Índice

Introducción

El **Certificado de Profesionalidad** es un instrumento de acreditación, en el ámbito laboral, de las cualificaciones profesionales del Catálogo Nacional de Cualificaciones Profesionales. Son emitidos por el Servicio de Empleo Público Estatal o, en su caso, por las Comunidades Autónomas y tienen validez en todo el territorio nacional.

Poseer un certificado de profesionalidad supone, sin lugar a dudas, incrementar las posibilidades laborales, ya que, al ser un documento oficial, se valora en cualquier proceso de selección que convoquen las Administraciones Públicas y le acredita profesionalmente ante la empresa privada.

El elemento mínimo acreditable es la **Unidad de Competencia** (UC). La suma de las acreditaciones de las unidades de competencia conforma la acreditación de la competencia general. Una Unidad de Competencia es una agrupación de tareas específicas de un profesional. Las distintas unidades de competencia de un certificado de profesionalidad conforman la **Competencia General**, que permiten el ejercicio de una determinada actividad profesional.

Cada **Unidad de Competencia** lleva asociado un **Módulo Formativo** (MF), donde se describe la formación necesaria para adquirir esa Unidad de Competencia, que puede dividirse, a su vez, en **Unidades Formativas** (UF).

Este libro desarrolla las tres unidades formativas que pertenecen al Módulo Formativo **MF0492_3: Programación web en entorno servidor**, asociado a la unidad de competencia **UC0492_3: Desarrollar elementos software en el entorno servidor**, del certificado de Profesionalidad **Desarrollo de aplicaciones con tecnologías web (IFC154_3)**.

Los cuatro primeros capítulos del libro se corresponden con la **UF1844: Desarrollo de aplicaciones web en el entorno servidor**. Los siguientes cuatro capítulos desarrollan la **UF1845: Acceso a datos en aplicaciones web del entorno servidor** y, los últimos dos, hacen lo propio con la **UF1846: Desarrollo de aplicaciones web distribuidas.**

Aunque el libro esté enmarcado dentro de los certificados de profesionalidad, la forma en que se ha desarrollado permite que sea accesible a cualquier persona que quiera profundizar en el conocimiento de los sistemas informáticos y su administración.

1 El proceso de desarrollo de software

A lo largo de los años, han evolucionado mucho las formas de producir bienes de mejor calidad en el sector de las manufacturas. Este conocimiento puede extenderse a la construcción de productos software de mejor calidad cuando los profesionales del software entienden las características propias del software.

El software es esencialmente un conjunto de instrucciones (programas) que proporcionan la funcionalidad requerida, los datos relacionados y documentos. Por lo tanto, el software es un elemento lógico y se diferencia del hardware, un elemento físico, en sus características. El software se desarrolla, no se fabrica en el sentido clásico. Aunque existen similitudes entre el desarrollo del software y la construcción del hardware, ambas actividades son fundamentalmente distintas. Cada producto software es diferente porque se construye para cumplir los requisitos únicos de un cliente. Cada software necesita, por lo tanto, ser construido usando un enfoque de ingeniería.

Construir un producto software implica entender qué es necesario, diseñar el producto para que cumpla los requisitos, implementar el diseño usando un lenguaje de programación y comprobar que el producto cumple con los requisitos. Todas estas actividades se llevan a cabo mediante la ejecución de un proyecto software y requiere un equipo trabajando de una forma coordinada, siguiendo un mismo proceso.

La industria del software tiene procesos bien definidos para el desarrollo de software; en este capítulo, veremos los modelos más utilizados y las actividades que se deben realizar en cada uno de ellos.

1.1 MODELOS DEL CICLO DE VIDA DEL SOFTWARE

Los modelos de ciclo de vida definen el estado de las fases por las que pasa un proyecto de desarrollo de software. Es una visión de los hechos que ocurren durante el desarrollo de software e intenta determinar el orden de cada una de las etapas involucradas y los criterios de transición entre dichas etapas.

Los modelos de ciclo de vida proporcionan una metodología común entre el cliente y los desarrolladores, reflejan las etapas de desarrollo involucradas y la documentación requerida, de manera que cada etapa se valide antes de continuar con la siguiente etapa. De esta manera, por una parte se suministra una guía para los ingenieros de software con el fin de ordenar las diversas actividades técnicas en el proyecto, y por otra parte se suministra un marco para la administración del desarrollo y el mantenimiento, en el sentido en que permiten estimar recursos, definir puntos de control intermedios, monitorizar el avance, etc.

En otras palabras, los modelos son estrategias de desarrollo que ayudan a organizar las diferentes etapas y actividades del ciclo de vida del software, ayudando al control y a la coordinación del proyecto.

1.1.1 EN CASCADA

El modelo en cascada es el primer ciclo de vida del software. Fue definido por Winston Royce a finales de 1970. Este es el más básico de todos los modelos. Su visión dice que el desarrollo de software es a través de una secuencia simple de fases.

Es el enfoque metodológico que ordena rigurosamente las etapas del ciclo de vida del software, de forma que el inicio de cada etapa debe esperar a la finalización de la inmediatamente anterior. El modelo en cascada es un proceso de desarrollo secuencial, en el que el desarrollo se ve fluyendo hacia abajo (como una cascada) sobre las fases que componen el ciclo de vida. Cada fase tiene un conjunto de metas bien definidas y ayudan a administrar el progreso y desarrollo, además de proveer un espacio de trabajo detallado de la elaboración del software. Se dice que es un modelo dirigido por documentos, ya que en cada fase se genera todo un conjunto de documentos que ayudan a determinar el progreso del proyecto.

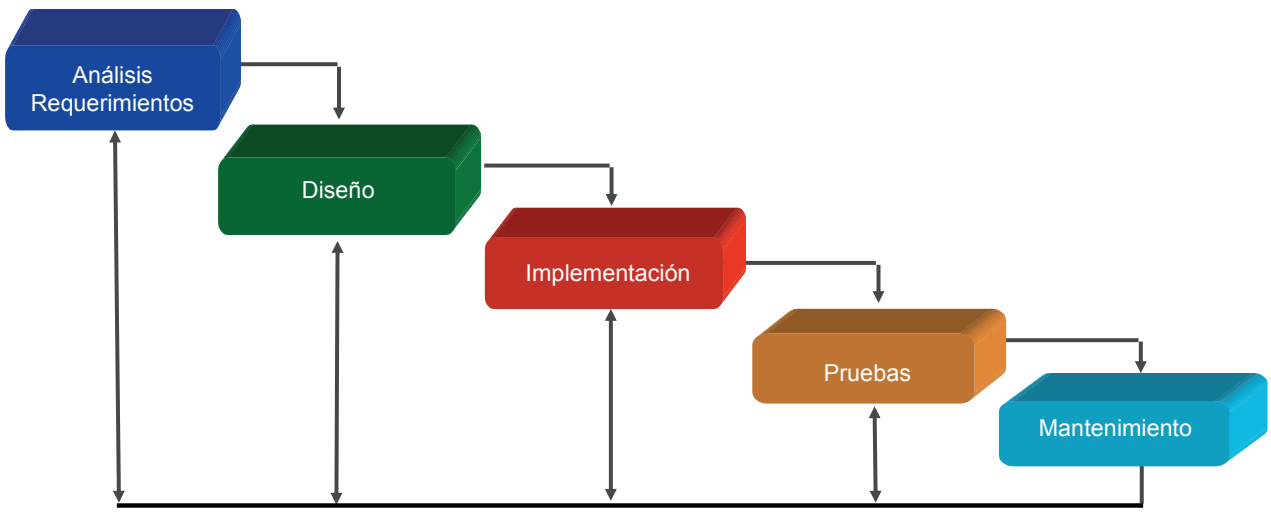

Figura 1.1. *Fases del modelo en cascada*

En la Figura 1.1 vemos las diferentes etapas definidas por el modelo en cascada:

