y programadores con respecto a los técnicos de pruebas se reduce. Pero, si se produce un cambio en las especificaciones del sistema cuando el proceso está avanzado, ¿cómo afectaría ese cambio al sistema?

2.5. Modelo basado en componentes (CBSE)

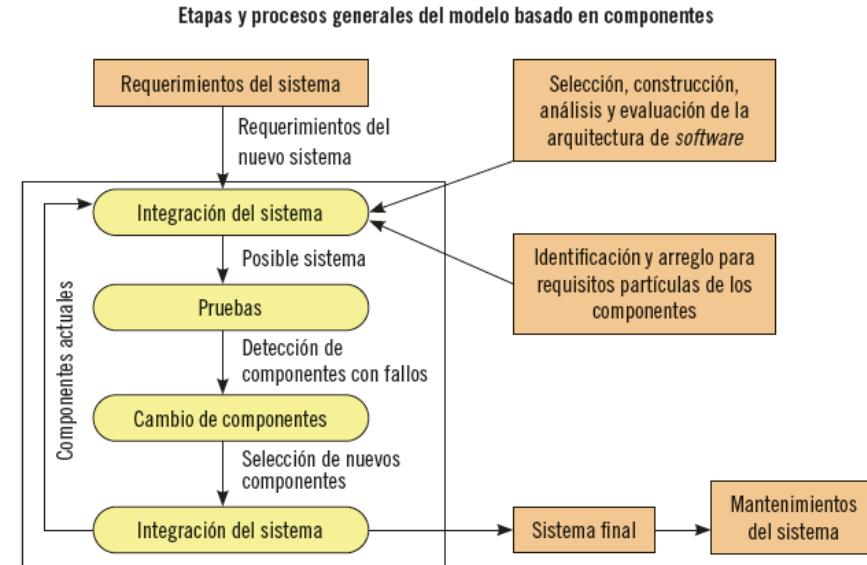
Hasta ahora, los modelos vistos no asumían una dependencia directa con alguna metodología de programación. Este modelo exige un desarrollo basado en componentes. El diseño basado en componentes es una evolución del desarrollo orientado a objetos. Busca, entre muchos objetivos, reducir los tiempos de desarrollo y de implementación y, por lo tanto, reducir los costes del proyecto.

En un enfoque orientado a objetos, el software es desarrollado de acuerdo a modelos mentales de representación de los conceptos de la vida real o imaginaria. Sin embargo, el enfoque de componentes se basa en la idea del desarrollo a partir de módulos prefabricados. De esta manera, se puede definir un componente como una parte casi independiente y reemplazable de un sistema que compone una parte de la funcionalidad dentro del contexto de la arquitectura del sistema a desarrollar. Debe tener una interfaz bien definida para que facilite la comunicación al resto de los componentes.

El modelo se compone de las etapas clásicas:

- **Análisis de requisitos:** donde, como se ha visto, se realiza el estudio de las necesidades del negocio y se descubren y expresan los requisitos funcionales y no funcionales del sistema.

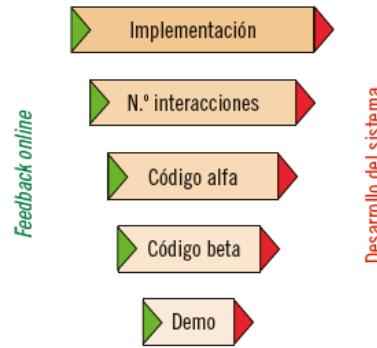
- **Selección, construcción, análisis y evaluación de la arquitectura del sistema:** se define el sistema en función de los componentes computacionales y la interacción entre ellos. Se definen las propiedades que deben poseer los componentes del sistema.
- **Identificación y arreglo para requisitos particulares del componente:** los componentes se seleccionan en función de los requisitos funcionales y de calidad. Y evaluar si es necesaria su modificación.
- **Integración del sistema:** determinar la comunicación y coordinación entre los componentes. Realizar el ensamblado y las pruebas del sistema para determinar su integridad.
- **Pruebas:** posteriormente a la integración, se deben realizar las pruebas para evaluar la funcionalidad de los componentes que han integrado.
- **Mantenimiento:** en esta etapa se vigila que el sistema se comporte exactamente como se determinó en la fase de análisis, corrigiendo cualquier comportamiento que no se adapte a los requisitos.



2.6. Modelo de desarrollo rápido (RAD)

El modelo RAD tiene como objetivo minimizar el tiempo de desarrollo basándose en la implementación de una versión prototípico y, a continuación, integrar la funcionalidad de forma iterativa para satisfacer todos los requisitos de cliente.

Etapas del modelo de desarrollo rápido



Este modelo surge debido a que a veces es muy difícil que el cliente sea capaz de definir todos los requisitos al inicio del proyecto. Se necesita de una capacidad de adaptación a los cambios que se producen en la especificación durante el desarrollo del sistema y eso es lo que este modelo intenta conseguir.

Los puntos clave que caracterizan a este modelo son:

- La comunicación con los *stakeholders* es fundamental para resolver la ambigüedad en el análisis de requisitos.
- Mayor importancia en la codificación en detrimento de la documentación.
- Asumir que se van a producir cambios e integrarlos en el proyecto de forma natural.



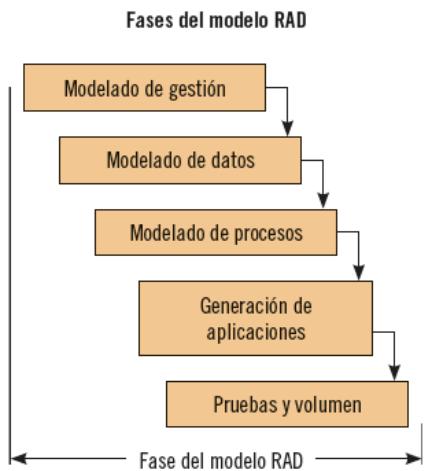
Definición

Stakeholder

Es toda parte interesada en el desarrollo del sistema.

Las fases de las que se compone el modelo RAD son:

- **Modelado de gestión.** El flujo de información se modela respondiendo a las siguientes preguntas: ¿Qué información conduce el proceso de gestión? ¿Qué información se genera? ¿Quién la genera? ¿A dónde va la información? ¿Quién la procesa?
- **Modelado de datos.** El flujo anterior se refina para obtener un conjunto de objetos de datos necesarios para el modelo de negocio. Definiendo además los atributos de estos.
- **Modelado de proceso.** Se elabora el conjunto de interacciones y comunicaciones entre objetos para ajustarlo a los flujos de proceso del negocio.
- **Generación de aplicaciones.** Se utilizan técnicas de última generación para determinar los módulos o componentes necesarios para componer el sistema.
- **Pruebas de entrega.** Se realizan pruebas sobre aquellos módulos creados o modificados desde cero para determinar su validez en el sistema. Se asume que los componentes reutilizados ya han sido probados previamente.



RAD se ayuda de técnicas estructuradas y prototipos para que el cliente evalúe si el sistema cumple con los requisitos. El desarrollador primero construye los modelos de datos y los modelos de procesos de negocio. Los prototipos permiten a los usuarios y a los analistas verificar los requisitos y refinar los modelos de datos y procesos.

Como ventajas, cabe destacar: la velocidad de desarrollo, la visibilidad temprana debido al uso de prototipos para la verificación de requisitos, la flexibilidad y los ciclos de desarrollo.

La principal desventaja del modelo es que puede conducir a una sucesión de prototipos que no se adapten a la aplicación final objetivo.



Actividades

3. ¿Cuál sería el impacto de un cambio en los requisitos del sistema en una etapa de desarrollo avanzada si utilizamos el desarrollo rápido como modelo de ciclo de vida?

2.7. Pautas para la selección de la metodología más adecuada. Ventajas e inconvenientes

Algunos de los factores que influyen a la hora de elegir un modelo para el desarrollo de un sistema son:

- Disponibilidad de recursos, ya sean económicos, de tiempo, de equipo, de personal cualificado, etc.
- Entender los requerimientos.
- Dominio del problema, si se tienen los conocimientos para dar solución al problema central.
- Complejidad y magnitud del proyecto.

No existe una metodología universal para hacer frente con éxito a cualquier proyecto de desarrollo de software. Toda metodología debe ser adaptada al contexto del proyecto (recursos técnicos y humanos, tiempo de desarrollo, tipo de sistema, etc.).



Sabía que...

A lo largo de la historia, los métodos tradicionales han intentado ajustarse a la mayor cantidad de situaciones y problemas dentro del desarrollo, sobre todo en pequeños proyectos, donde los cambios de requisitos son muy frecuentes.

Las **metodologías ágiles** son una solución para proyectos de pequeña y mediana envergadura. La cualidad más destacada e importante es la sencillez, tanto en su aprendizaje como en su aplicación. De esta forma, se reducen los costes de implantación en un equipo de desarrollo.

Sin embargo, el inconveniente es que están dirigidas a equipos pequeños o medianos, donde el contexto interno debe estar favorecido por un ambiente donde sea importante la comunicación y colaboración entre todos los miembros del equipo. Esto es muy importante, ya que cualquier resistencia del cliente o del equipo de desarrollo hacia estas prácticas puede llevar al fracaso de la metodología.

En la siguiente tabla se muestran las principales ventajas y desventajas de cada uno de los modelos metodológicos.

Ventajas	Desventajas	Proyectos en que puede ser usado
Modelo en cascada		
<ul style="list-style-type: none">- La planificación es sencilla.- La calidad del producto resultante es alta.- Permite trabajar con personal poco cualificado.	<ul style="list-style-type: none">- Modelo lineal.- Necesidad de tener todos los requisitos al principio.- No hay posibilidad de corregir errores a tiempo.- Aumento de los costos del desarrollo.	<ul style="list-style-type: none">- Aquellos para los que se dispone de todas las especificaciones desde el principio.- Proyectos de reingeniería.- Proyectos complejos que se entienden bien desde el principio.
Modelo iterativo		

- El cliente puede monitorizar el desarrollo a través de los entregables de cada iteración.
- No hace falta que todos los requisitos estén totalmente definidos al inicio del desarrollo.
- Desarrollo en pequeños ciclos.
- Mejor gestión de los riesgos y entregas.

Modelo incremental

- Difícil evaluar el costo final.
- A veces es complicado ajustar los requisitos a las iteraciones.
- Introducir cambios en medio del desarrollo produce "parches" en el software.

- Sistemas de tiempo no real.
- Sistemas de bajo nivel de seguridad.
- Sistemas de bajo índice de riesgo.

- Con un paradigma incremental se reduce el tiempo de desarrollo inicial.
- También provee un impacto ventajoso frente al cliente, que es la entrega temprana de partes operativas del software.
- Permite entregar al cliente un producto más rápido en comparación del modelo en cascada.

- Requiere de mucha planificación, tanto administrativa como técnica.
- Requiere de metas claras para conocer el estado del proyecto.

- Sistemas de tiempo no real.
- Sistemas de bajo nivel de seguridad.
- Sistemas de bajo índice de riesgo.

Modelo en V

<ul style="list-style-type: none"> - Más robusto y completo que el modelo en cascada. - Sencillo y fácil de aprender. - Involucra al cliente en los procesos de pruebas. - Hace explícita parte de la iteración y trabajo a realizar. 	<ul style="list-style-type: none"> - Es difícil que el cliente exponga explícitamente todos los requisitos. - El cliente obtiene el producto al final del ciclo de vida. - No se contempla volver a etapas inmediatamente anteriores del ciclo de vida. - Un proceso mal desarrollado ocasiona una revisión completa de todo el proceso. - Las pruebas son caras y a veces no son suficientemente efectivas. 	<ul style="list-style-type: none"> - Sistemas en los que la diferencia entre lo que se quiere y lo que se entiende es muy pequeña. 	<ul style="list-style-type: none"> - Reutilización de software. - Simplificación de pruebas. - Simplificación del mantenimiento del sistema. - Mayor calidad. El componente puede ser construido y después mejorado. 	<ul style="list-style-type: none"> - Hace falta un gran compromiso en los requisitos. - La actualización de los componentes no está en manos de los desarrolladores. - Genera mucho tiempo el desarrollo del sistema. - El modelo es costoso. 	<ul style="list-style-type: none"> - Sistemas de gran complejidad. - Sistemas con muchos requisitos.
---	---	---	--	---	--

Modelo basado en componentes

Modelo de desarrollo rápido

- Más robusto y completo que el modelo en cascada.
- Sencillo y fácil de aprender.
- Involucra al cliente en los procesos de pruebas.
- Hace explícita parte de la iteración y trabajo a realizar.

- Es difícil que el cliente exponga explícitamente todos los requisitos.
- El cliente obtiene el producto al final del ciclo de vida.
- No se contempla volver a etapas inmediatamente anteriores del ciclo de vida.
- Un proceso mal desarrollado ocasiona una revisión completa de todo el proceso.
- Las pruebas son caras y a veces no son suficientemente efectivas.

- Sistemas en los que la diferencia entre lo que se quiere y lo que se entiende es muy pequeña.

- Reutilización de software.
- Simplificación de pruebas.
- Simplificación del mantenimiento del sistema.
- Mayor calidad. El componente puede ser construido y después mejorado.

- Hace falta un gran compromiso en los requisitos.
- La actualización de los componentes no está en manos de los desarrolladores.
- Genera mucho tiempo el desarrollo del sistema.
- El modelo es costoso.

- Sistemas de gran complejidad.
- Sistemas con muchos requisitos.

Modelo de desarrollo rápido

Modelo basado en componentes

- | | |
|---|---|
| <ul style="list-style-type: none"> - El desarrollo se realiza a un nivel de abstracción mayor. - Los clientes están más involucrados. - Ciclos de desarrollo más pequeños. - Mayor flexibilidad. - Menor codificación manual. - Menos fallos y costo. | <ul style="list-style-type: none"> - Los requerimientos deben ser bien entendidos. - Se requieren múltiples desarrolladores. - Debe existir compromiso entre desarrolladores y clientes. - No adecuado para sistemas que no son mantenidos correctamente. - El progreso es más difícil de medir. - Menos eficiente. - Los prototipos pueden no ser escalables. |
|---|---|

taquilla, el sistema debe dar soporte a la venta de entradas por internet, debiendo estar ambos coordinados. Como técnico de software, debe seleccionar, justificando la elección, el ciclo de vida más adecuado para este sistema.

SOLUCIÓN

No le han detallado el conjunto completo de especificaciones que debe cumplir el sistema, además tampoco han hecho hincapié en el tiempo que debe llevarle como máximo el desarrollo del sistema. Debido a que el cliente debe estar muy presente en el proyecto y, en base a las anteriores características, podría seleccionar un ciclo de vida iterativo.