- **Análisis de requerimientos:** Es la primera fase propuesta por el modelo en cascada, en ella se obtiene una visión profunda del problema desde el punto de vista de los desarrolladores y el usuario y se especifica la información sobre la cual el software se va a desarrollar. Se dice que en el análisis se debe determinar **qué** debe hacer el software.
- **Diseño:** Comienza una vez finalizada la etapa de análisis. Tomando como entrada principal el documento de Especificación de requisitos, desarrollado en la fase anterior, permite describir **cómo** el software va a satisfacer dichos requisitos.
- **Implementación:** Partiendo del diseño realizado en la etapa anterior, en esta fase se codifica el software.
- **Pruebas:** También conocida como fase de validación y verificación. Es la fase donde el software es probado para verificar que es consistente con las definiciones realizadas en las etapas de análisis y diseño.
- **Mantenimiento:** Comienza una vez que el software se ha implantado en el cliente y lo ha comenzado a utilizar. En esta fase se realizan modificaciones producto de errores, adecuaciones, nuevos requerimientos, etc.

1.1.2 ITERATIVO

El modelo iterativo es derivado del ciclo de vida en cascada y nace para reducir el riesgo que surge entre las necesidades del usuario y el producto final por malentendidos durante la etapa de recogida de requisitos. Consiste en la iteración de varios ciclos de vida en cascada. Al final de cada iteración se le entrega al cliente una versión mejorada o con mayores funcionalidades del producto. El cliente es quien después de cada iteración evalúa el producto y lo corrige o propone mejoras. Estas iteraciones se repetirán hasta obtener un producto que satisfaga las necesidades del cliente.

Este modelo se suele utilizar en proyectos en los que los requisitos no están claros por parte del usuario, por lo que se hace necesaria la creación de distintos prototipos para presentarlos y conseguir la conformidad del cliente.

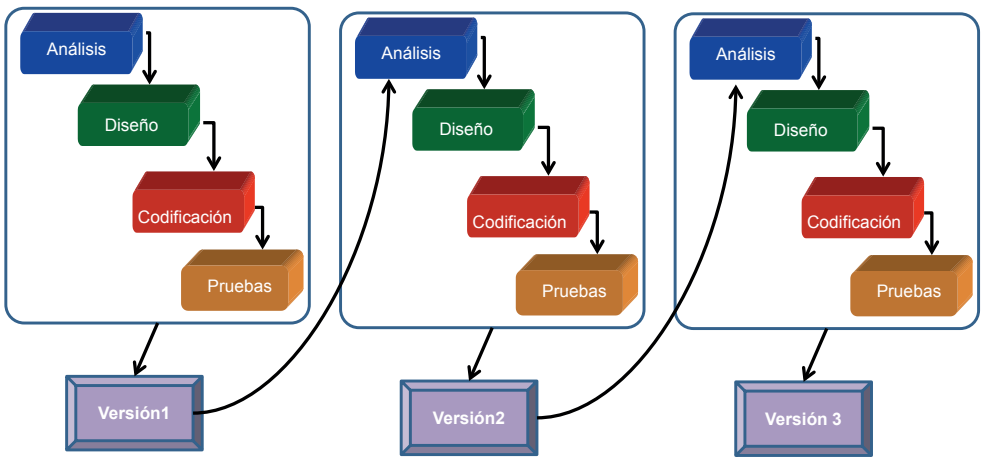

Figura 1.2. *Modelo Iterativo*

1.1.3 INCREMENTAL

El modelo incremental combina elementos del modelo en cascada con la filosofía interactiva de construcción de prototipos. Se basa en la filosofía de la construcción, incrementando a cada paso las funcionalidades del programa. Este modelo aplica secuencias lineales de forma escalonada, mientras progresa el tiempo en el calendario. Cada secuencia lineal produce un incremento del software.

Esta forma de construcción reduce los riesgos en el desarrollo de sistemas largos y complejos, ya que el software se va construyendo por partes. Un sistema pequeño es siempre menos riesgoso que construir un sistema grande, es más fácil determinar si los requerimientos para los niveles subsiguientes son correctos. Reduciendo el tiempo de desarrollo de un sistema decrecen las probabilidades que esos requerimientos de usuarios puedan cambiar durante el desarrollo. Los errores de desarrollo realizados en un incremento, pueden ser arreglados antes del comienzo del próximo incremento.

La principal característica de estos modelos es que permite crear cada vez versiones más completas de software, para esto construimos versiones sucesivas de un producto. Se crea una primera versión que es utilizada por el usuario donde se provee retroalimentación al desarrollador, y según los requerimientos especificados de éste usuario se crea una segunda versión.

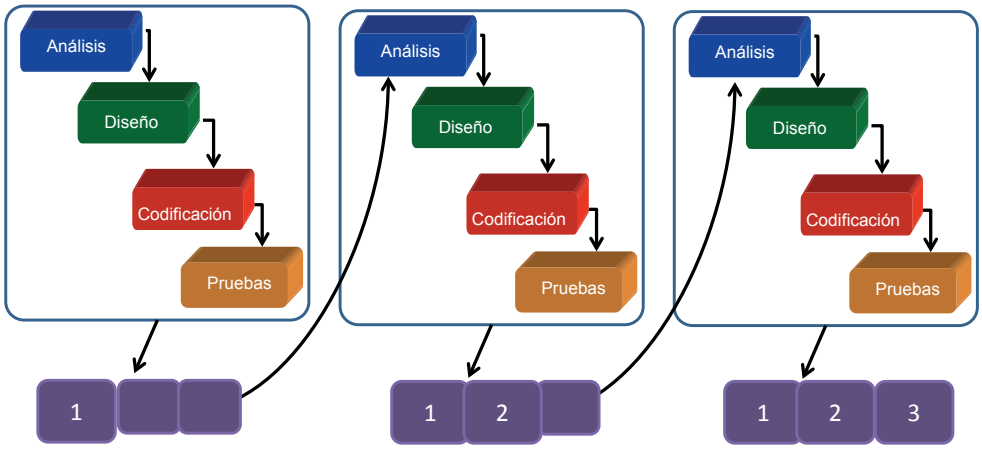

Figura 1.3. *Modelo Incremental*

1.1.4 EN V

El modelo en V nace para terminar con algunos de los problemas que surgen al utilizar el enfoque de cascada tradicional, en particular el hecho de que los errores se encuentren hacía el final proyecto cuando comienzan las pruebas.

El modelo en V propone que las pruebas deben comenzar lo más pronto posible en el ciclo de vida, por lo que define cómo las actividades de prueba (verificación y validación) se pueden integrar en cada fase del ciclo de vida. Dentro del modelo en V, las pruebas de validación tienen lugar especialmente durante las etapas tempranas, por ejemplo, revisando los requisitos de usuario y después, por ejemplo, durante las pruebas de aceptación de usuario. El modelo en V es un proceso que representa la secuencia de pasos en el desarrollo del ciclo de vida de un proyecto. Describe las actividades y resultados que han de ser producidos durante el desarrollo del producto. La parte izquierda de la V representa la descomposición de los requisitos y la creación de las especificaciones del sistema. El lado derecho de la V representa la integración de partes y su verificación. V significa "Validación y Verificación".