Actividades

4. Si fuera necesario seleccionar un modelo de ciclo de vida para el desarrollo de un sistema, donde es necesario que los resultados del trabajo se expongan lo antes posible, ¿cuál sería el modelo adecuado?



Aplicación práctica

Se quiere desarrollar un sistema de software que controle la venta de entradas de un cine denominado Cine +. Paralelamente a la venta de entradas en la

3. Descripción de las fases en el ciclo de vida del software

Independientemente del modelo de ciclo de vida a seguir para el desarrollo, se observa que existen determinadas etapas que se repiten en todos los modelos y, dentro de estas etapas, se encuentran numerosas tareas que la caracterizan. A partir de este momento, se

puede asumir un modelo de ciclo de vida secuencial y analizar en profundidad cada una de estas etapas y cómo el resultado de ellas permite avanzar en el proceso de desarrollo.

3.1. Análisis y especificación de requisitos

La fase de análisis es la primera dentro del desarrollo de un sistema. Su principal objetivo es obtener una especificación detallada no ambigua de los requisitos que debe satisfacer el sistema.

Pero el concepto requisito es un poco general, de forma que se va a definir antes de continuar con la explicación del capítulo.

Un **requisito** o **requerimiento** especifica qué es lo que el sistema debe hacer, entendiéndolo desde el punto de vista funcional, además de las propiedades y atributos que deben ser deseables. Un requisito expresa cuál debe ser la función del sistema, pero no determina cómo debe el sistema alcanzar esa función.

Esta es, sin duda, la etapa más importante en el desarrollo de un proyecto de *software* debido a que un error o mal entendimiento por parte del cliente y el analista pueden provocar que el costo del desarrollo aumente considerablemente.

El análisis de requisitos es el conjunto de técnicas y procedimientos que permiten obtener y analizar un modelo de negocio para obtener esos requisitos y elaborar la especificación completa del sistema antes de involucrarse en las fases técnicas.

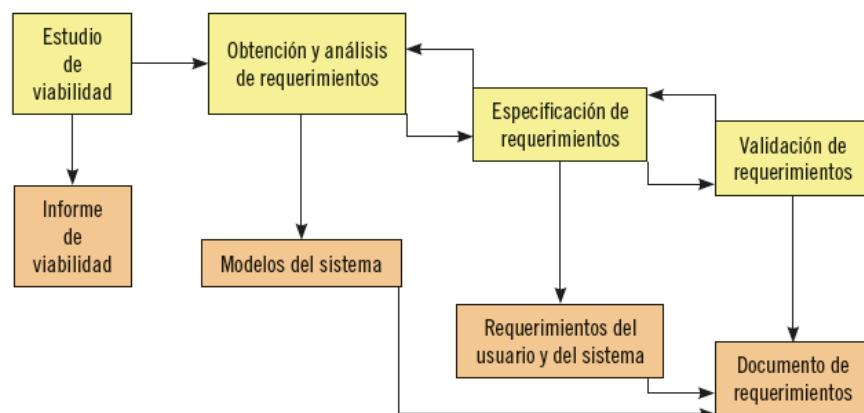
Se podría decir que se trata del primer documento que sirve de contrato entre las partes involucradas en el proyecto.



Importante

El resultado de esta fase es una especificación de la funcionalidad del sistema y las propiedades que debe cumplir. Esta especificación es el medio para llegar a una comprensión completa del sistema por parte de los desarrolladores y el propio cliente.

Secuenciación de tareas que se producen en una etapa de análisis y que dan como resultado el documento de requisitos detallado





Sabía que...

"La carencia de buenos requisitos ha sido la causa del fracaso de proyectos con presupuestos de millones de euros, ha impedido el desarrollo productivo, y ha sido el mayor contribuyente de los costes elevados del mantenimiento del software" (Dr. Raymond: *Yeh in the forward to System and Software Requirements Engineering, IEEE Computer Society Press Tutorial*, Editors: M. Dorfman and R. H. Thayer, 1990).

Tipos de requisitos: funcionales/no funcionales, de usuario, de interfaz, de seguridad y de rendimiento

Los tipos de requisitos se clasifican en función de la pregunta que da como respuesta el propio requisito. De esta forma, se pueden dividir en los siguientes tipos:

- **Ambiente físico.** Este tipo de requisitos intentan especificar aspectos como el tipo de equipo informático necesario para desplegar el sistema, la ubicación exacta de dichos equipos informáticos, así como las restricciones ambientales para que los equipos informáticos así como funcionen correctamente (temperatura, humedad, interferencia magnética).
- **Interfaces.** Especifican si el sistema debe comunicarse con otros sistemas, el tipo de la información que se comunica y sus formatos.
- **Usuarios y factores humanos.** Aspectos como quién usará el sistema, si será un usuario o varios, el nivel de conocimientos informáticos de estos usuarios, el entrenamiento necesario para el manejo del sistema y dificultad de manejo del sistema.

▪ **Funcionalidad.** Estos requisitos responden a qué hará el sistema, cuándo lo hará, si existen varios modos de ejecución de operación, si existen limitaciones de velocidad de ejecución, de respuesta o de rendimiento; y cómo y cuándo puede mejorarse el sistema.

▪ **Documentación.** Determinan el tipo de documentación (*online*, papel, etc.), la cantidad de documentación y a qué tipo de audiencia se orienta.

▪ **Datos.** Se especifica el formato de datos, frecuencia de recepción y envío, su exactitud, grado de precisión de los cálculos, cantidad que fluye por el sistema y si es necesario retener alguno.

▪ **Recursos.** Se utilizan para especificar los recursos materiales y personales que se requieren, determinar las habilidades de los desarrolladores, el espacio físico que debe ocupar el sistema, requisitos de energía, calefacción y acondicionamiento de aire, si debe existir cronograma prescrito para el desarrollo y si existe un límite en el costo del proyecto a nivel de *software* y *hardware*.

▪ **Seguridad.** Responden a si se debe controlar el acceso al sistema o a la información, si es necesario aislar los datos de un usuario de los de otros, con qué frecuencia se deberían de realizar copias de respaldo, dónde deben almacenarse y las precauciones sobre ellas.

▪ **Rendimiento.** Determinan restricciones de tipo temporal que debe cumplir el producto. En algunos casos, velocidad de ejecución, tiempo de respuesta, etc. Los sistemas en tiempo real son los que contienen mayor número de este tipo de requisitos.

Una clasificación muy importante que suelen hacer los analistas es la de requisito **funcional** y **no funcional**. Un requisito funcional sería aquel que determina una característica requerida por el sistema que expresa una capacidad de acción, es decir, una funcionalidad; generalmente expresada en una declaración en forma verbal. Un requisito no funcional sería aquel que determina una característica requerida del proceso de desarrollo, del servicio prestado o de cualquier aspecto de dicho desarrollo. Es decir, una restricción del mismo.



Actividades

5. Si en la fase de análisis obtiene el requisito de que es necesaria la implantación de un sistema distribuido, ¿qué tipo de requisito sería considerando la clasificación vista anteriormente? Justifique su respuesta.

Modelos para el análisis de requisitos

Los modelos de análisis dan una primera representación técnica del sistema. Se componen de un conjunto de diagramas y descripciones textuales que representan los requisitos, la funcionalidad y el comportamiento del sistema. Y, además, proporcionan al desarrollador y al cliente una forma de evaluar los objetivos del sistema cuando haya finalizado el desarrollo.

Los objetivos que debe cumplir todo modelo de análisis son:

- Describir las necesidades del cliente.
- Delimitar el sistema.
- Capturar la funcionalidad del sistema.
- Crear una base para el diseño del sistema.
- Crear los objetivos que se evaluarán una vez terminado el desarrollo.

Para conseguir estos objetivos, existen varios tipos de diagramas que intentan capturar los requisitos del sistema desde varios puntos de vista diferentes. En la mayoría de los casos, todos estos diagramas son complementarios, por lo que todos ellos en conjunto conforman el modelo de análisis, mientras que en otros casos suele bastar con un solo tipo de diagramas.

Categorías de diagramas que ayudan a especificar el modelo de análisis desde cualquier punto de vista del *software*

Elementos basados en escenarios

Casos de Usos, textos
Casos de Usos, diagramas
Diagramas de actividad
Diagramas de carril

Elementos orientados al flujo

Diagrama de flujo de datos
Diagrama de flujo de control
Narrativas de procesamiento

Modelo de análisis

Elementos basados en clases

Diagrama de clases
Paquete de análisis
Modelos CRC
Diagramas de colaboración

Elementos de comportamiento

Diagramas de estado
Diagramas de secuencias

Modelos basados en escenarios

Un caso de uso es una descripción textual sobre un comportamiento funcional del sistema en un escenario específico. Estos modelos ayudan a explicar qué debe hacer el sistema. Cada caso de uso se puede describir en forma de diagrama donde se puede observar la interacción entre los usuarios y el propio sistema. Son los elementos más utilizados para elaborar el modelo de análisis. Un caso de uso debe estar descrito en un lenguaje cercano al cliente.

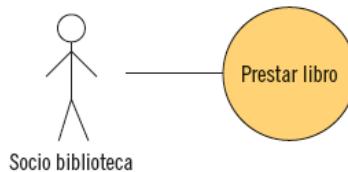


Ejemplo

Imagínese que se quiere construir un sistema para una biblioteca. Existen dos usuarios potenciales del sistema: el bibliotecario y el usuario final o socio de la biblioteca. El sistema podrá realizar las siguientes operaciones:

- | Reservar Libro.
- | Prestar Libro.
- | Dar de alta socio.
- | Dar de baja socio.
- | Actualizar catálogo.

El diagrama del caso de uso correspondiente a la operación "Prestar Libro" sería:



Y una posible descripción textual del mismo:

Nombre	Prestar Libro
Descripción	Un socio intenta obtener en préstamo un libro de la biblioteca.
Actores	Usuario/Bibliotecario.
Precondiciones	El usuario que intenta solicitar el préstamo debe ser socio de la biblioteca. El socio no sobrepasa el número máximo de préstamos simultáneos. El libro solicitado no debe estar prestado o reservado a otro socio.
Flujo Normal	Se introducen los datos del socio. Se introduce el libro a prestar. El sistema valida el préstamo, lo registra. El sistema devuelve la fecha máxima de devolución.
Flujo Alternativo	El sistema detecta un fin inesperado de la conexión y se aborta el proceso.
Postcondiciones	El sistema ha registrado el préstamo del libro.

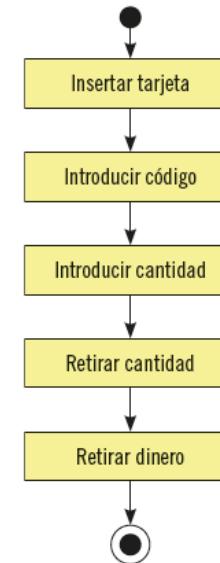
Como se ha visto, un caso de uso debe especificar el comportamiento del sistema, pero no imponer cómo se realizará ese comportamiento.

Los diagramas de actividades corresponden al tipo de análisis de comportamiento, igual que los casos de uso. Sin embargo, a diferencia de estos, los diagramas de actividades buscan representar el comportamiento dinámico del sistema haciendo hincapié en la secuencia de actividades de un proceso del modelo de negocio y las condiciones que producen y modifican esas actividades.

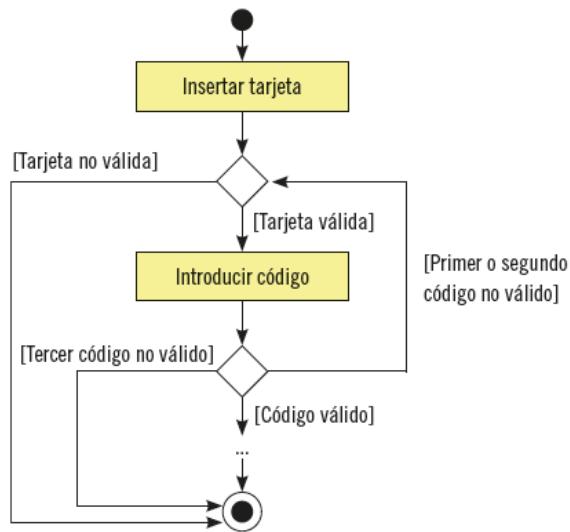


Ejemplo

Imagínese que se quiere elaborar el diagrama de actividades de un sistema que controlará un cajero automático para retirar saldo.



Incluso se podría elaborar un diagrama con un poco más de detalle, si se tiene en cuenta que se pueden insertar tarjetas que no son válidas:

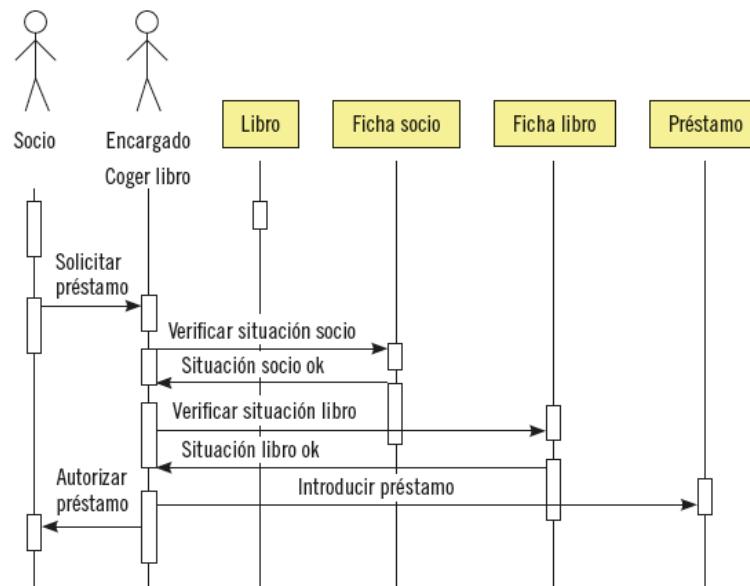


Otro tipo de diagramas muy utilizados son los de secuencia o de carril. Estos sirven para especificar un comportamiento del sistema y cómo interactúan los elementos del sistema para llevar a cabo ese comportamiento.



Ejemplo

Si se vuelve al sistema de gestión de una biblioteca y se quiere especificar el comportamiento del sistema cuando alguien solicita un préstamo de libro, el resultado podría ser el siguiente:



Aquí se puede apreciar cómo se produce la interacción entre los usuarios y el propio sistema. En concreto, cómo se realiza el paso de mensajes entre los objetos del sistema para llevar a cabo la acción.



Actividades

6. Los casos de uso son descripciones textuales de un determinado requisito funcional acordado en la fase de análisis. ¿Por qué no tienen sentido los casos de uso de los requisitos no funcionales de un sistema? Ponga un ejemplo.