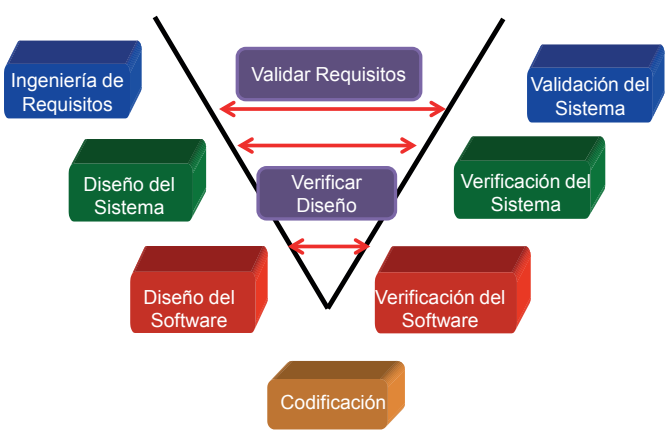

Figura 1.4. *Modelo en V*

Realmente las etapas individuales del proceso pueden ser casi las mismas que las del modelo en cascada. Sin embargo hay una gran diferencia. Sin embargo, hay una gran diferencia: en vez de ir hacia abajo de una forma lineal, las fases del proyecto vuelven hacia arriba tras la fase de codificación formando una V. La razón de esto es que para cada una de las fases de diseño se ha encontrado que hay un homólogo en las fases de pruebas que se correlacionan.

1.1.5 BASADO EN COMPONENTES (CBSE)

En este modelo, se construye el software con la ayuda de productos prefabricados de software, que reciben el nombre de componentes y generalmente son comerciales.

Cada componente tiene una funcionalidad y se comunica a través de una interfaz con entradas y salidas claramente definidas. Incorpora etapas nuevas ya que hay que analizar cuidadosamente la aplicación y estudiar los posibles componentes a incorporar, aparte de diseñar la arquitectura que permita la integración de los mismos.

1.1.6 DESARROLLO RÁPIDO (RAD)

La metodología de desarrollo rápido de aplicaciones (RAD) se desarrolló para responder a la necesidad de entregar sistemas muy rápidamente, sin embargo, no es apropiado para todos los proyectos. El alcance, el tamaño y las circunstancias, todo ello determina el éxito de un enfoque RAD. Es un proceso de desarrollo de software, desarrollado inicialmente por James Martin en 1980. La metodología implicaba desarrollo iterativo y la construcción de prototipos. El desarrollo rápido de aplicaciones fue la respuesta a unos procesos de desarrollo poco ágiles puestos en marcha en los 70 y 80, tales como el método de análisis y diseño de sistemas estructurados y otros modelos en cascada.

Es una fusión de varias técnicas estructuradas, especialmente la ingeniería de información orientada a datos con técnicas de prototipos para acelerar el desarrollo de sistemas software. RAD requiere el uso interactivo de técnicas estructuradas y prototipos para definir los requisitos de usuario y diseñar el sistema final. Usando técnicas estructuradas, el desarrollador primero construye modelos de datos y modelos de procesos de negocio preliminares de los requisitos. Los prototipos ayudan entonces al analista y a los usuarios a verificar tales requisitos y a refinar formalmente los modelos de datos y procesos. El ciclo de modelos resulta a la larga una combinación de requisitos de negocio y una declaración de diseño técnico para ser usado en la construcción de nuevos sistemas.

Figura 1.5. *Flujo de proceso del RAD*

1.1.7 VENTAJAS E INCONVENIENTES. PAUTAS PARA LA SELECCIÓN DE LA METODOLOGÍA MÁS ADECUADA

Modelo	Ventaja	Inconvenientes	Adecuación
Cascada	Está bien organizado y no se mezclan las fases. Es simple y fácil de usar. Debido a la rigidez del modelo es fácil de gestionar ya que cada fase tiene plazos de entrega específicos y un proceso de revisión. Las fases son procesadas y completadas de una vez.	Rara vez sigue una secuencia lineal, esto crea una mala implementación del modelo, lo cual hace que lo lleve al fracaso. Difícilmente un cliente va a establecer al principio todos los requisitos necesarios. Los resultados y/o mejoras no son visibles progresivamente, el producto solo se ve cuando ya está finalizado.	Proyectos estables (con requisitos no cambiantes). Funciona bien para proyectos pequeños donde los requisitos están bien entendidos.
Iterativo	No hace falta que los requisitos estén totalmente definidos al inicio del desarrollo, sino que se pueden ir refinando en cada una de las iteraciones. Permite gestionar mejor los riesgos, gestionar mejor las entregas.	El no ser necesario tener los requisitos definidos desde el principio, puede verse también como un inconveniente ya que pueden surgir problemas relacionados con la arquitectura.	Proyectos con requisitos no definidos o muy cambiantes.
Incremental	Se genera software operativo de forma rápida y en etapas tempranas del ciclo de vida del software. Es más flexible, por lo que se reduce el coste en el cambio de alcance y requisitos. Es más fácil probar y depurar en una iteración más pequeña. Es más fácil gestionar riesgos. Cada iteración es un hito gestionado fácilmente.	Se requiere una experiencia importante para definir los incrementos y distribuir en ellos las tareas de forma proporcionada. Cada fase de una iteración es rígida y no se superponen con otras. Pueden surgir problemas referidos a la arquitectura del sistema porque no todos los requisitos se han reunido, ya que se supone que todos ellos se han definido al inicio.	Proyectos con requisitos no definidos o muy cambiantes.
En V	Es un modelo simple y fácil de utilizar. En cada una de las fases hay plazos de entrega específicos. Tiene una alta probabilidad de éxito sobre el modelo en cascada debido al desarrollo de planes de prueba en etapas tempranas del ciclo de vida. Es un modelo que suele funcionar bien para proyectos pequeños donde los requisitos son entendidos fácilmente.	Es un modelo muy rígido, como el modelo en cascada. Tiene poca flexibilidad y ajustar el alcance es difícil y caro. El software se desarrolla durante la fase de implementación, por lo que no se producen prototipos del software. El modelo no proporciona caminos claros para problemas encontrados durante las fases de pruebas	Proyectos estables (con requisitos no cambiantes). Funciona bien para proyectos pequeños donde los requisitos están bien entendidos.

CBSE	Reutilización de software Reducción del tiempo de desarrollo	Depende de las limitaciones de componentes. Los componentes pueden ser caros y pueden quedar obsoletos. Hay que realizar pruebas exhaustivas.	Equipos de desarrollo con una amplia biblioteca de componentes.
RAD	Velocidad de desarrollo. El RAD aumenta la calidad con la implicación del usuario en las etapas del análisis y del diseño. Visibilidad temprana debido al uso de técnicas de "prototipado". Mayor flexibilidad que otros modelos. Ciclos de desarrollo más cortos.	Características reducidas. Escalabilidad reducida. Más difícil de evaluar el progreso porque no hay hitos clásicos.	Proyectos con poco tiempo de desarrollo.