Documentación de requisitos

La documentación de requisitos es el paso donde se recopilan todos los requisitos que han sido obtenidos como resultado del análisis. Actualmente, existen dos estándares de documentos que proporcionan un formato muy apropiado para elaborar el documento de requisitos:

- **DRU (Documento de Requisitos de Usuario).** Este tipo de documento contiene una descripción del problema y los objetivos que se esperan alcanzar desde el punto de vista del usuario o cliente.
- **ERS (Especificación de Requisitos de Software).** Es una descripción más detallada y desarrollada de los contenidos en DRU. También puede haber preguntas de cómo deben alcanzarse los objetivos.

ISO/IEC/IEEE 29148:2018 es el estándar que describe el formato ERS de lo que se podría considerar un buen documento de requisitos. Según el estándar, el documento ERS debería contener los siguientes puntos desarrollados:

- Introducción

- Propósito.
- Ámbito del sistema.
- Definiciones, acrónimos y abreviaturas.
- Referencias.
- Visión general del documento.

- Descripción general.

- Perspectiva del producto.
- Funciones del sistema.
- Características de los usuarios.
- Restricciones.
- Suposiciones y dependencias.

- Requisitos específicos.

- Requisitos funcionales.
- Requisitos de rendimiento.
- Requisitos tecnológicos.
- Atributos.

- Apéndices.



Actividades

7. ¿Por qué es tan necesaria la documentación de requisitos antes de que comiencen las etapas técnicas del desarrollo del sistema?

Validación de requisitos

La validación de los requisitos sirve para demostrar que todos los requisitos obtenidos definen el sistema tal y como el cliente desea que funcione.



Importante

Encontrar un error en esta etapa evita el costo de una modificación en etapas posteriores y reduce el impacto de un cambio funcional en mitad del desarrollo del proyecto.

Plantea para cada requisito las siguientes verificaciones antes de que se pueda considerar ese requisito como tal:

- **Verificación de validez.** Intenta identificar si el requisito pertenece o no a las funciones del sistema que solicitaron los stakeholders.
- **Verificación de consistencia.** Intenta comprobar que no existen requisitos contradictorios o restricciones incompatibles.

- **Verificación de completitud.** Se comprueba que el documento que describe los requisitos define todas las funciones y restricciones que los stakeholders propusieron para el sistema.
- **Verificación de realismo.** Realiza un análisis de factibilidad a partir del tipo de tecnología existente, el presupuesto y el tiempo disponible.
- **Verificabilidad.** Todos los requisitos deben poderse someter a pruebas que determinen si se cumplen o no en el sistema desarrollado finalmente.

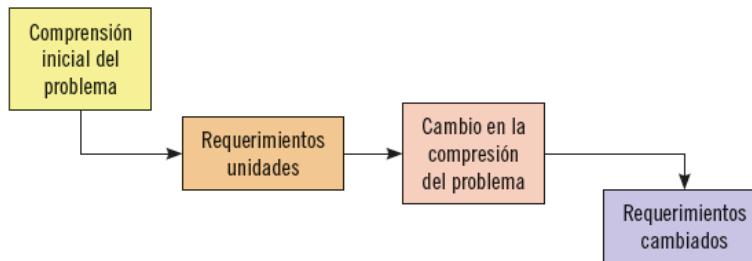
Cuando los requisitos cumplen todas estas pruebas de verificación se puede asegurar que se está ante una descripción completa y exacta del sistema.

Gestión de requisitos

Es normal que, a medida que se desarrolla el sistema, las personas involucradas de forma directa o indirecta en él desarrollen una mejor compresión de lo que quieren que el software haga. Esto desencadena un cambio en las especificaciones de los requisitos y, cuando se produce, puede tener un gran impacto en el desarrollo normal del proyecto. Incluso una vez que el sistema está desplegado en los equipos informáticos y en uso por parte del cliente, pueden surgir nuevos requisitos. "La gestión de requisitos es el proceso de comprender y controlar los cambios en los requisitos del sistema" [Sommerville, 2005].

Para ello, se necesita mantener los requisitos particulares y los vínculos entre los requisitos que dependan de otros requisitos. De esta forma, se podrán evaluar fácilmente las consecuencias de un posible cambio en uno de ellos y establecer el alcance total del cambio en el sistema.

Secuencia de acciones cuando se produce un cambio en una especificación

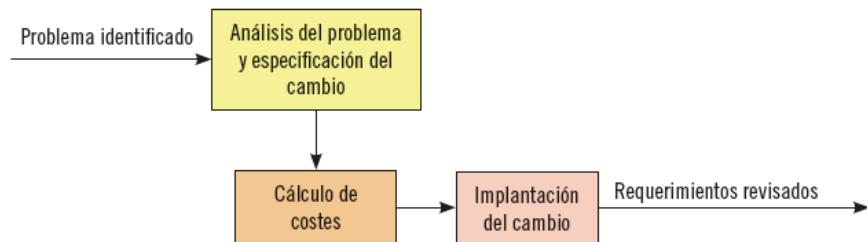


Se puede hacer una clasificación de los requisitos en función de que estos puedan cambiar o no con el tiempo.

Cuando se identifica el cambio, se debe aplicar el protocolo de gestión del cambio de requisitos que se muestra en la imagen inferior.

Utilizar un mecanismo de este tipo permite que los cambios en los requisitos sean tratados de forma consistente y controlada sobre el sistema, minimizando costes en el desarrollo.

Protocolo de gestión del cambio de requisitos



Primero, se debe identificar el problema, analizar la propuesta de cambio y sus dependencias frente a los demás requisitos. Verificar si es válido. Tras esto, se valora el efecto del cambio para analizar su costo. El costo se calculará en función de las modificaciones en el documento de requisitos, y/o en el diseño e implementación del sistema. Tras esta valoración, se determinará si se continúa con el cambio o bien se descarta su implantación.



Actividades

8. La gestión de requisitos ayuda a calcular el costo de un posible cambio en un requisito y sus dependencias con otros requisitos. ¿Quiere decir esto que se podría abordar cualquier tipo de cambio con garantías de éxito en el sistema? Justifique la respuesta.



Aplicación práctica

Imagine una página web con un sistema de autenticación en la que se permita crear mensajes en un foro de discusión. Como técnico responsable de análisis de requisitos, dé una posible descripción del caso de uso “Crear mensaje foro” para el sistema. Se debe poder añadir un título de mensaje y un comentario más detallado. El moderador debe aceptar el comentario antes de que pueda ser publicado para toda la comunidad. Además, podría rechazar el comentario si lo determina no adecuado al hilo de discusión y solicitar al internauta algún dato que no ha sido rellenado correctamente.

SOLUCIÓN

Nombre	Crear mensaje foro.
Descripción	Permite crear un nuevo mensaje (hilo) en el foro de discusión.
Actores	Usuario/Moderador.
Precondiciones	El usuario debe estar autenticado en el sistema.

Flujo normal	<ol style="list-style-type: none">1. El actor pulsa sobre el botón para crear un nuevo mensaje.2. El sistema muestra una caja de texto para introducir el título del mensaje y una zona de mayor tamaño para introducir el cuerpo del mensaje.3. El actor introduce el título del mensaje y el cuerpo del mismo.4. El sistema comprueba la validez de los datos y los almacena.5. El moderador recibe una notificación de que hay un nuevo mensaje.6. El moderador acepta y el sistema publica el mensaje.
Flujo alternativo	<ol style="list-style-type: none">A. El sistema comprueba la validez de los datos, si los datos no son correctos, se avisa al actor de ello permitiéndole que los corrija.B. El moderador rechaza el mensaje, de modo que no es publicado sino devuelto al usuario.
Poscondiciones	El mensaje ha sido almacenado en el sistema y publicado.

3.2. Diseño

El principal objetivo de la etapa de diseño de *software* es determinar la estructura global del sistema, es decir, su arquitectura. Existen tantas definiciones de arquitectura del *software*

como autores que hablan de ella. Se ha seleccionado una definición que contiene todos los puntos comunes de la mayoría de definiciones que se pueden encontrar.

Según Pressman, arquitectura es la estructura jerárquica de los módulos del sistema, la manera de interactuar y comunicarse de los componentes que los forman y las estructuras de datos usadas.

Esta arquitectura servirá como documento para discutir sobre cómo debe resolver el desarrollo *software* los objetivos marcados por la fase de análisis, también para aumentar la precisión en las estimaciones de coste y tiempo, así como ayudar a abordar el entendimiento sobre la complejidad del sistema. Durante el desarrollo, debe servir como vista general del sistema, proporcionar una relación de los componentes que deben ser desarrollados, facilitar el desarrollo simultáneo y permitir detectar errores.

Los diseñadores suelen utilizar lo que se conoce como **patrones arquitectónicos** para elaborar la arquitectura.



Definición

Patrón arquitectónico

Solución a un problema común que aparece en el diseño de la arquitectura y que alguien ha resuelto ya anteriormente.

Modelos para el diseño de sistemas: contexto y arquitectura, procesos, datos, objetos, interfaces de usuario, componentes y despliegues

Los modelos de diseño permiten representar visualmente cómo debe funcionar un determinado aspecto del *software*. Lo mejor de los modelos en esta etapa es que garantizan coherencia en la comunicación entre los módulos que forman parte del sistema y reducen el impacto que supone comprender y modificar el código.

Un modelo se representa mediante un documento que contiene los siguientes elementos:

- Nombre del modelo.
- Descripción del contexto para el que es válido el modelo.
- Breve explicación del aspecto del *software* y del problema que resuelve.
- Elementos principales y relaciones entre ellos. Estos elementos podrían ser: clases, componentes, interfaces, etc., dependiendo del tipo de modelo.
- Interacciones entre los elementos.
- Explicación de la nomenclatura utilizada.
- Descripción de cómo el modelo resuelve el problema.
- Descripción de variantes que puede elaborar el desarrollador.

Existen muchos tipos de modelos, pero básicamente cada uno puede pertenecer a uno de los siguientes tipos de diseño:

- Diseño procedural o a nivel de componentes.
- Diseño de interfaz.
- Diseño arquitectónico.
- Diseño de los datos.



El **diseño a nivel de componentes (diseño procedimental)** se encarga de transformar los elementos estructurales del diseño arquitectónico en la descripción procedural de los componentes, es decir, diseñar los algoritmos e interfaces de estos componentes.

El **diseño de la interfaz** consiste en elaborar y diseñar el conjunto de interfaces de comunicación entre la máquina y el usuario para el uso del sistema. Debe ser capaz de mostrar la información al usuario y permitir que este la procese.

Hay que tener en cuenta aspectos como la complejidad del sistema, la sobrecarga de información y el grado de control que se quiere que el usuario posea sobre el sistema.

El **diseño arquitectónico** sería equivalente al diseño del plano de una casa. En él se podrá observar la distribución de los componentes que forman parte del sistema y cuáles serían las relaciones entre estos componentes que permitan la comunicación.

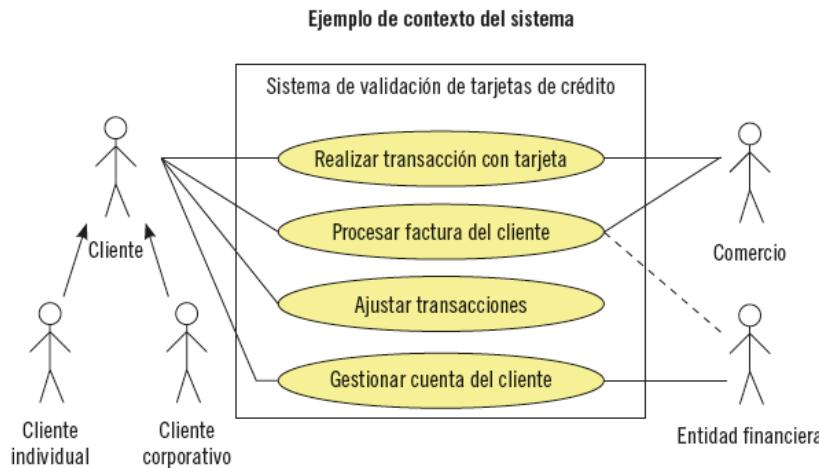
El **diseño de datos** intenta transformar el dominio del sistema elaborado en la fase de análisis en las estructuras de datos necesarias para la fase de implementación del sistema.

Se debe tener en cuenta que existe una gran dependencia directa entre las estructuras de datos escogidas y la complejidad de los procedimientos que se deben utilizar para manejar estas estructuras, así como sobre la calidad de la modularidad y la estructura del sistema final.

También debe encargarse de determinar qué información se ocultará y cuál se mostrará al usuario.

El **modelo de objetos** captura la estructura estática del sistema. Se trata de una representación visual de los objetos del sistema y sus relaciones para llevar a cabo un escenario determinado. Un modelo de objetos es una particularización de un momento del sistema, donde los objetos intervienen como entidades lógicas para llevar a cabo la funcionalidad del sistema.

Establecer el **contexto** en la fase de análisis es importante porque ayuda a identificar las fronteras del sistema, definiendo qué funcionalidad estaría dentro del sistema y cuál es externa. Pero también trata de identificar a los actores que son externos al sistema.



El modelo de procesos o diagrama de flujo es un tipo de modelo que se puede encontrar en gran cantidad de disciplinas. Permite representar el flujo de trabajo o de negocio paso a paso y en un sentido operacional. En *software*, suele capturar la ejecución algorítmica de algún proceso dentro de la funcionalidad que se espera del sistema y ayuda a analizar y discutir sobre ella.

El modelo de despliegue intenta dar una visión estructurada de la distribución de los artefactos del *software*. Un artefacto es un elemento físico resultado del proceso de desarrollo. Algunos ejemplos son ficheros ejecutables, bibliotecas, ficheros de configuración, bases de datos, etc. Los artefactos se encuentran ubicados en nodos computacionales que suelen estar conectados entre sí.

Diagramas de diseño: diagramas de entidad-relación, diagramas de flujo, diagramas de contexto y UML. Diagramas UML de uso común en diseño de sistemas

Como se ha estudiado, los modelos de desarrollo sirven para especificar cómo se desarrolla el sistema. Estos modelos se apoyan en representaciones gráficas, es decir, en diagramas.

Los **diagramas de entidad-relación** se usan para diseñar la base de datos que utilizará el sistema. Son el modelo de datos que representan el esquema de la base de datos en forma de entidades y relaciones entre ellas. Intentan describir una base de datos de forma sencilla y en un lenguaje de alto nivel que sirva para entenderla.

Se elaboran desde una especificación de datos que refleja la lógica de negocio del sistema.

Los principales elementos de estos diagramas son: entidades y asociaciones. Las entidades representan objetos del mundo real con existencia propia. Estas entidades poseen atributos, que son propiedades de las entidades y que sirven para caracterizar a la entidad. Cada atributo puede tomar un conjunto de valores posibles que se conoce como dominio. Por otro lado, las relaciones entre entidades, conocidas como asociaciones, expresan una determinada dependencia conceptual entre las entidades involucradas.