1.2 ANÁLISIS Y ESPECIFICACIÓN DE REQUISITOS

Un requisito es una "condición o capacidad que necesita el usuario para resolver un problema o conseguir un objetivo determinado". Extraer los requisitos de un producto software es la primera etapa para crearlo. El resultado del análisis de requisitos con el cliente se plasma en el documento Especificación de Requisitos. La captura, análisis y especificación de requisitos (incluso pruebas de ellos), es una parte crucial; de esta etapa depende en gran medida el logro de los objetivos finales. Se han ideado modelos y diversos procesos de trabajo para estos fines. Aunque aún no está formalizada, se habla de la Ingeniería de Requisitos.

La IEEE Std. 830-1998 normaliza la creación de las especificaciones de requisitos software. Definiendo como especificación a la tarea de escribir detalladamente el software a ser desarrollado, en una forma matemáticamente rigurosa. En la realidad, la mayoría de las buenas especificaciones han sido escritas para entender y afinar aplicaciones que ya estaban desarrolladas.

1.2.1 TIPOS DE REQUISITOS

Existen diferentes tipos de requisitos:

- **Requisitos de Usuario:** Declaraciones en lenguaje natural y en diversos diagramas de los servicios del sistema y de las restricciones bajo las que debe operar. Normalmente son necesidades que los usuarios expresan verbalmente.
- **Requisitos del Sistema:** Son los componentes que el sistema debe tener para realizar determinadas tareas. Se escriben como contrato entre el cliente y el desarrollador, deben ser una especificación completa y consistente del sistema, y sirven de base a los desarrolladores para diseñar el sistema.

- **Requisitos Funcionales**: Servicios (funciones) que el sistema debe proporcionar. Indican características y restricciones sobre la funcionalidad del software, además son la condición necesaria de un atributo para que cumpla una función determinada, por ejemplo.
- **Requisitos no funcionales**: Restricciones que afectarán al sistema. Son propiedades o cualidades que el producto debe tener. Los Requisitos no funcionales deben establecer restricciones en el producto que está siendo desarrollado, en el proceso de desarrollo y en restricciones específicas que el producto pueda tener. Pueden clasificarse en:
  - Requisitos de rendimiento: son límites al rendimiento y volúmenes de información que el software debe tratar.
  - Requisitos de seguridad: son características de control de acceso al software y copias de seguridad, entre otros relacionados con la seguridad del sistema y la información.
  - Requisitos de frecuencia de tratamiento: son características sobre la frecuencia con que se ejecutan las diferentes funciones del software.

1.2.2 MODELOS PARA EL ANÁLISIS DE REQUISITOS

El análisis de requerimientos es el conjunto de técnicas y procedimientos que nos permiten conocer los elementos necesarios para definir un proyecto de software. Es una tarea de ingeniería del software que permite especificar las características operacionales de éste e indicar la interfaz con otros elementos del sistema y establecer las restricciones que debe cumplir el mismo.

La especificación de requerimientos suministra al técnico y al cliente los medios para valorar el cumplimiento de resultados, procedimientos y datos, una vez que se haya construido. La tarea de análisis de los requerimientos es un proceso de descubrimiento y refinamiento. El cliente y el desarrollador tienen un papel activo en la ingeniería de requerimientos del software. El análisis proporciona una vía para que los clientes y los desarrolladores lleguen a un acuerdo sobre lo que debe hacer el sistema. La especificación, producto de este análisis, proporciona las pautas a seguir por los diseñadores del sistema.

El modelo de análisis debe cumplir tres objetivos primarios:

- Describir lo que requiere el cliente.
- Establecer una base para la creación de un diseño de software.
- Definir un conjunto de requisitos que pueda validarse una vez construido el software.

Existen diferentes técnicas para realizar la recopilación de los requerimientos como, por ejemplo:

- Entrevistas: Son utilizadas para recopilar información de los interesados. Sin embargo, la predisposición y experiencias de la persona entrevistada influirán en la obtención de resultados. Es conveniente la utilización de preguntas abiertas que no sugieran una determinada respuesta.
- Análisis de Documentos: Todo establecimiento de requisitos implica un cierto estudio de los documentos que definen la razón de ser del proyecto, tales como planes de negocio, estudios de mercado, contratos, etc.
- Tormenta de ideas: Es una técnica eficaz porque las ideas más creativas y efectivas son, a menudo, el resultado de la combinación de ideas aparentemente inconexas. Además, esta técnica alienta el pensamiento de ideas inusuales.

1.2.2.1 Tareas de análisis

Entre las tareas del análisis de requisitos, podemos mencionar las siguientes:

- Construir un modelo a partir de los objetivos y requisitos documentados, teniendo en cuenta también el modelo del negocio, el glosario de términos, etc.
- Registrar los problemas identificados durante la construcción del modelo para su posterior resolución.
- Proponer una primera arquitectura del sistema teniendo en cuenta los requisitos no funcionales.

1.2.3 DOCUMENTACIÓN DE REQUISITOS

Después de que los requisitos han sido recolectados, hay que analizarlos en detalle y documentarlos en una especificación de requisitos. El resultado de la especificación de requisitos y de cualquier especificación de requisitos de componentes hardware/software derivado sirve como registro de convenio con el cliente y compromiso con el proveedor. Estas especificaciones son rastreadas utilizando una matriz de trazabilidad de requerimientos y están sujetos a verificación y gestión de cambio a través del ciclo de vida del producto.

Existen dos tipos de documentos de requerimientos: el documento de especificación de requerimientos y el de definición de requerimientos. Ambos cubren exactamente lo mismo, la diferencia la marca el lector, es decir, a quien están dirigidos. El nivel en el que están escritos depende si se trata del cliente o de las personas que van a desarrollar el sistema. Normalmente, la definición de los requerimientos está redactada en lenguaje natural, mientras que la especificación de los requerimientos se redacta de una forma más técnica, por ejemplo, puede definir un requerimiento que se hizo en lenguaje natural, como una serie de ecuaciones, un diagrama de flujo de datos, casos de uso, etc.

1.2.4 VALIDACIÓN DE REQUISITOS

La validación de requerimientos sirve para demostrar que éstos realmente definen el sistema que el cliente desea. Asegura que los requerimientos están completos, son exactos y consistentes. Debe garantizar que lo descrito es lo que el cliente pretende ver en el producto final. Esta validación es importante porque la detección de errores durante el proceso de análisis de requerimientos reduce mucho los costes. Si se detecta un cambio en los requerimientos una vez que el sistema está hecho, los costos son muy altos, ya que significa volver a cambiar el diseño, modificar la implementación del sistema y probarlo nuevamente.

Las verificaciones que deben llevarse a cabo durante el proceso de validación, son las siguientes:

- Verificación de validez. El análisis puede identificar que se requieren funciones adicionales o diferentes a las que pidieron los *stakeholders*.
- Verificación de consistencia. No debe haber restricciones o descripciones contradictorias en el sistema.
- Verificación de completitud. El documento de requerimientos debe incluir requerimientos que definan todas las funciones y restricciones propuestas por el usuario del sistema.
- Verificación de realismo. Asegurar que los requerimientos pueden cumplirse teniendo en cuenta la tecnología existente, el presupuesto y el tiempo disponible.
- Verificabilidad. Para reducir la posibilidad de discusiones con el cliente, los requerimientos del sistema siempre deben redactarse de tal forma que sean verificables. Esto significa que se debe poder escribir un conjunto de pruebas que demuestren que el sistema a entregar cumple cada uno de los requerimientos especificados.