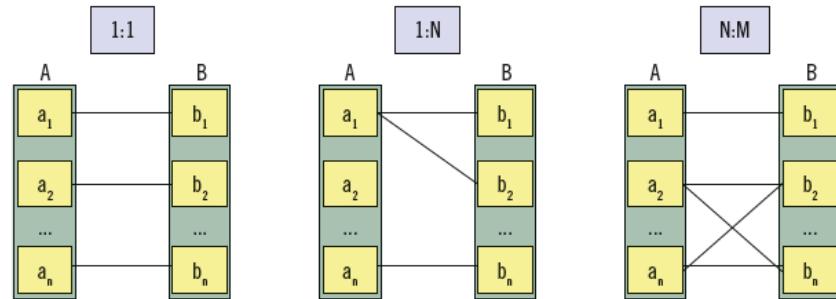
Ejemplo

Las entidades "Empleado" y "Departamento" se encuentran relacionadas a través de la asociación "trabaja en".



Una asociación debe estar también caracterizada por una restricción de cardinalidad, es decir, las relaciones entre las entidades pueden ocurrir en diferentes grados de ocurrencias:

Tipos de cardinalidad para las asociaciones entre entidades

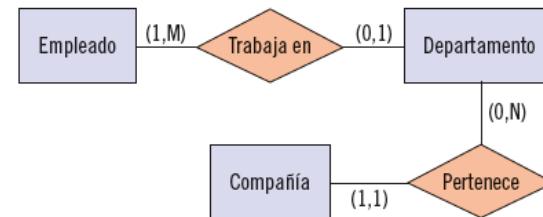


Un empleado trabaja en un departamento, y en un departamento solo trabaja un empleado

Un empleado trabaja en varios departamentos, y en un departamento solo trabaja un empleado

Un empleado trabaja en varios departamentos, y en un departamento trabajan varios empleados

Ejemplo de diagrama entidad-relación



El anterior diagrama se puede leer de la siguiente manera: un empleado puede trabajar o no en algún departamento. En un departamento puede trabajar más de un empleado, pero siempre al menos uno de ellos. Todo departamento pertenece a una compañía. Y una compañía puede estar formada por algún departamento o no.



Actividades

9. ¿Qué finalidad tienen los diagramas de entidad-relación? ¿Es posible utilizarlos para analizar los requisitos funcionales del sistema? Justifique la respuesta.



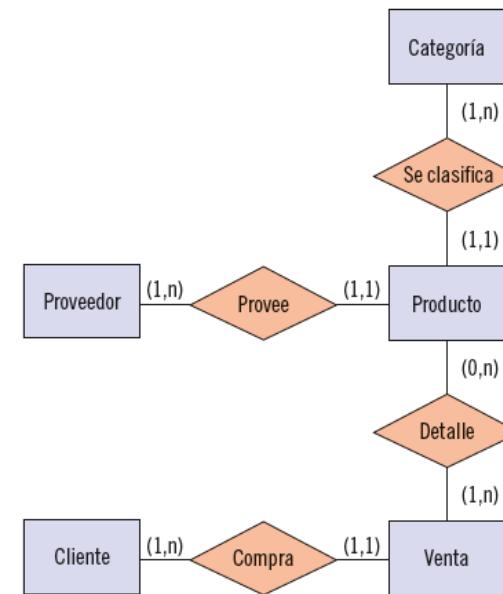
Aplicación práctica

Forma parte del equipo de desarrollo de un proyecto de software y le piden que obtenga un diagrama de entidad-relación del sistema. Para ello, le dan el siguiente documento con las especificaciones.

El sistema debe llevar un control de los proveedores, clientes, productos y ventas, teniendo en cuenta que cada proveedor provee a la empresa con productos y que estos se clasifican en categorías. Cada venta está compuesta de uno o varios de estos artículos y el cliente puede realizar todas las compras que desee.

Dé una posible solución.

SOLUCIÓN

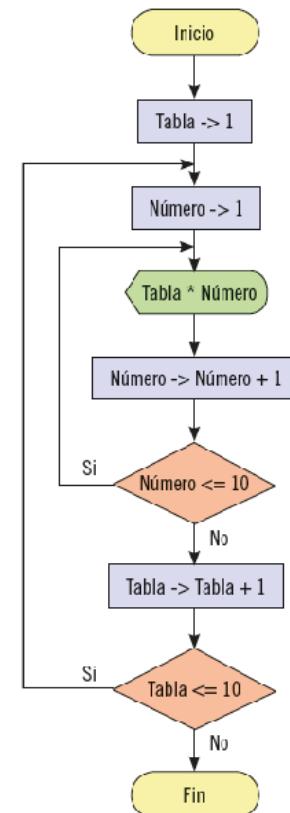


Los **diagramas de flujo** se utilizan para representar gráficamente la secuencia de pasos que se debe realizar para obtener un resultado. En concreto, se suelen utilizar para representar la ejecución de un algoritmo o proceso.



Ejemplo

El diagrama de flujo para un algoritmo que imprime las 10 tablas de multiplicar sería:



Un diagrama de flujo proporciona la secuencia lineal de actividades que se producen en el proceso de forma visual. Actualmente, se utilizan los diagramas de actividades para reemplazar a los diagramas de flujo.



Actividades

10. ¿Por qué son insuficientes los diagramas de flujo cuando se habla del diseño de la arquitectura del sistema?

Los **diagramas de contexto** son casos especiales de diagramas de flujo de datos. En ellos, se representa habitualmente el sistema como una burbuja en el centro del diagrama y a todos los agentes externos alrededor de ella, definiendo entradas y salidas a la burbuja, que representarán los flujos de datos y control.

Ejemplo de diagrama de contexto



Las características que se intentan mostrar con este tipo de diagramas son:

- Los **agentes externos** que se comunican con el sistema de algún modo: personas, organizaciones, otros sistemas, etc.
- Datos que el sistema recibe de otros **agentes externos** y que debe procesar de algún modo.
- Datos que el propio sistema produce y comparte con los **agentes externos**.
- Almacenes de datos que son compartidos por los propios **agentes externos**.
- Representar la **frontera lógica** del sistema, es decir, qué es lo que se va a programar y qué es lo que no se va a programar.

Lenguaje UML. Diagramas UML de uso común en diseño de sistemas

UML (*Unified Modeling Language*) es un lenguaje de modelado utilizado para la especificación, visualización y documentación de esquemas de sistemas de software. No es un método para determinar cómo diseñar el sistema, solamente permite describir visualmente su diseño para poder analizarlo y comprenderlo fácilmente.



Sabía que...

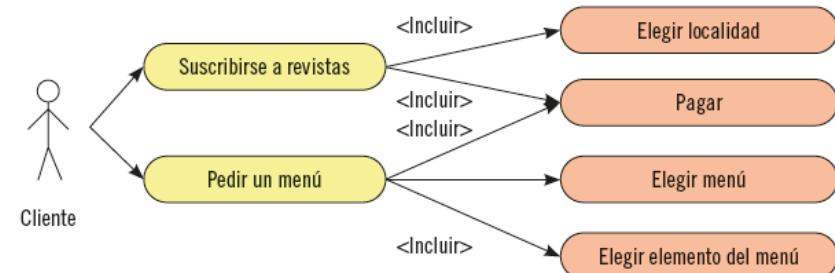
UML se diseñó pensando en la metodología orientada a objetos, aunque se puede usar en sistemas con desarrollo basado en componentes e incluso en programación estructurada.

Se compone de una gran cantidad de elementos para esquematizar y construir diagramas que representen el sistema desde múltiples puntos de vista: estática, dinámica, estructural, arquitectónica, funcional, etc.

Los diagramas útiles más comunes para el desarrollo de sistemas en UML son:

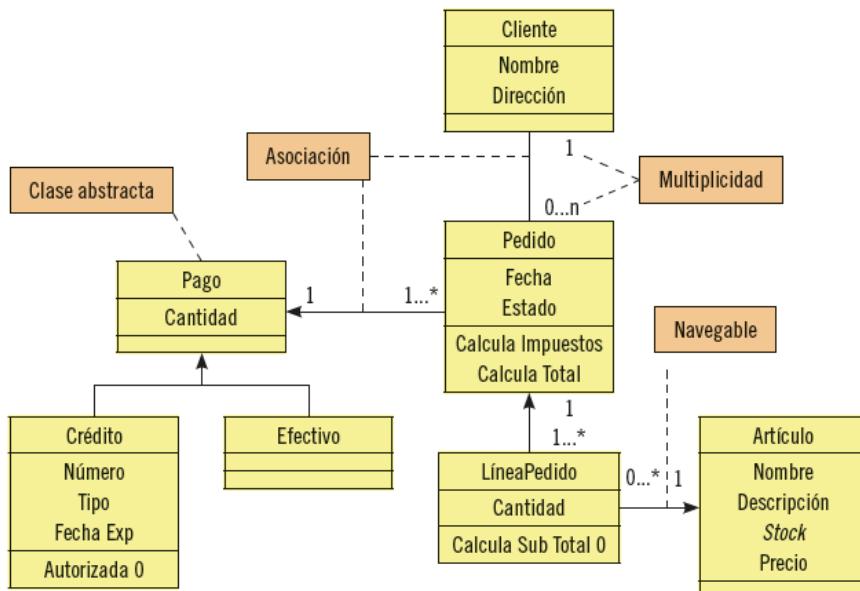
- **Diagramas de casos de uso:** muestran a los actores (otros usuarios del sistema), los casos de uso (las situaciones que se producen cuando utilizan el sistema) y sus relaciones.

Diagrama de casos de uso



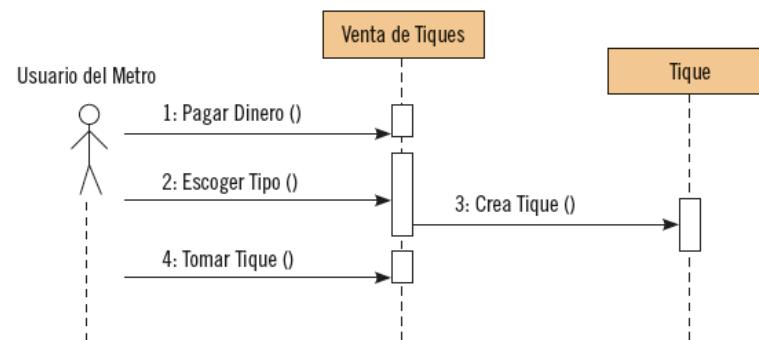
- **Diagramas de clases:** muestran las clases y las relaciones entre ellas.

Ejemplo de diagrama de clases donde aparecen elementos entidad y relaciones entre ellos



• **Diagramas de secuencia:** muestran los objetos y las relaciones entre ellos.

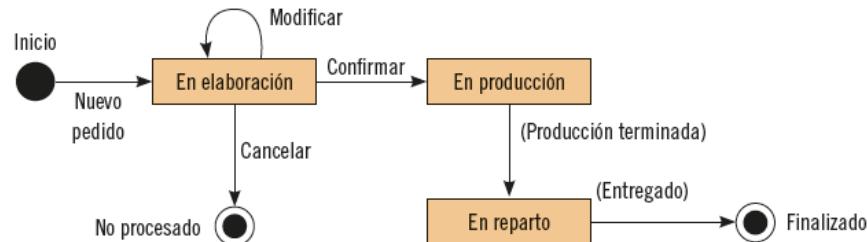
Ejemplo de diagrama de secuencia (especifica el paso de mensajes entre los objetos del modelo de objetos)



▪ **Diagramas de comunicación:** intentan representar la colaboración entre los elementos (objetos) del sistema. Y cómo estos, a través del paso de mensajes, definen un comportamiento del sistema.

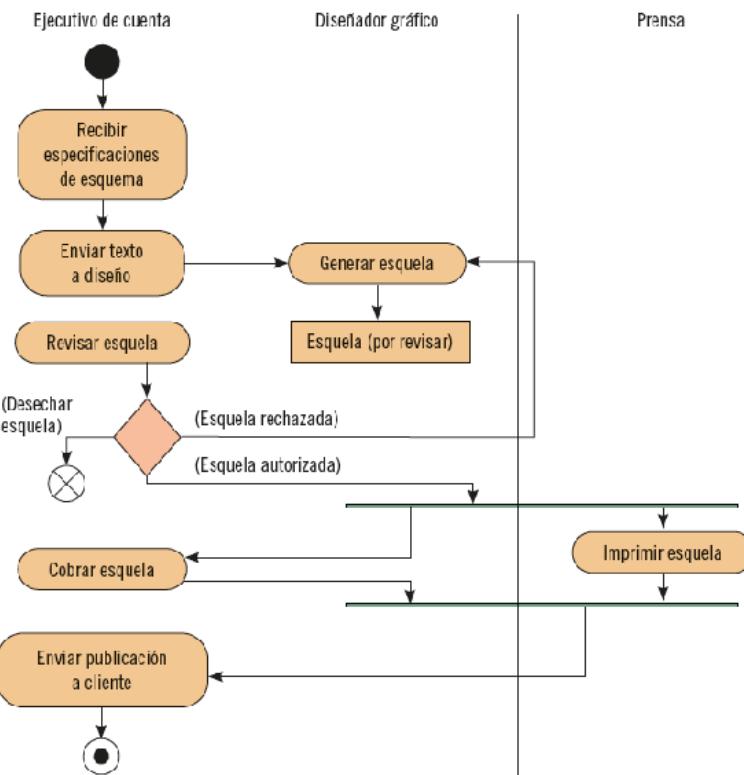
▪ **Diagramas de estado:** se utilizan cuando el sistema se apoya en una máquina de estados para definir los procesos. En estos casos, los cambios de estado se realizan por acciones o eventos que se producen en el sistema de forma manual o automática.

Ejemplo de diagrama de estado



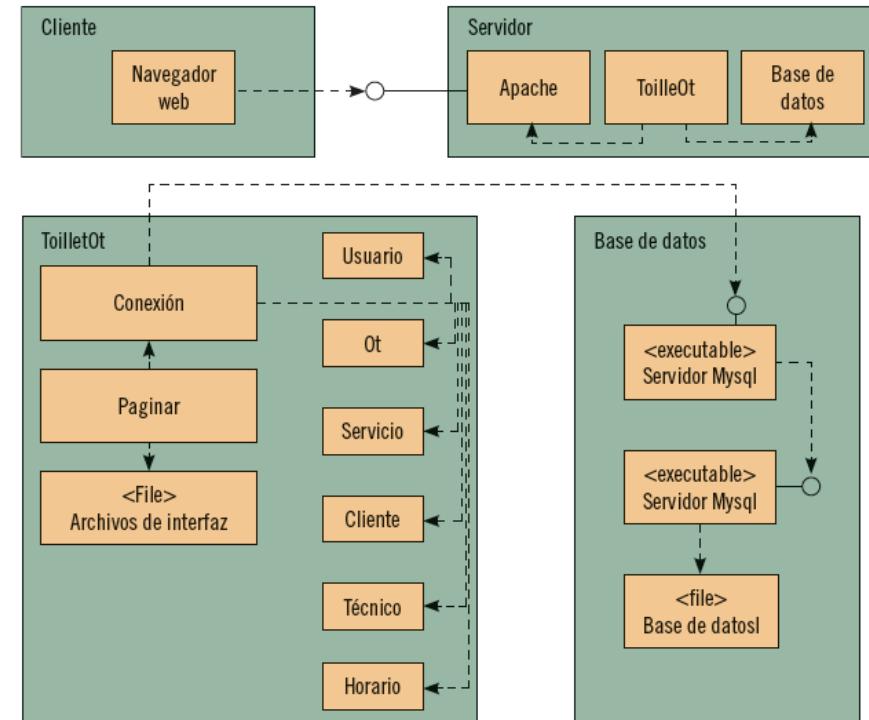
• Diagramas de actividad: se usan para representar el comportamiento del sistema en un escenario específico. Se apoyan en la ejecución de una secuencia de actividades, por lo que el elemento más importante es el flujo de trabajo.

Ejemplo de diagrama de actividad basado en carriles



- **Diagramas de componentes:** se componen de los elementos lógicos que forman el sistema y sus relaciones. Cada elemento o componente es una unidad autónoma dentro del sistema. Los distintos componentes se comunican mediante sus interfaces.

Ejemplo de diagrama de despliegue



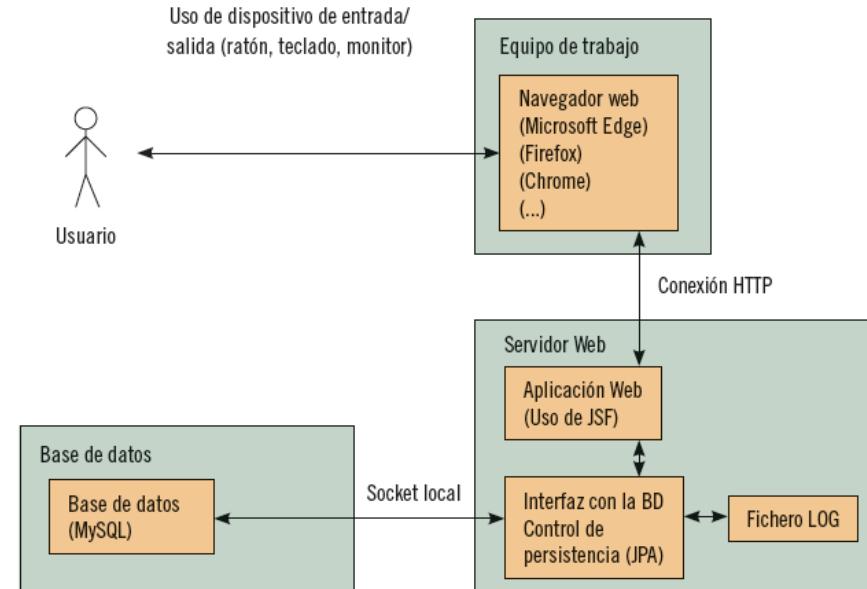
- **Diagramas de implementación o despliegue:** representan los elementos físicos del sistema a nivel *hardware*, como por ejemplo un equipo informático, a nivel *software*, como un fichero ejecutable que realiza alguna acción y artefactos del sistema, como una base de datos.



Actividades

11. ¿Qué tipos de diagramas escogería si necesita describir un sistema basado en estados? Justifique la respuesta.

Ejemplo de diagrama de despliegue formado por tres equipos informáticos



En cada equipo se encuentra un conjunto de subsistemas que colaboran entre sí localmente y remotamente sobre los demás subsistemas.



Aplicación práctica

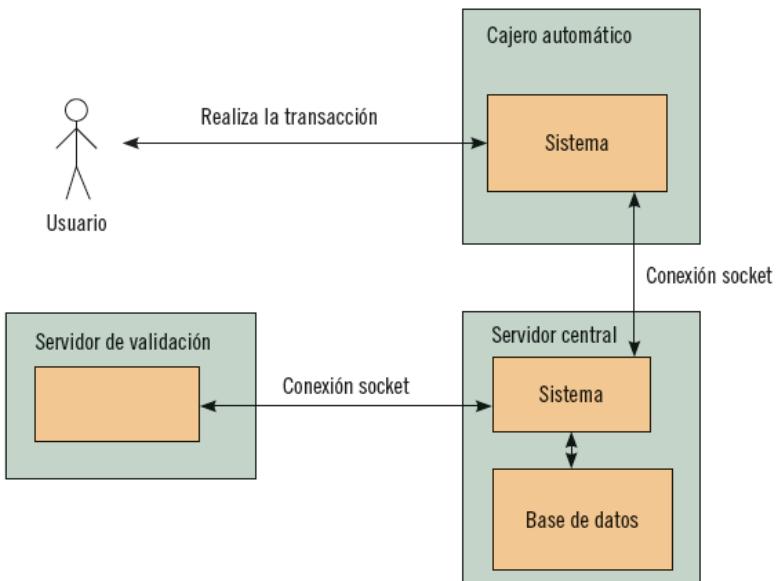
Imagine que está diseñando el sistema de un cajero automático de un banco con las siguientes especificaciones:

El sistema se conectará al servidor central del banco, registrará la transacción que realice el usuario en la base de datos y acto seguido se conectará a otro servidor para validar dicha transacción. Ambas conexiones se realizan a través de un **socket** de comunicación.

Elabore el diagrama de despliegue del sistema y explíquelo.

SOLUCIÓN

Ejemplo de diagrama de despliegue



En el diagrama de despliegue se pueden identificar tres equipos o nodos informáticos. El primero es el cajero automático, donde el *software de sistema* está integrado y opera en función de los requisitos de los clientes del banco. El sistema, a través de una conexión remota, conecta con el subsistema del servidor central, el cual com-

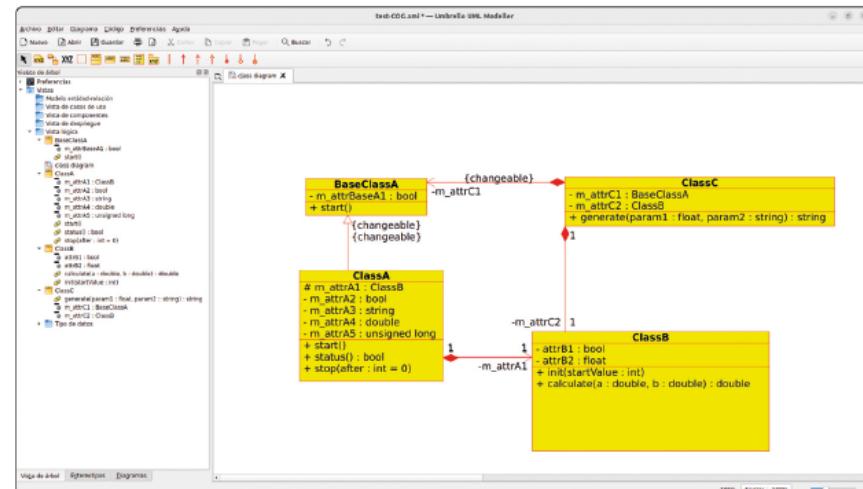
prueba la validez de la operación en otro equipo informático dedicado a esa función y, tras aceptar la validación, realiza una conexión con la base de datos para registrar la operación.

Documentación: herramientas de generación de documentación y documentación de código

La gran cantidad de documentación que debe ser tratada en el desarrollo de un proyecto software provoca que sea necesario el uso de herramientas que permitan una gestión más desatendida y centralizada. Algunas de estas herramientas resultan valiosas a la hora de abordar proyectos de gran envergadura. A continuación, se repasan algunas.

Umbrello

Es una herramienta que permite crear y editar diagramas UML fácilmente. Está basada principalmente en *KDE* (tipo de interfaz gráfica utilizada en *Linux*), pero existen versiones para otras plataformas. Una de las opciones más potentes que posee es la posibilidad de generar código a partir de los diagramas elaborados.

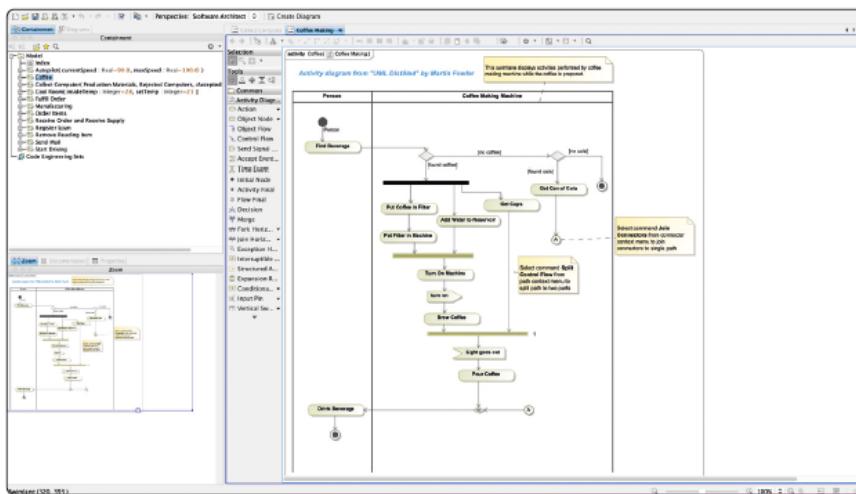


Captura de pantalla de un diagrama de clases elaborado con Umbrello

MagicDraw

Una de las herramientas más potentes que existen para la documentación es *MagicDraw*. Se trata de una herramienta CASE, compatible con *UML 2.3*. Algunos de los entornos de programación más importantes, como: *Sun Java Studio 8*, *NetBeans* y *Eclipse*,

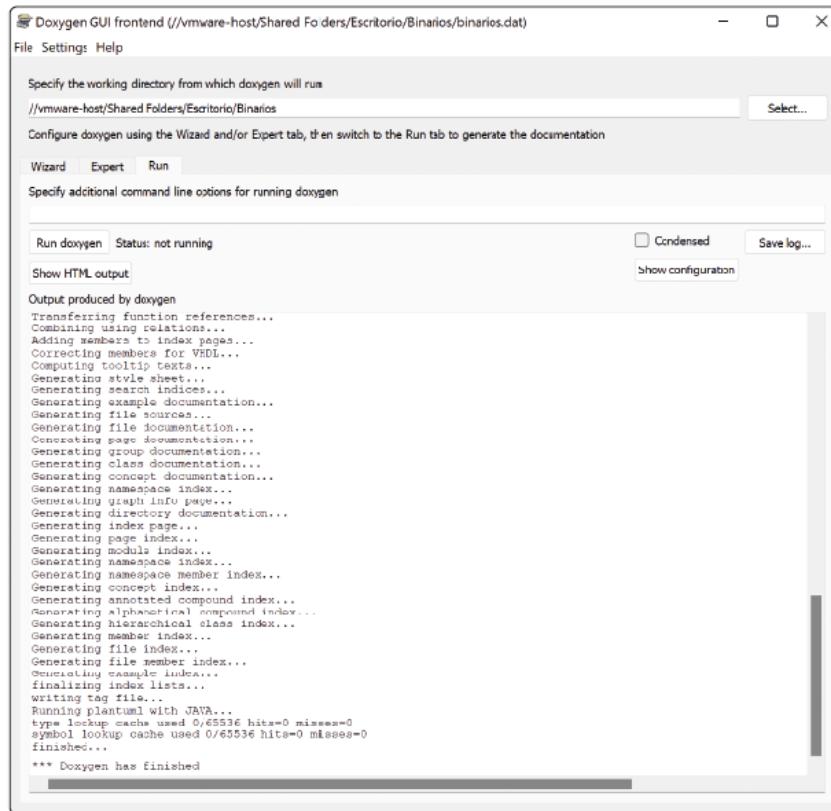
poseen *plugins* para la integración y sincronización del código desarrollado con los esquemas de *MagicDraw*, dando la posibilidad de que se puedan generar los diagramas a partir del código y viceversa (ingeniería inversa).



Captura de pantalla de un diagrama de clases elaborado con la herramienta MagicDraw

Doxxygen

Permite la generación de documentación a partir del código fuente de un sistema. El funcionamiento se basa en recorrer la estructura en árbol del sistema de ficheros que forma el sistema, generando un formato de documento donde se agrupen las clases por biblioteca, y los métodos y funciones por clase, añadiendo la información referente a los comentarios del código. La salida de la documentación puede presentarse en cualquiera de los siguientes formatos: HTML, LaTeX, RTF, PDF, Unix (*man*), XML y HTML (CHM).



Resultado de la documentación de código recopilado por Doxygen en formato HTML

3.3. Implementación. Conceptos generales de desarrollo de software

En la etapa de implementación se aplican los métodos de programación para el desarrollo de los componentes lógicos que conforman la arquitectura del sistema y que han sido identificados en la etapa de diseño. En concreto, si se está en una metodología orientada a objetos, el propósito de esta fase sería obtener las clases que han resultado de la fase de diseño.

Algunas de las tareas involucradas serían:

- Estructurar el modelo de implementación.
- Crear el plan de integración.
- Implementar componentes.
- Validar componentes implementados.
- Integrar subsistemas.
- Validar subsistemas implementados.
- Integrar el sistema.

Los productos que se obtienen en esta etapa son:

- Modelo de implementación: componentes, subsistemas y producto.
- Informe del modelo de implementación.
- Arquitectura del software (modelo de implementación).
- Plan de integración.

A continuación, se dará una explicación más detallada de algunas de las técnicas de programación utilizadas en esta etapa.

Principios básicos del desarrollo de software

La etapa de implementación del *software* guía la tarea de codificación. Pero dicha codificación está muy estrechamente relacionada con los aspectos más intrínsecos de los métodos y lenguajes de programación que se van a utilizar. Aun así, existen principios fundamentales independientemente de las técnicas que se adopten:

- **Principio de preparación.** Es esencial que, antes de escribir una línea de código, se esté seguro de entender el problema que hay que resolver, entender el diseño, escoger un lenguaje que cumpla las necesidades del sistema que se va a desarrollar, seleccionar el ambiente de programación adecuado y crear el conjunto de pruebas que se aplicarán sobre el componente cuando esté desarrollado.
- **Principio de codificación.** Durante el comienzo de la escritura del código se debe estar seguro de que los algoritmos respetan los principios de programación estructurada, que las estructuras de datos definidas satisfacen las necesidades del diseño, crear las interfaces consistentes con el diseño, intentar mantener simple la lógica condicional, elegir convenciones de código que puedan seguir todos los participantes de la codificación y codificar con abundante documentación.
- **Principio de validación.** Tras la codificación, se debe asegurar que se realizan todas las pruebas necesarias y suficientes sobre el componente implementado, se han corregido todos los errores detectados y, si es necesario, se ha producido la refabricación del código.