Existen varias técnicas de validación de requerimientos, estas son: revisiones de requerimientos y construcción de prototipos y generación de casos de prueba. La revisión de requerimientos es un proceso manual en el que intervienen tanto el cliente como personal involucrado en el desarrollo del sistema; Esta puede ser formal o informal y tiene el fin de verificar que el documento de requerimientos no presente anomalías ni omisiones. Los requerimientos deben poder probarse, es por esto que debe hacerse una generación de casos de prueba. Si una prueba es difícil o imposible de diseñar, normalmente significa que los requerimientos serán difíciles de implantar y deberían ser considerados nuevamente. En resumen, la validación pretende asegurar que los requerimientos satisfarán las necesidades del cliente.

1.2.5 GESTIÓN DE REQUISITOS

En la práctica, en casi todos los sistemas los requerimientos cambian. Las personas involucradas desarrollan una mejor comprensión de lo que quieren que haga el software; la organización que compra el sistema cambia; se hacen modificaciones a los sistemas de hardware, software y al entorno organizativo. El proceso de organizar y llevar a cabo los cambios en los requerimientos se llama gestión de requerimientos. El objetivo del analista es reconocer los elementos básicos de un sistema tal como lo percibe el usuario/cliente. El analista debe establecer contacto con el equipo técnico y de gestión del usuario/cliente y con la empresa que vaya a desarrollar el software.

Una vez que un sistema se ha instalado, inevitablemente surgen nuevos requerimientos. Es difícil para los usuarios y clientes del sistema anticipar qué efectos tendrá el sistema nuevo en la organización. Cuando los usuarios finales tienen experiencia con un sistema, descubren nuevas necesidades y prioridades. Las personas que pagan por el sistema y los usuarios de este, rara vez son la misma persona. Los clientes del sistema imponen requerimientos debido a las restricciones organizativas y de presupuesto. Éstos pueden estar en conflicto con los requerimientos de los usuarios finales y, después de la entrega, pueden tener que añadirse nuevas características de apoyo al usuario para que el sistema cumpla con sus objetivos.

La gestión de requerimientos es el proceso de comprender y controlar los cambios en los requerimientos del sistema. Es necesario mantenerse al tanto de estos y mantener vínculos entre los requerimientos dependientes de forma que se pueda evaluar el impacto de los cambios. El proceso de gestión de requerimientos debe empezar cuando esté disponible una versión preliminar del documento de requerimientos. Hay que establecer un proceso formal para implantar las propuestas de cambios y planear cómo se va a gestionar los requerimientos que cambian durante el proceso de obtención de requerimientos.

La gestión del cambio en los requerimientos se debe aplicar a todos los cambios propuestos en los requerimientos. Las ventajas de utilizar un proceso formal para gestionar el cambio son que todos los cambios propuestos son tratados de forma consistente y que los cambios en el documento de requerimientos se hacen de forma controlada.

1.3 DISEÑO

El diseño de sistemas se ocupa de desarrollar las directrices propuestas durante el análisis en términos de aquella configuración que tenga más posibilidades de satisfacer los objetivos planteados, tanto desde el punto de vista funcional como del no funcional.

El proceso de diseño de un sistema complejo se suele realizar de forma descendente:

- Diseño de alto nivel (o descomposición del sistema a diseñar en subsistemas menos complejos).
- Diseño e implementación de cada uno de los subsistemas:
  - Especificación consistente y completa del subsistema de acuerdo con los objetivos establecidos en el análisis.
  - Desarrollo según la especificación.
  - Prueba.
  - Integración de todos los subsistemas.
  - Validación del diseño.

Dentro del proceso de diseño de sistemas hay que tener en cuenta los efectos que pueda producir la introducción del nuevo sistema sobre el entorno en el que deba funcionar, adecuando los criterios de diseño a las características del mismo. En otras palabras, el diseño de sistemas es el arte de definir la arquitectura de hardware y software, componentes, módulos y datos de un sistema de cómputo para satisfacer ciertos requerimientos. Es la etapa posterior al análisis de sistemas.

1.3.1 MODELOS PARA EL DISEÑO DE SISTEMAS

Dependiendo del paradigma que se utilice para desarrollar el software, existirán diferentes modelos que se pueden utilizar para hacerlo, entre los que podemos mencionar: el modelo de arquitectura, el de componentes y el de interfaz.

En todos los casos, los objetivos del diseño deberían ser:

- Acercar el modelo de análisis al modelo de implementación.
- Identificar requisitos no funcionales y restricciones en relación a:
  - Lenguajes de programación, reutilización de componentes, sistemas operativos, tecnologías de distribución, concurrencia, bases de datos, interfaces de usuario, gestión de transacciones, etc.
- Descomponer el modelo de análisis en subsistemas que puedan desarrollarse en paralelo.
- Definir la interfaz de cada subsistema.
- Derivar una representación arquitectónica del sistema.

1.3.2 DIAGRAMAS DE DISEÑO. EL ESTÁNDAR UML

El estándar UML (*Unified Modeled Language*) es un lenguaje unificado de modelado que permite capturar información acerca de la estructura estática y el comportamiento dinámico de un sistema e incluye conceptos semánticos, notación y reglas de creación de diferentes tipos de diagramas.

Para dar soporte a la etapa del diseño de un sistema con UML, se propone el uso de dos tipos de diagrama: el diagrama de casos de uso, cuya misión es ayudar a determinar la funcionalidad del sistema desde la perspectiva del usuario, y el diagrama de interacción o diagrama de secuencia, donde se muestran las interacciones entre objetos mediante transferencias de mensajes entre objetos o subsistemas.

1.4 IMPLEMENTACIÓN. CONCEPTOS GENERALES DE DESARROLLO DE SOFTWARE

La implementación es la etapa en la que se comienza a codificar el diseño realizado en la etapa anterior. A continuación, veremos algunos principios básicos a tener en cuenta y las principales técnicas que pueden utilizarse a lo largo de todo el desarrollo.

1.4.1 PRINCIPIOS BÁSICOS DEL DESARROLLO DE SOFTWARE

En general, la elección de principios y técnicas está determinada por las cualidades que se desean para el software. Los principios que se enunciarán están orientados a obtener sistemas confiables y evolutivos.

El desarrollo de software no sólo necesita buenos principios para obtener productos de calidad sino también técnicas, metodologías y herramientas que se apoyen sobre ellos.

- Rigor y formalidad: una aproximación rigurosa produce productos más fiables, permite controlar sus costos e incrementar su fiabilidad. La ventaja de la formalidad sobre el rigor es que la formalidad puede ser la base para la mecanización del proceso.
- Separación de intereses: permite lidiar con aspectos individuales del problema. Primero, se debe intentar separar los temas que no están íntimamente relacionados entre sí.
- Modularidad: un sistema complejo debe ser dividido en partes. No sólo se aplica a los aspectos estructurales, sino a todo el proceso de desarrollo. Se basa en: descomposición, composición y comprensión, cohesión y acoplamiento.
- Abstracción: los modelos que construimos para entender los fenómenos son abstracciones de la realidad. El uso de modelos formales abstractos permite aproximarse mediante refinamientos a la solución final pudiendo demostrar que cada descripción verifica la anterior.
- Anticipación del cambio: el software sufre cambios permanentemente. En las fases iniciales se requiere un esfuerzo especial para anticipar cómo y dónde será probable que se den los cambios. Los cambios probables deben ser aislados en porciones específicas del software.
- Generalidad: la idea es tratar de focalizar la atención en el descubrimiento de un problema más general que puede estar oculto detrás del problema que se intenta resolver. Puede suceder que el problema generalizado no sea más complejo. Siendo más general, la solución será más reutilizable.
- Incrementalidad: este principio puede aplicarse al identificar tempranamente subconjuntos útiles de una aplicación para así obtener rápido *feedback*. Este principio surge después de que la experiencia ha demostrado concluyentemente que los requerimientos del usuario cambian a medida que se desarrolla el producto.

1.4.2 TÉCNICAS DE DESARROLLO DE SOFTWARE

Existen diferentes técnicas para el desarrollo de software, estas técnicas pueden ser agrupadas de acuerdo al propósito de las mismas:

- Técnicas para la recopilación de datos: son las técnicas que se utilizan para determinar los requerimientos. Entre estas podemos mencionar la entrevista, los cuestionarios, la observación *in situ* y el análisis de documentos.

- Técnica de costo-beneficio: es una técnica analítica que enumera y compara el costo neto de una intervención con los beneficios que surgen como consecuencia de aplicar dicha intervención.
- Técnica de planificación y control de proyectos: es la técnica que permite realizar la planificación del proyecto, definiendo los objetivos, el alcance, el calendario del proyecto. Para esto se puede utilizar los diagramas de hitos, el diagrama de Gantt, los diagramas PERT, etc.

1.5 VALIDACIÓN Y VERIFICACIÓN DE SISTEMAS

La Verificación y Validación (V&V) es un conjunto de procedimientos, actividades, técnicas y herramientas que se utilizan, paralelamente al desarrollo de software, para asegurar que un producto software resuelve el problema inicialmente planteado. Actúa sobre los productos intermedios que se generan durante el desarrollo para detectar y corregir cuanto antes sus defectos y las desviaciones respecto al objetivo fijado.

La verificación se refiere al conjunto de actividades que aseguran que el software implementa correctamente una función específica. La técnica más utilizada para hacer verificación es la revisión del SW.

La validación se refiere a un conjunto diferente de actividades que aseguran que el software construido se ajusta a los requisitos del cliente. La técnica más utilizada en la validación son las pruebas SW.

La definición de V&V comprende muchas de las actividades a las que se refiere la garantía de calidad del software (SQA o *Software Quality Assurance*). En particular, la verificación y la validación abarcan una amplia lista de actividades SQA que incluyen revisiones técnicas formales, auditorías de calidad y de configuración, monitorización de rendimientos, simulación, estudios de factibilidad, revisión de la documentación, revisión de la base de datos, análisis algorítmico, pruebas de desarrollo pruebas de validación y pruebas de instalación.

Los objetivos concretos de la V&V son:

- Detectar y corregir los defectos tan pronto como sea posible en el ciclo de vida del software.
- Disminuir los riesgos, las desviaciones sobre los presupuestos y sobre el calendario o programa de tiempos del proyecto.
- Mejorar la calidad y fiabilidad del software.
- Mejorar la visibilidad de la gestión del proceso de desarrollo.
- Valorar rápidamente los cambios propuestos y sus consecuencias.

1.5.1 PLANIFICACIÓN

La visión del desarrollo de software como un conjunto de fases con posibles realimentaciones facilita la V&V. De hecho, al inicio del proyecto es necesario hacer un Plan de V&V del SW (IEEE 1012), cuyas actividades se realizan de forma iterativa durante el desarrollo.

En este plan se debe definir claramente el objetivo de la V&V, los tiempos, recursos, herramientas y técnicas que se utilizarán, las etapas del ciclo de vida en la que se llevará a cabo el proceso de V&V, que informes se emitirán y los procedimientos de control que se llevarán a cabo para asegurar que el proceso se realice correctamente.

1.5.2 MÉTODOS FORMALES DE VERIFICACIÓN

Entre los métodos formales de verificación más utilizados, se encuentran:

- Aserciones E/S: Basado en la lógica de Hoare. El programa se específica mediante aserciones que relacionan las entradas y salidas del programa. Se garantiza que si la entrada actual satisface las restricciones de entrada (pre-condiciones), la salida satisface las restricciones de salida (pos-condiciones).
- Precondición más débil: Básicamente, consiste en dada una pos-condición POST, encontrar, operando hacia atrás, un programa S tal que wp(S, POST) (la pre-condición) se satisfaga en un amplio conjunto de situaciones. Dada una proposición (P) S (Q) donde S es un conjunto de sentencias de un módulo de un programa, y donde P y Q son los predicados que se cumplen antes y después de S respectivamente, se dice que P es la precondición más débil (wp) de S, si es la condición mínima que garantiza que Q es cierto tras la ejecución de S.
- Inducción estructural: La inducción estructural es una técnica de verificación formal que se basa en el principio de inducción matemática. Dado un conjunto S con una serie de propiedades y una proposición P que desea probar la inducción matemática.

1.5.3 MÉTODOS AUTOMATIZADOS DE ANÁLISIS.

Herramientas de software que recorren el código fuente y detectan posibles anomalías y faltas. En general no requieren ejecución del código a analizar, ya que mayoritariamente se realiza un análisis sintáctico con instrucciones bien formadas e inferencias sobre el flujo de control. Estas herramientas complementan al compilador.

1.6 PRUEBAS DE SOFTWARE

Las pruebas de software consisten en la comprobación dinámica del comportamiento de un programa para un conjunto finito de casos de prueba, convenientemente seleccionados entre los, habitualmente infinitos, dominios de ejecución, comparándolo con el comportamiento esperado. En general, las pruebas (*testing*) se realizan para evaluar la calidad de un producto y/o mejorarlo, mediante la identificación de defectos y problemas.

Las pruebas del software son un elemento crítico para la garantía de calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación. La creciente percepción del software como un elemento del sistema y la importancia de los costes asociados a un fallo del propio sistema, están motivando la creación de pruebas minuciosas y bien planificadas.

1.6.1 TIPOS

Cualquier producto de ingeniería puede probarse de una de estas dos formas:

- Conociendo la **función específica** para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa y, al mismo tiempo, buscando errores en cada función.

- Conociendo **el funcionamiento del producto**, se pueden desarrollar pruebas que aseguren que todas las piezas encajan, o sea, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada.

Estos enfoques no son excluyentes entre sí, ya que se pueden combinar para conseguir una detección de defectos más eficaz.

1.6.2 PRUEBAS FUNCIONALES (BBT)

El enfoque funcional o de caja negra consiste en estudiar la especificación de las funciones, la entrada y la salida para derivar los casos. Aquí, la prueba ideal del software consistiría en probar todas las posibles entradas y salidas del programa. Son, por tanto, pruebas que se llevan a cabo sobre la interfaz del software y pretenden demostrar que el software hace lo que se esperaba de él: que la entrada se acepta de forma adecuada y que se produce un resultado correcto, mientras se mantiene la integridad de la información externa. Una prueba de caja negra no contempla la estructura lógica interna del software.