Técnicas de desarrollo de software: basadas en prototipos, basadas en componentes, métodos de desarrollo rápido y otras técnicas de desarrollo

Cuando no se han identificado todos los requisitos que debe satisfacer el sistema, no se está seguro de la eficiencia de cierto algoritmo o de la forma de implementar una interfaz de tipo máquina-usuario, se pueden utilizar **técnicas de desarrollo basadas en prototipos**.



Definición

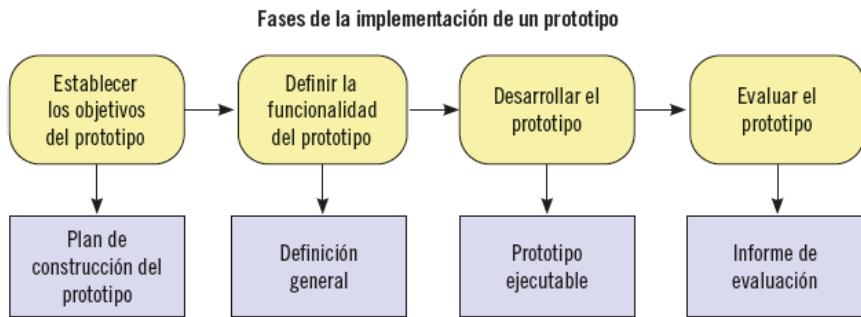
Prototipo

Versión preliminar del sistema que se está desarrollando permitiendo analizar y evaluar los requisitos a los que se ha llegado de acuerdo entre los clientes y los técnicos informáticos.

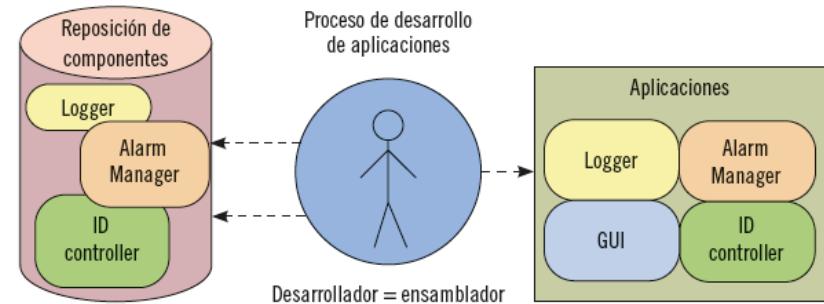
Estas técnicas permiten aplicar cuatro claves importantes en cada fase de desarrollo:

- Identificación de requisitos que debe de cumplir el prototipo.
- Diseñar e implementar el prototipo.
- Utilizar el prototipo con el fin de probar que cumple los requisitos para los que fue diseñado.
- Revisar y mejorar el prototipo.

Finalmente, el prototipo puede formar parte del producto final o ser desecharido.



Método basado en componentes



La **metodología basada en componentes** aplica la técnica de “divide y vencerás” para reducir la complejidad del sistema.



Recuerde

Es un desarrollo basado en componentes, el análisis y el diseño están centrados en obtener componentes software, que son piezas de código que encapsulan cierta funcionalidad, poniéndola a disposición a través de una interfaz.

Este modelo de desarrollo se basa en la reutilización de los componentes para el desarrollo del sistema, reduciendo los tiempos de desarrollo y mejorando la calidad del producto final.

Los **métodos de desarrollo rápido** reducen el tiempo del ciclo de vida del software desarrollando una versión prototípica para después integrar la funcionalidad de forma iterativa y haciendo cumplir uno o varios de los requisitos del cliente en cada iteración. Las técnicas de desarrollo rápido han evolucionado de los lenguajes de última generación y son utilizadas principalmente para desarrollar aplicaciones que hacen un uso intensivo de datos.

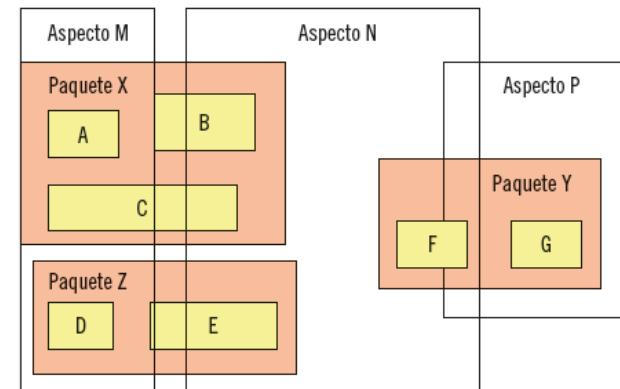
Se componen de un lenguaje de programación de base de datos, con la estructura de la base de datos necesaria para la aplicación y las operaciones básicas de manipulación sobre los datos. Un generador de interfaces, para elaborar la interfaz gráfica que permita mostrar y manipular la información. Enlaces a las aplicaciones de oficina, como módulos para mostrar los datos en formatos específicos de hojas de cálculo y procesadores de texto. Y un generador de informes, que permita mostrar la información de la base de datos en un formato imprimible.

Algunas veces, no es posible, por razones prácticas, llevar a cabo un enfoque incremental del desarrollo del *software*. Incluso el documento de especificaciones es incompleto o confuso. Una posible solución a estos obstáculos es el desarrollo de un prototipo de *software*. Un prototipo es una versión inicial de un sistema *software* que se utiliza para demostrar los mecanismos y las funciones del sistema con el fin de llegar a un entendimiento con el cliente.

Existen otras técnicas de desarrollo, como los **desarrollos basados en aspectos** que intentan mejorar el enfoque orientado a objetos. Se centran en la separación de intereses de un sistema para producir una modularización más adecuada. Para conseguir esto, en la fase de análisis se capturan los casos de uso y se clasifican en requisitos de aplicación y de infraestructura. Los de aplicación describen la funcionalidad básica del sistema y los de infraestructura agregan cualidades de uso como velocidad de ejecución, rendimiento, etc.

Se entiende por **aspecto** a un elemento de código que modifica el código base de un componente para dotarlo en un elemento de diseño independiente del lenguaje.

Desarrollo basado en aspectos



Actividades

12. Elabore una tabla con las principales ventajas e inconvenientes de las técnicas de desarrollo vistas en este apartado.

3.4. Validación, verificación y pruebas

Tras la etapa de implementación, el *software* debe someterse a unos controles que permitan evaluar si se ha alcanzado la calidad y los objetivos que se propusieron en las primeras etapas del desarrollo. Existen dos conjuntos de actividades que permiten aplicar métodos exactos y rigurosos sobre el *software* y sacar conclusiones. Estos dos conjuntos son los que se denominan: verificación y validación.

La **verificación** es el conjunto de actividades que permite evaluar si una función del *software* se ha implementado correctamente. Es decir, se responde a la pregunta: "¿Se está construyendo el producto correctamente?"

Por otro lado, la **validación** es el conjunto de actividades que permiten asegurar que el *software* desarrollado cumple los requisitos del cliente. Esto es, responder a la pregunta: "¿Se está construyendo el producto correcto?"

Auditorías de calidad, revisión técnica periódica, simulación, pruebas de desarrollo y análisis de algoritmos son algunas de las actividades de las que se componen la verificación y validación del *software*.

Las **pruebas del software** son la tarea más pragmática que permite el descubrimiento de errores. Son el último eslabón para determinar y evaluar la calidad del *software*, pero no deben considerarse como la única tarea principal para determinar dicha calidad. Esto debe estar garantizado en gran medida por las etapas de análisis y diseño.

Autores como Miller afirman que "lo que motiva la prueba de los programas es la confirmación de la calidad del *software* con métodos que se puedan aplicar de manera económica y efectiva en sistemas grandes y pequeños".

Validación y verificación de sistemas: planificación, métodos formales de verificación y métodos automatizados de análisis

La validación y verificación del *software* suele ser un proceso muy caro. Es por ello necesaria una planificación donde se puedan realizar las actividades de validación y verificación durante todo el desarrollo del sistema y no únicamente en las etapas finales. También es necesario establecer un equilibrio entre las actividades de índole estática, como el análisis formal del código, y las actividades dinámicas que intentan verificar el *software* a partir de su uso.

Una buena **planificación** debe estar relacionada con estándares para el proceso de pruebas. Debe ayudar a la asignación de recursos y a la estimación del calendario de pruebas y, sobre todo, a los ingenieros implicados en la fase de diseño y en la realización de pruebas para obtener una visión general de todo el proceso y delimitar el trabajo de estos en el contexto del desarrollo del sistema.

Un plan de pruebas debería tener los siguientes apartados:

- **El proceso de prueba:** una descripción de las principales fases del proceso de prueba.
- **Trazabilidad de requerimientos:** es necesario hacer un seguimiento de los requerimientos y dependencias entre ellos cuando se producen las pruebas. De todas formas, es primordial que cada requerimiento se someta a pruebas de forma individual.
- **Elementos probados:** especificación de los elementos que van siendo probados.
- **Calendario de pruebas:** un calendario donde se asignen recursos a los procesos de prueba.
- **Procedimientos de registro de las pruebas:** se debe llevar una documentación de pruebas coherente y consistente. De esta forma, se podrá realizar cualquier tipo de auditoría del proceso de pruebas para comprobar que ha sido satisfactorio y óptimo.

▪ **Requerimientos de hardware y software:** existen herramientas que permiten hacer una gestión más completa de los procesos de prueba.

▪ **Restricciones:** durante el proceso de prueba pueden aparecer eventualidades que afecten negativamente al desarrollo del plan. Esta sección es un intento de anticipación a todas aquellas circunstancias.

Los planes de pruebas deben ser flexibles y evolucionar durante el proceso de desarrollo. Muchas veces, se pueden producir retrasos en otras etapas y es necesario realizar continuas revisiones del plan de pruebas para ajustarlo a las necesidades del proyecto.

Los **métodos formales** son técnicas que utilizan métodos matemáticos para describir las propiedades del sistema y poder verificar a través de análisis matemático que el sistema realiza las operaciones correctas.

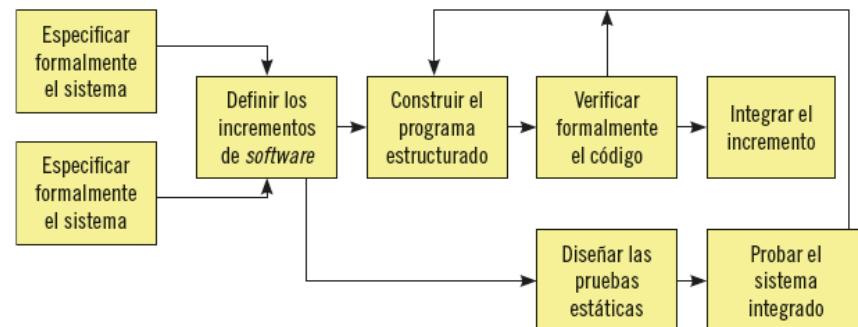


Importante

Para el uso de estos métodos es necesario que la fase de análisis produzca especificaciones descritas mediante lenguajes formales.

Entre los múltiples modelos que usan métodos formales para la verificación y validación, destaca el modelo del proceso de **sala limpia**. Su objetivo es obtener *software* con cero defectos utilizando métodos formales.

Fases de pruebas dentro del desarrollo



Este modelo se basa en cinco puntos clave:

1. **Especificación formal:** el *software* se debe especificar formalmente, utilizando un modelo de transición de estados donde se muestren las respuestas a los diversos tipos de eventos.
2. **Desarrollo incremental:** el *software* se divide en incrementos que se desarrollan y validan de forma independiente utilizando el proceso de sala limpia. Estos incrementos se especifican, con la información de los clientes, en una etapa temprana del proceso. La ingeniería del *software* de sala limpia es un enfoque que defiende la necesidad de corregir el *software* a medida que este se desarrolla.
3. **Programación de estructura:** se utiliza solo un número limitado de estructuras de control y de abstracciones de datos.

4. **Verificación estática:** el software desarrollado se verifica estáticamente utilizando inspecciones de software rigurosas. No existe ningún proceso de prueba de unidades o módulos para los componentes del código.
5. **Pruebas estadísticas del sistema:** el incremento del software integrado es probado estadísticamente, para determinar su fiabilidad. Estas pruebas estadísticas se basan en un perfil operacional, el cual se desarrolla paralelo a la especificación del sistema.

Los **métodos de análisis estático** son un tipo de métodos formales que permiten validar y verificar el software sin necesidad de ejecutarlo. Simplemente, analizando el código fuente del sistema se reconocen anomalías y defectos que pueden derivar en un mal funcionamiento del software o en una semántica errónea respecto a las especificaciones. Existen herramientas de software que escanean el código en busca de estos fallos y, en algunos casos, permiten obtener un flujo de control del programa y calcular el dominio de todos los posibles valores de entrada.

A estos métodos se les conoce como **métodos automatizados** de análisis y su principal objetivo es avisar a los técnicos de pruebas de posibles anomalías en el sistema. A continuación, se muestra un listado de posibles anomalías que se pueden obtener de los métodos de automatización:

Clase de defecto	Comprobación del análisis estático
Defectos de datos	Variables utilizadas antes de su inicialización. Variables declaradas pero nunca utilizadas. Variables asignadas dos veces pero nunca utilizadas entre asignaciones. Posibles violaciones de los límites de los vectores. Variables no declaradas.
Defectos de control	Código no alcanzable. Saltos incondicionales en bucles.
Defectos de entrada/salida	Las variables salen dos veces sin intervenir ninguna asignación.
Defectos de interfaz	Inconsistencias en el tipo de parámetros. Inconsistencias en el número de parámetros. Los resultados de las funciones no se utilizan. Existen funciones y procedimientos a los que no se les llama.
Defectos de gestión de almacenamiento	Punteros sin asignar. Aritmética de punteros.

Sin embargo, estos métodos automatizados no pueden sustituir a otros métodos basados en la revisión informal del código, ya que existen muchos tipos de anomalías y fallos que no son capaces de detectar.



Actividades

13. ¿Qué diferencia existe entre el concepto de validación y verificación del software?



Aplicación práctica

Se ha finalizado la fase de implementación en su proyecto y, a continuación, se debe validar y verificar el programa. Para detectar posibles anomalías, se van a utilizar métodos automatizados de análisis. Elabore un informe con el resultado de las posibles anomalías que encontrarían estos métodos si se aplicasen al siguiente trozo de código:

```
<int binsearch(char *word, char *wordaux, int n)
{
    int cond;
    int low, high, mid;
    int suma;

    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        cond = strcmp(word, wordaux));
        low = low + 1;
    }
    return -1;

    low = low + 1;
}
```

SOLUCIÓN

Clase de defecto	Comprobación del análisis estático
Defectos de datos	Variables utilizadas antes de su inicialización. Variables declaradas pero nunca utilizadas.
Defectos de control	Código no alcanzable.
Defectos de interfaz	Los resultados de las funciones no se utilizan.

Pruebas de software: tipos, diseño de pruebas, ámbito de aplicación, automatización de pruebas, herramientas y estándares sobre pruebas de software

Ya se sabe que una prueba es una actividad en la cual un sistema o alguno de sus componentes se ejecutan dentro de unas condiciones especificadas y se obtienen unos resultados que se registran y se evalúan posteriormente. El caso de prueba sería el contexto (conjunto de entradas y condiciones de ejecución) en el que se realiza dicha prueba.

Es necesario tener en cuenta que no es posible probar exhaustivamente un sistema, porque sería impráctico probar todas las posibilidades de su funcionamiento. El objetivo de las pruebas es detectar los defectos en el software lo que conlleva una mejora en la calidad.

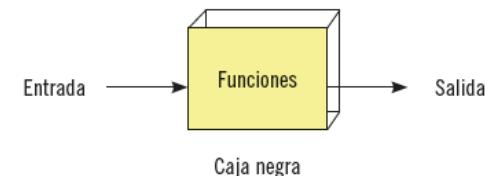
Tipos

Los tipos de pruebas se pueden clasificar en función de qué se conoce, de qué se prueba y del grado de automatización.

Si se atiende a qué se conoce, se pueden identificar dos tipos de pruebas: las pruebas de caja negra y las de caja blanca:

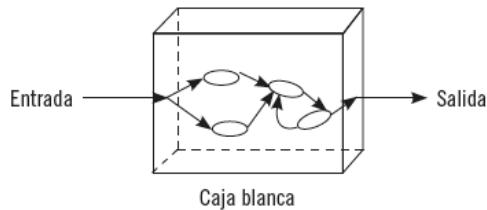
■ Las pruebas de caja negra se centran en las funciones del software. A partir de las entradas, se obtienen las salidas y se contrasta con la respuesta que debería proporcionar el software. En estos casos, no importan los detalles internos del programa, solo se quiere verificar que el programa realiza la función para la que se desarrolló. Las entradas serán escogidas en función de las especificaciones.

Una caja negra indica que no se tiene conocimiento sobre qué sucede en su interior. De esta manera, se ve la prueba en términos de entrada y salida



■ Las pruebas de caja blanca se centran en verificar que el código funciona como está definido. En este tipo de pruebas se utilizan las estructuras de control del diseño procedural.

En la caja blanca, la prueba se centra en el código y en cómo está desarrollado el mecanismo para realizar las pruebas



Si se atiende al grado de automatización, se pueden encontrar pruebas manuales y automáticas.

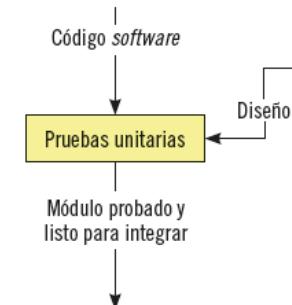
Las pruebas manuales son descripciones de pasos de prueba que define un usuario experto con el fin de especificar las situaciones en las que otro tipo de pruebas no serían nada efectivas (por ejemplo: si hubiera que averiguar el comportamiento de un componente cuando se produce una desconexión de red).

Las pruebas automáticas, por el contrario, son realizadas por un determinado *software* de forma sistemática (por ejemplo: verificar un método de ordenación).

Atendiendo a qué es lo que se prueba, se pueden encontrar pruebas unitarias, de integración, de aceptación, funcionales y de rendimiento. Las pruebas unitarias son pruebas que ejecuta el desarrollador para comprobar si un determinado fragmento del código funciona correctamente (por ejemplo: probar una consulta de base de datos o una función).

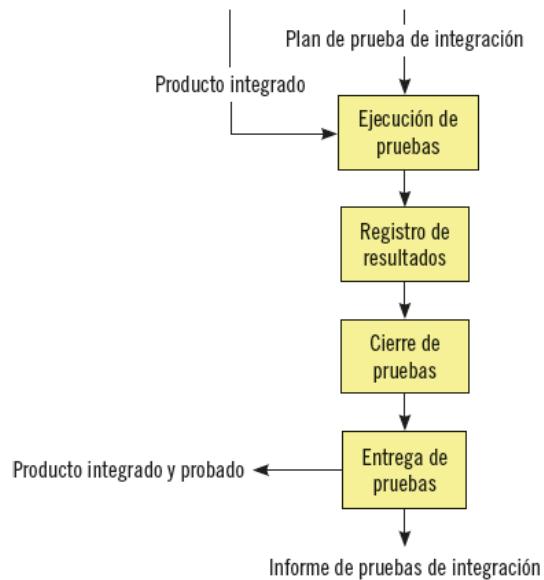
Deberían ser automatizables, es decir, sin intervención manual; repetibles e independientes de otras pruebas.

Flujo de proceso para una prueba unitaria

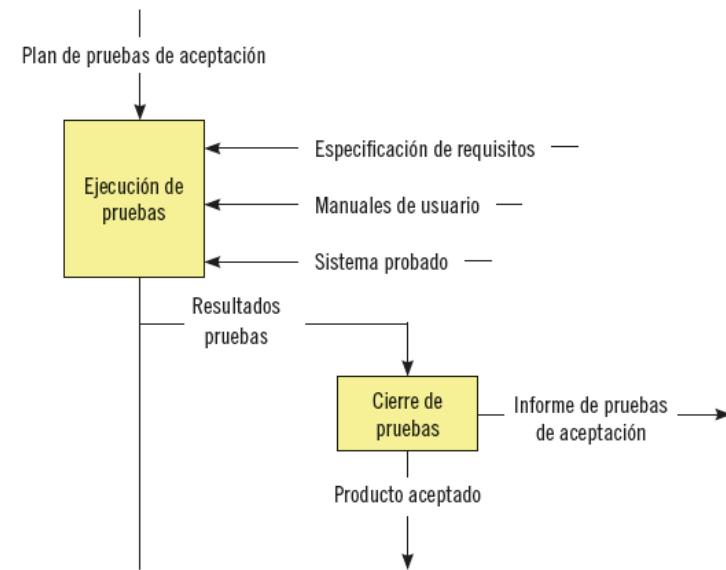


Las pruebas de integración se aplican sobre el sistema una vez integrados todos los componentes.

Flujo de proceso para la integración de pruebas



Flujo de proceso para la aceptación de pruebas



Las pruebas de aceptación corren a cargo del usuario del sistema. Estos usuarios comprueban que el producto es completo y está preparado para su implantación. Las pruebas de aceptación de tipo alfa se llevan a cabo en presencia del equipo de desarrollo; mientras que las de tipo beta las realiza el usuario después de la entrega de la versión casi definitiva del sistema.

Las pruebas funcionales se realizan sobre el sistema final para determinar si cumplen las especificaciones de la fase de análisis.



Actividades

14. ¿Existe alguna incompatibilidad a la hora de aplicar pruebas de caja blanca y caja negra sobre un mismo requisito funcional? Justifique la respuesta.

Las pruebas de rendimiento permiten probar el sistema para determinar la eficiencia final y su capacidad de cómputo.

Diseño de pruebas, ámbito de aplicación, automatización de pruebas

El diseño de pruebas permite definir un entorno o escenario contra el que verificar si un determinado requisito del sistema se cumple o no.

Para ello, se determinan un conjunto de valores de entrada, unas precondiciones y postcondiciones de ejecución, y el conjunto de valores de salida para ese entorno. Acto seguido, se prueba el sistema para esos valores de entrada y se contrasta el resultado obtenido con el que se definió en el diseño de la prueba.

Para diseñar un caso de pruebas, se debe seguir una secuencia de pasos: identificar todos los posibles escenarios para verificar el cumplimiento total de un requisito del sistema, tanto los flujos principales como los alternativos, identificar las condiciones de entrada, definir cuáles de las entradas son válidas y cuáles no, y, por último, realizar el caso de prueba.

El **ámbito de aplicación** de una prueba puede ser de tres tipos:

■ **De módulo único.** En este tipo el **ámbito** se refiere a un módulo o **prueba unitaria** donde se realiza la comprobación sobre una pequeña parte del código incluida en un módulo del sistema.

■ **De grupo de módulos.** En este tipo, las pruebas suelen ser de integración de los módulos del sistema.

■ **De sistema completo.** Suelen ser pruebas que intentan comprobar las funcionalidades generales del sistema e involucran a varios subsistemas.

La **automatización de pruebas** se **puede** considerar como una estrategia con fin de aumentar la calidad del producto con un bajo coste y optimizar la ingeniería de pruebas.

A veces, suele ser la única manera de repetir todas las pruebas en sistemas de gran complejidad y sistemas grandes. Sin embargo, en ocasiones, la inversión para generarlas es incompatible con la estimación de costos del proyecto.

Entre las ventajas, destacan:

- Reducción del esfuerzo de la realización de pruebas.
- Realizar validaciones cuando se producen cambios.
- Habilitar consistencia y cobertura lógicas.
- Herramientas de prueba

Las herramientas de prueba permiten ofrecer grandes beneficios al diseñador:

- Reducen los trabajos repetitivos en el diseño de pruebas.
- Mejoran la consistencia.
- Facilitan las evaluaciones para verificar con objetividad.

Dependiendo de las áreas o actividades a las que se enfocan las pruebas, se pueden distinguir varios tipos de herramientas:

■ **Herramientas de soporte para la gestión de pruebas.** Debería ser la primera en utilizarse debido a que la gestión de pruebas se aplica sobre todo el ciclo de vida del desarrollo del sistema. Dentro de esta categoría, se distinguen cuatro subcategorías:

■ **Herramientas de gestión de pruebas.** Permiten registrar información de las pruebas, organizarlas y monitorizarlas a lo largo del desarrollo.

■ **Herramientas de gestión de requisitos.** Permiten registrar los requisitos y las pruebas asociadas a cada uno. Así, se puede realizar un seguimiento bidireccional entre los requisitos y las pruebas si se produce algún cambio en las especificaciones de requisitos.

■ **Herramientas de gestión de incidencias.** Permiten registrar las incidencias que ocurren a lo largo del desarrollo.

■ **Herramientas de gestión de configuración.** Permiten registrar la información necesaria para la configuración del sistema.

■ **Herramientas de soporte para pruebas estáticas.** Dan soporte a las pruebas estáticas, que son las primeras que se aplican y buscan defectos sin ejecutar el código. Esta categoría abarca los siguientes tipos de herramientas:

■ **Herramientas de revisión de procesos.** Útiles para revisiones formales donde la participación en el desarrollo del sistema esté distribuida geográficamente o bien sea muy amplia.

■ **Herramientas de análisis estático.** Aquí, el código no se ejecuta, sino que se utiliza como entrada a la herramienta. El resultado es un análisis de la complejidad, cálculo de métricas, uso de estándares e identificación de anomalías en el código.

■ **Herramientas de modelado.** Validan modelos de desarrollo y la consistencia de los objetos y suelen ser muy útiles en fases de diseño.

■ **Herramientas de soporte para la especificación de pruebas.** Dan soporte al análisis de pruebas, al diseño de pruebas o a la implementación de pruebas. Existen dos grandes conjuntos:

■ **Herramientas de diseño de pruebas.** Ayudan al diseño de pruebas, permitiendo identificar valores de entradas, requisitos, condiciones de pruebas, etc.

■ **Herramientas de preparación de datos de pruebas.** Ayudan a determinar los datos de pruebas y a gestionarlos, ya que algunas veces las pruebas necesitan un gran volumen de datos.

■ **Herramientas para ejecución y registro de pruebas.** Permiten gestionar la ejecución de las pruebas, para su posterior evaluación. Existen tres grandes grupos:

■ **Herramientas de ejecución.** Utilizan lenguajes de scripting para la captura y registro de la ejecución de las pruebas.

■ **Comparadores de pruebas.** Ayudan a contrastar el resultado obtenido en las pruebas con el funcionamiento correcto del requisito que se esté probando.

■ **Herramientas de seguridad.** Permiten realizar pruebas de seguridad sobre el sistema.

■ **Herramientas de monitorización y rendimiento.** Permiten monitorizar las pruebas durante el proceso de pruebas como de liberación del sistema. Los dos grandes grupos son:

■ **Herramientas de análisis dinámico.** Requieren de la ejecución del código y analizan lo que está ocurriendo durante la ejecución.

■ **Herramientas de monitorización.** Permiten realizar un seguimiento del estado del sistema y así detectar problemas lo antes posible.

Estándares de pruebas

Los estándares de pruebas tienen como principal objetivo definir modelos para los procesos de prueba de software que han sido testados anteriormente con buenos resultados y asegurando que su uso permitirá un aumento de la calidad del software. Algunos de estos estándares son los siguientes:

CMMI

CMMI (*Capability Maturity Model Integration*) es un conjunto de modelos que ayudan a los ingenieros informáticos y sus organizaciones a mejorar los procesos de desarrollo, reflejando una abstracción que permite a dichas organizaciones adoptar prácticas útiles para que se cumplan los objetivos marcados cuando se elaboran proyectos de negocio. Los modelos cubren diferentes situaciones, necesidades y objetivos para lograr esos resultados.

CMMI no posee modelos que permitan mejorar los procesos de pruebas de software. Por ello, surge TMMI (*Test Maturity Model Integration*) como un modelo de procesos para las pruebas de software, que se desarrolló pensando en complementar CMMI.

TMMI

TMMI se basa en niveles de madurez como CMMI. Contempla las actividades dentro del ciclo de vida de las pruebas de software, para la planificación, preparación y evaluación del software. Es el modelo más usado para las pruebas de software. La

principal desventaja es que se trata de un modelo muy reciente y hay poca experiencia de uso al respecto. TMMI se compone de cinco niveles de madurez y 16 áreas de proceso, con metas generales y específicas.

Niveles del modelos TMMI

Nivel 1: Inicial



Nivel 2: Gestionado

Política y estrategia de prueba; planificación de prueba; monitoreo y control de prueba; diseño y ejecución de prueba; ambiente de prueba.

Nivel 3: Definido

Organización dedicada a prueba; entrenamiento para prueba; ciclo de vida de prueba; prueba no funcional; revisión de pares.

Nivel 4: Medido

Medición de prueba; evaluación de la calidad del producto; revisiones avanzadas.