También conocidas como Pruebas de Comportamiento, estas pruebas se basan en la especificación del programa o componente a ser probado para elaborar los casos de prueba. El componente se ve como una "caja negra" cuyo comportamiento solo puede ser determinado estudiando sus entradas y las salidas obtenidas a partir de ellas. Es decir, se centra en los requisitos funcionales. O sea, la prueba de caja negra permite al ingeniero obtener conjuntos de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa.

Lamentablemente, la prueba exhaustiva de caja negra también es generalmente impracticable. Pensemos en el ejemplo de la suma ya mencionado. Un programa que debe sumar dos números del 1 al 100, admite 10.000 entradas. Ya que no podemos ejecutar todas las posibilidades de funcionamiento, es decir, todas las combinaciones de entradas y salidas, debemos buscar criterios que permitan elegir un subconjunto de casos cuya ejecución aporte una cierta confianza en detectar los posibles defectos del software.

Existen distintas técnicas de diseño de casos de caja negra como la "conjetura de errores", donde se enumera una lista de posibles equivocaciones típicas que pueden cometer los desarrolladores y de situaciones propensas a ciertos errores. En los métodos de prueba basados en grafos, a partir de la identificación de los objetos importantes del software, se crea un grafo que permite analizar las diferentes relaciones. Por su parte, la técnica de partición equivalente intenta dividir el dominio de entrada de un programa en un número finito de clases de equivalencia, de tal modo que se pueda asumir razonablemente que una prueba realizada con un valor representativo de cada clase es equivalente a una prueba realizada con cualquier otro valor de dicha clase.

1.6.3 PRUEBAS ESTRUCTURALES (WBT)

El enfoque estructural o de caja blanca consiste en centrarse en la estructura interna (implementación) del programa para elegir los casos de prueba. En este caso, la prueba ideal (exhaustiva) del software consistiría en probar todos los posibles caminos de ejecución, a través de las instrucciones del código, que puedan trazarse. Se basa en un examen minucioso de los detalles procedimentales. Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles.

La principal técnica es la prueba del camino básico que permite al diseñador obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución. Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa.

1.6.4 COMPARATIVA. PAUTAS DE UTILIZACIÓN

La idea principal es que un producto de software pueda probarse en base a su funcionalidad, realizando pruebas que demuestren que ésta es llevada a cabo correctamente, y a su funcionamiento interno, garantizando que las operaciones internas cumplen con las especificaciones. Así, se puede decir que el primer enfoque es una visión externa (prueba de cajas negras) y que el segundo enfoque es una visión interna que conlleva revisar la estructura interna del programa (pruebas de caja blanca).

1.6.5 DISEÑO DE PRUEBAS

Una vez conocidas las técnicas de diseño y ejecución, debemos analizar cómo se plantea la utilización de las pruebas en el ciclo de vida. La estrategia de aplicación y la planificación de las pruebas pretenden integrar el diseño de los casos de prueba en una serie de pasos bien coordinados a través de la creación de distintos niveles de prueba, con diferentes objetivos. Además, permite la coordinación del personal de desarrollo, del departamento de certificación de calidad y del cliente, gracias a la definición de los papeles que deben desempeñar cada uno y la forma de llevarlos a cabo.

La estrategia proporciona un mapa que describe los pasos que hay que llevar a cabo como parte de la prueba, cuándo se deben planificar y realizar esos pasos y cuánto esfuerzo, tiempo y recursos se van a requerir. Por tanto, cualquier estrategia de prueba debe incorporar la planificación, el diseño de casos, la ejecución y la agrupación y evaluación de los datos resultantes.

Una estrategia de prueba del software debe ser suficientemente flexible para promover la creatividad y la adaptabilidad necesarias para adecuar la prueba a todos los grandes sistemas basados en software. Al mismo tiempo, la estrategia debe ser suficientemente rígida para promover un seguimiento razonable de la planificación y la gestión a medida que progresa el proyecto.

1.6.6 ÁMBITOS DE APLICACIÓN

El ámbito de aplicación de las pruebas es el objetivo a probar durante la realización de las mismas. En función del alcance de dichas pruebas, se establecen tres niveles (o ámbitos) de aplicación, partiendo del nivel más específico para llegar al más general:

- Módulo único: pruebas unitarias.
- Grupo de módulos: pruebas de integración o pruebas de componentes.
- Sistema completo: pruebas de sistemas.

1.6.7 PRUEBAS DE SISTEMAS

Este tipo de pruebas tiene como propósito ejercitar profundamente el sistema para verificar que se han integrado adecuadamente todos los elementos hardware y software del propio sistema entre sí y con los elementos software o hardware de otros sistemas con los que deba interactuar. Concretamente se debe comprobar:

- El cumplimiento de los requisitos funcionales establecidos. Para ello, se diseñan casos de prueba aplicando técnicas de caja negra a las especificaciones.
- El funcionamiento y rendimiento de las interfaces hardware, software y de usuario. Para ello se deben diseñar casos de prueba que verifiquen toda la información procedente de otros elementos del sistema.

- La adecuación de la documentación del usuario.
- El rendimiento y la respuesta en condiciones límite y de sobrecarga. Para ello, se diseñarán casos de prueba que comprueben el rendimiento del sistema (pruebas de volumen de datos, de límites de procesamiento, etc.) Este tipo de pruebas suelen llamarse pruebas de sobrecarga (*stress-testing*)

Hay cuatro tipos de pruebas de sistema. Aunque cada una tiene un propósito distinto, todas trabajan para verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas.

- Prueba de recuperación: el objetivo de la prueba de recuperación es comprobar que el sistema es capaz de recuperarse de un fallo grave y seguir funcionando. Para ello, se fuerza el fallo del sistema de diferentes formas y verifica que la recuperación se lleva a cabo de manera apropiada. De esta forma, podremos identificar cuánto tiempo necesita el sistema para volver a su "estado normal" y si efectivamente recupera dicho estado normal.
- Prueba de seguridad: intenta verificar que los mecanismos de protección incorporados en el sistema lo protegerán de cualquier tipo de acceso no deseado. Con tiempo y recursos suficientes, una buena prueba de seguridad terminará por acceder al sistema. El papel del diseñador del sistema es hacer que el coste de la entrada ilegal sea mayor que el valor de la información obtenida.
- Prueba de resistencia: está diseñada para enfrentar a los programas con situaciones anormales. En esencia, el objetivo de la prueba de resistencia es responder a la pregunta: "¿A qué potencia puedo ponerlo a funcionar antes de que falle?". Para ello, la prueba de resistencia ejecuta un sistema de forma que se produzca una demanda de recursos en cantidad, frecuencia o volúmenes anormales.
- Prueba de rendimiento: está diseñada precisamente para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado. En realidad abarca todos los pasos del proceso de prueba, ya que deberíamos ser capaces de asegurar que se satisfacen los requisitos de rendimiento incluso a nivel de módulo. Las pruebas de rendimiento suelen realizarse en paralelo con las de resistencia y se acostumbra a utilizar software y hardware que soporte la medición exacta de la utilización de recursos.