Nivel 5: Optimizado

Prevención de defectos; control de calidad; optimización del proceso de prueba.

ISO 29119

La ISO 29119 es un estándar que intenta unificar algunas de las principales normas para pruebas de software. En definitiva, se centra en definir el vocabulario, los procesos, las técnicas, la documentación y un modelo de evaluación que se pueda utilizar dentro de cualquier ciclo de vida del desarrollo del sistema. Hay hasta 27 países involucrados en el desarrollo de este estándar y se prevé que se convierta en el estándar definitivo para las pruebas de software.

Se compone de 4 partes:

I Parte 1: contiene definiciones y vocabulario para que los técnicos informáticos puedan entenderse a la hora de hablar de los procesos de pruebas, así como los roles y responsabilidades e implicaciones en los diferentes ciclos de vida del software.

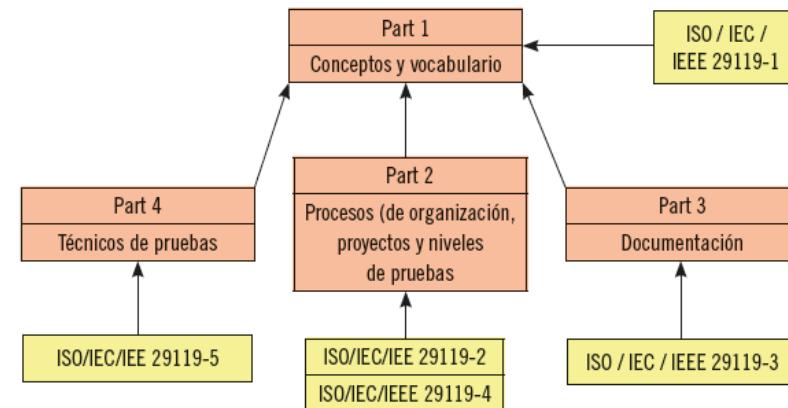
I Parte 2: define un modelo de prueba de proceso genérico para cualquier ciclo de vida y cualquier desarrollo de software. Basado en cuatro capas:

- | Política organizativa del proceso de pruebas.
- | Estrategia de la organización.
- | Gestión del proyecto.
- | Nivel de pruebas.

I Parte 3: cubre la documentación de pruebas para todo el ciclo de vida del software, incluyendo plantillas personalizables para todas las fases del proceso de pruebas.

I Parte 4: cubre numerosas técnicas dinámicas para las pruebas de software y proporciona definiciones informativas de calidad sobre las pruebas.

Relación entre las partes que forman el estándar



Actividades

15. Una organización que utilice un estándar de pruebas de software ¿tiene la garantía de que el sistema alcanzará sus objetivos? Justifique la respuesta.

4. Calidad del software

La garantía de la calidad debe ser siempre una actividad esencial en cualquier negocio cuando este elabore productos de consumo. La necesidad de definir formalmente una función para garantizar la calidad surge a principios del siglo XX y, actualmente, toda compañía que se precie posee mecanismos para garantizarla en sus productos.

En el desarrollo de *software*, el concepto de calidad ha cambiado igual que en otros ámbitos de la ingeniería. Al principio, era responsabilidad exclusiva del programador. Sin embargo, surgen los estándares para la calidad y se extienden muy rápidamente.

4.1. Principios de calidad del software

La calidad del *software* debe estar presente a lo largo de todo el desarrollo del sistema. Sin embargo, cuando es necesario definir qué es calidad del *software* surgen los problemas. Según el *American Heritage Dictionary* se define la calidad como "una característica o atributo de algo". Es decir, algo que se puede comparar, como longitud, color, etc. Por otro lado, es más complicado comparar algo que no es tangible, como es el *software*. Las características intrínsecas de lo que es el *software* hacen muy difícil imaginarse cómo se puede aplicar el concepto de calidad.

No obstante, se puede medir el *software* en términos computacionales. Propiedades como la complejidad ciclomática, la cohesión, las líneas de código e incluso el número de variables utilizadas en promedio por función. Dependiendo del tipo de medidas a las que se esté haciendo referencia, se pueden establecer dos tipos de calidades: calidad del diseño y calidad de concordancia.

La **calidad del diseño** se relaciona con los requisitos, las especificaciones y el diseño del sistema. Por el contrario, la **calidad de concordancia** se relaciona con los aspectos de implementación del sistema.

Por otro lado, en el desarrollo es necesario establecer mecanismos para controlar y monitorizar la calidad del *software*. Estas tareas las desempeña el **control de la calidad**, que involucra inspecciones, revisiones y pruebas que deben llevar a cabo durante todo el proceso de desarrollo para garantizar que se satisfacen los requisitos asignados en la fase de análisis, incluyendo retroalimentación sobre el producto de trabajo. La combinación entre la medición y la retroalimentación permite refinar los procesos cuando los productos de trabajo no alcanzan las especificaciones.

Otro concepto importante es el de **garantía de la calidad**, que está compuesta de un conjunto de funciones que permiten evaluar la efectividad y la complejidad de las actividades de control. Su objetivo es proporcionar los datos necesarios para informar sobre la calidad del producto en cada momento y, de esta manera, saber si el producto final va por buen camino.

Por último, el **coste de la calidad** lo forman todos los costos necesarios para establecer un mecanismo de búsqueda de calidad del producto final. Divididos en tareas de prevención: planificación, revisiones técnicas formales y equipo de pruebas; tareas de evaluación: inspecciones, calibración, mantenimiento del equipo y pruebas; y tareas de fallas: reelaboración, reparación, análisis, y costes cuando el producto final falla en la entrega al cliente.



Actividades

16. ¿En qué etapas del ciclo de vida del software es más común encontrarse con los procesos de calidad del software? ¿Cuál sería el lugar más correcto?

Las métricas proporcionan una forma fiable de estimar la calidad de los atributos internos del producto.

4.2. Métricas y calidad del software

Las métricas de calidad intentan medir el grado en que un sistema, componente o proceso posee un atributo. Es necesario poder medir el software con el fin de suministrar información que ayude a mejorar tanto los procesos como los productos generados.

Además, las métricas permiten analizar y comparar estos atributos para valorar la calidad antes de construir el producto. De esta manera, el tiempo invertido y el esfuerzo del desarrollo será el menor posible. Se conseguirá una manera de controlar el proyecto y alcanzar una planificación más acertada.

Concepto de métrica y su importancia en la medición de la calidad

Es frecuente asociar e incluso confundir los términos calidad, medición y medida, aunque sean muy distintos. Cuando se habla de medición, se hace referencia al proceso por el cual se asignan números o símbolos a propiedades o atributos que representan conceptos o entidades en el mundo real, y que son descritos mediante reglas muy bien definidas. Sin embargo, una medida no es más que una indicación cuantitativa de alguna magnitud medible, como pueden ser cantidad, dimensiones, capacidad o tamaño de alguna propiedad, un proceso o un producto. Pues bien, aclarados estos dos conceptos, se puede definir una métrica como "una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado" [Len O. Ejiogo, 1991].

Principales métricas en las fases del ciclo de vida software

Muchos desarrolladores de software han intentado agrupar en una sola métrica una medida completa y fiable de la complejidad del software. Y aunque hasta ahora se han propuesto un gran número de métricas o medidas, ninguna es capaz de ser lo suficientemente característica para alcanzar la categoría de medida única. Casi todas las métricas proporcionan medidas de diversos puntos de vista del software y de diferentes atributos internos del programa.

Pero eso no implica que las métricas sean algo que no ayude a mejorar y clasificar el software, sino todo lo contrario. Algunos tipos de métricas más utilizados a lo largo de las fases del ciclo de vida del software se detallan a continuación:

- **Métricas de complejidad:** establecen una medida para la complejidad. Entre algunos aspectos que ayudan a medir se encuentran el tamaño del software en líneas de código, el número de anidaciones en iteraciones y el coste computacional.
- **Métricas de calidad:** definen aspectos como la exactitud, la estructura o la modularidad, el grado de reutilización y el acoplamiento entre módulos y componentes.



Importante

- **Métricas de competencia:** se encargan de medir aspectos como la productividad de las actividades de los programadores, la rapidez de codificación y la eficiencia de recursos.
- **Métricas de desempeño:** intentan medir el comportamiento del sistema con respecto al sistema operativo o *hardware* en el que se despliega.
- **Métricas estilizadas:** aspectos como el estilo de código, el tipo de identificación y las convenciones de programación utilizadas durante el desarrollo.
- Algunas **otras métricas** son la portabilidad, la facilidad de localización y la consistencia.

4.3. Estándares para la descripción de los factores de calidad

Algunos modelos que describen los factores de calidad son:

- **Modelo de Mc Call**, de 1977. Describe la calidad elaborando relaciones jerárquicas entre factores de calidad. Dichos factores se concentran en las características operativas, la capacidad de cambio y la adaptabilidad a nuevos entornos, del sistema.
- **Modelo de Furps**, de 1987. Desarrollado por HP (Hewlett-Packard). Basado en factores como funcionalidad, usabilidad, confiabilidad y capacidad de soporte.
- **Modelo de Dromey**, de 1996. Según este modelo, la calidad está determinada por los componentes del sistema (documentos, guías de usuario, diseños y código).

UNE-ISO/IEC 9126-1

La ISO, bajo la norma UNE-ISO/IEC 9126-1, es un estándar internacional para la evaluación de la calidad de productos de software. Fue publicada en 1992 con el nombre de *Information technology-Software product evaluation: Quality characteristics and guidelines for their use*.

Se basa en la idea de que cualquier componente de software puede ser descrito en términos de una o más de seis características básicas: funcionalidad, confiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad, de forma que todas estas características pueden llegar a determinar un valor medible de la calidad del software. A continuación, se explica en profundidad cada una de ellas.

Funcionalidad

Esta etapa de la ISO permite evaluar y calificar el software para determinar si cumple con las necesidades para las cuales fue diseñado. Utiliza cinco atributos:

- **Adecuación:** determina el grado en el que el software cuenta con funciones apropiadas para realizar las tareas que se especificaron.
- **Exactitud:** determina el grado en que los resultados del software son acordes a las necesidades de los stakeholders.
- **Interoperabilidad:** determina el grado de interacción con otros sistemas.
- **Conformidad:** determina el grado en el que el software se adhiere a estándares o regulaciones en leyes.
- **Seguridad:** determina el grado en el que el software previene el acceso no autorizado, ya sea accidental o premeditado.

Confiabilidad

Recopila los atributos que tienen que ver con la capacidad del *software* para mantener el nivel de ejecución durante el periodo establecido. Se apoya en los siguientes conceptos:

- **Nivel de madurez:** medición de la frecuencia de fallos por errores del *software*.
- **Tolerancia a fallos:** mide la habilidad del *software* para mantener la ejecución de la funcionalidad en caso de fallos.
- **Recuperación:** determina el grado en el que el *software* es capaz de recuperar datos que han sido afectados por fallos.

Usabilidad

Determina el esfuerzo del usuario para usar el sistema. Las claves son:

- **Comprensibilidad:** esfuerzo para reconocer las estructuras lógicas del sistema y los conceptos referentes al *software*.
- **Facilidad de aprender:** esfuerzo para aprender a usar la aplicación.
- **Operabilidad:** conceptos que determinan la operación y el control del sistema.

Eficiencia

Intenta determinar la relación entre el funcionamiento del *software* y los recursos utilizados por este. Existen **dos aspectos** a tener en cuenta: el comportamiento **con respecto al tiempo** y el comportamiento **con respecto a los recursos**.

Mantenibilidad

Permite medir cuantitativamente el esfuerzo para modificar el *software*, ya sea por corrección de errores o por añadir funcionalidad. Es determinado por los siguientes conceptos:

- **Capacidad de análisis:** esfuerzo por determinar deficiencias y fallos en el *software*.
- **Capacidad de modificación:** esfuerzo por modificar el *software*, eliminar fallos o adaptar el sistema.
- **Estabilidad:** determina el grado en el que el sistema puede verse comprometido tras realizar modificaciones.
- **Facilidad de pruebas:** determina el esfuerzo para validar el *software* tras una modificación.

Portabilidad

Determina el conjunto de aspectos que se refieren a la posibilidad de que el sistema sea transferido a otro sistema. Los siguientes aspectos son clave:

- **Adaptabilidad:** posibilidad de adaptar sin necesidad de aplicar modificaciones.
- **Facilidad de instalación:** determina el esfuerzo para instalar el sistema.
- **Conformidad:** determina el grado en el que el *software* cumple los estándares o convenciones sobre la portabilidad.
- **Capacidad de reemplazo:** esfuerzo para sustituir un producto por otro con funciones similares.

Otros estándares. Comparativa

A continuación, se citan algunos estándares actuales, destacando como uno de los más importantes y ampliamente utilizado el CMMI (*Capability Maturity Model Integration*).

Estándar	Órgano	Método de aplicación
ISO/IEC 14598-1	ISO	Proporciona un marco de trabajo para evaluar la calidad del software e indica requisitos necesarios para establecer un proceso de evaluación y aplicar los métodos de medición.
ISO/IEC 2504n	ISO	Son una familia de estándares con recomendaciones, requisitos y guías que pueden ser utilizadas por clientes y desarrolladores para evaluar el software.
ISO/IEC 2500n	ISO	Las normas que forman este apartado definen todos los modelos, términos y definiciones comunes.
CMMI	ISO	Conjuntos de prácticas que ayudan a las organizaciones a mejorar sus procesos productivos.

PSP/TSP

ISO

Es un modelo utilizado para aumentar la calidad de los productos de software a través de un equipo disciplinado y autodirigido.



Actividades

17. ¿Cuáles son los aspectos comunes de todos los estándares orientados a la calidad del software?

5. Resumen

El desarrollo de software se guía por mecanismos y técnicas robustas de ingeniería, pero también debe adaptar todas esas técnicas a las condiciones particulares de cada proyecto, ya que la inflexibilidad en la aplicación de un determinado modelo puede ocasionar el fracaso del proyecto.

Sin embargo, se ha visto cómo en todos los modelos se repiten tareas fundamentales para el desarrollo, agrupándose en cuatro grandes etapas: la fase de análisis, que asegura una compresión del sistema y del problema antes de dar una solución; la fase de diseño, que muestra una aproximación teórica de la solución sobre la comprensión del problema; la fase de implementación, que da una solución técnica y precisa, y, finalmente, una fase de mantenimiento, donde se evalúa y mide la funcionalidad y la calidad del producto que finalmente se ha desarrollado.

Además, como en todo producto, hace falta medir la calidad y corregir sus defectos. Para ello, la ingeniería del *software* ha establecido estándares que permitan medir y determinar la calidad del *software* y así comparar de manera objetiva si alcanza los objetivos.