1.6.8 PRUEBAS DE COMPONENTES

Aun cuando los módulos o componentes de un programa funcionen bien por separado es necesario probarlos conjuntamente. Un módulo puede tener un efecto adverso o inadvertido sobre otro módulo. Las sub-funciones, cuando se combinan, pueden no producir la función principal deseada. La imprecisión aceptada individuamente puede crecer hasta niveles inaceptables al combinar los módulos. Los datos pueden perderse o malinterpretarse entre interfaces, etc. Por ejemplo, supongamos que la función B consume un valor producido por la función A. Aun siendo ambos reales, podría ser que al funcionar conjuntamente, B no pudiese procesar la entrada proporcionada por A porque fueran reales de distinta precisión. Por lo tanto, es necesario probar el software ensamblando todos los módulos probados previamente. Ésta es el objetivo de las pruebas de integración.

A menudo hay una tendencia a intentar una integración no creciente, también denominada enfoque *big-bang*. Es decir, se prueba cada módulo por separado y luego se integran todos de una vez y se prueba el programa completo. Obviamente, el resultado puede ser un poco caótico. Se pueden producir un gran número de errores y será prácticamente imposible identificar el módulo (o módulos) que los provocó.

Por contra, se puede aplicar la integración "incremental", donde se combina el siguiente módulo que se debe probar con el conjunto de módulos que ya han sido probados. De esta forma el programa se prueba en pequeñas porciones y los errores y los defectos que los provocan son más fáciles de detectar. Existen dos tipos de integración "incremental", la denominada ascendente y descendente. Veamos a continuación los pasos a seguir en cada caso.

1.6.9 AUTOMATIZACIÓN DE PRUEBAS. HERRAMIENTAS

La automatización de las pruebas consiste en el uso de un software especial que ejecuta pruebas de manera controlada, presentando resultados y comparándolos con los resultados esperados.

Para automatizar el proceso de pruebas, se recurre al uso de *frameworks* tales como JUnit (entornos Java) y Nunit (entornos .Net). Estos entornos forman parte de lo que se conoce como xUnit. JUnit, al trabajar con un lenguaje gratuito como JAVA, es uno de los más utilizados. Permite integrarse en entornos de desarrollo como NetBeans y Eclipse.

1.6.10 ESTÁNDARES SOBRE PRUEBAS DE SOFTWARE

El actual estándar es el ISO/IEC 29119. El objetivo principal es unificar estándares, abarcar completamente el ciclo de vida y ser consistente con otros estándares ISO.

Consta de seis partes:

- Parte 1: conceptos y definiciones, donde se definen los conceptos y definiciones utilizadas en las pruebas de software.
- Parte 2: procesos, que incluyen la evaluación de dichos procesos (ISO/IEC 33063), donde se definen cada uno para ser probados y las estrategias a aplicar en cada uno de ellos.
- Parte 3: documentación que sirve para definir plantillas que pueden ser utilizadas para generar nueva documentación.
- Parte 4: técnicas de pruebas que definen cada una de ellas, las medidas de cobertura, el alcance, etc.
- Parte 5: pruebas basadas en palabras claves que define el conjunto de palabras claves y las diferentes estrategias que se pueden aplicar.
- Parte 6: técnicas de pruebas estáticas.

1.7 CALIDAD DEL SOFTWARE

La calidad es una herramienta básica para una propiedad inherente de cualquier cosa que permite que esta sea comparada con cualquier otra de su misma especie. La palabra "calidad" tiene múltiples significados. De forma básica, se refiere al conjunto de propiedades inherentes a un objeto que le confieren capacidad para satisfacer necesidades implícitas o explícitas. Por otro lado, la calidad de un producto o servicio es la percepción que el cliente tiene del mismo, es una fijación mental del consumidor que asume conformidad con dicho producto o servicio y la capacidad del mismo para satisfacer sus necesidades.

Desde un punto de vista profesional se puede definir a la calidad como la totalidad de las características y aspectos de un producto o servicio en los que se basa su aptitud para satisfacer una necesidad dada (EOQ). O según la ISO 9000:2000, es el grado en el que un conjunto de características inherentes cumple con los requisitos.

1.7.1 PRINCIPIOS DE CALIDAD DEL SOFTWARE

Se pueden establecer unos aspectos generales sobre los que apoyar la calidad del software:

- **Corrección:** capacidad de realizar las actividades requeridas, es decir, la correcta adaptación de los requisitos.
- **Robustez:** capacidad de reacción ante excepciones e imprevistos durante la ejecución.
- **Eficiencia:** capacidad para hacer uso adecuado de los recursos del sistema.
- **Portabilidad:** facilidad de migración entre diferentes plataformas.
- **Integridad:** capacidad del sistema para proteger sus componentes contra accesos no permitidos.
- **Facilidad de uso:** facilidad de interacción con el usuario.
- **Verificabilidad:** facilidad para la realización de pruebas.
- **Extensibilidad:** facilidad de adaptación ante nuevos requisitos.
- **Reutilización:** capacidad de que el software, como bloque, actúe como componente dentro de un nuevo sistema.

1.7.2 MÉTRICAS Y CALIDAD DEL SOFTWARE

Dado que la finalidad general del software es ser funcional y de calidad, la métrica proporciona medidas que permiten evaluar esa calidad de manera objetiva.

Para controlar estas medidas se establecen modelos de calidad. Un modelo de calidad se define como "el conjunto de características y las relaciones entre ellas que proveen la base para la especificación de los requisitos de calidad y la evaluación de calidad".

1.7.3 CONCEPTO DE MÉTRICA Y SU IMPORTANCIA EN LA MEDICIÓN DE CALIDAD

Una métrica es una medida de alguna propiedad del software. Proporciona un dato de aplicación durante los procesos de evaluación en el ciclo de vida.

Pueden ser clasificadas según el criterio que se pretenda aplicar, de esta manera tenemos métricas de complejidad, de calidad y de desempeño.

Los objetivos de la medición tienen que estar lo suficientemente claros antes de comenzar el proceso de recogida de datos, recomendándose la automatización del proceso de captura de datos y análisis. Además, unas buenas métricas deben ser simples, fáciles de calcular, de naturaleza empírica, objetivas, independientes del lenguaje de programación, fáciles de usar.

1.7.4 PRINCIPALES MÉTRICAS EN LAS FASES DEL CICLO DE VIDA DEL SOFTWARE

Para cada una de las fases del ciclo de vida del software se pueden utilizar diferentes métricas, a continuación veremos algunas de ellas:

- Métricas en el modelo de análisis: intentan predecir el tamaño del sistema, ya que se puede establecer una relación directa entre el tamaño y la complejidad del diseño. En todas estas métricas se obtiene un valor, partiendo de ciertos datos y aplicando diferentes fórmulas. En esta etapa se pueden utilizar métricas basadas en función y las "métricas *bang*", que toman como referencia los diagramas de flujo.
- Métricas de diseño: podemos utilizar las métricas de diseño de nivel arquitectónico, que miden la arquitectura del sistema, sin importar cómo se comporta el módulo de manera interna; las métricas de diseño de nivel de

componentes, que tienen en cuenta la cohesión, el acoplamiento y la complejidad del módulo; y las métricas de diseño de interfaz, que se fundamentan en la asignación de un coste a la secuencia de acciones que el usuario debe realizar.

- Métricas de codificación: ayudan durante la fase de codificación del programa, principalmente ofreciendo datos sobre la complejidad del código, como la complejidad "ciclomática" o la complejidad lógica de un sistema.
- Métricas de pruebas: se aplican durante la fase de prueba, donde podemos evaluar el éxito de las pruebas, la cobertura de los requisitos y la tasa de eliminación de errores.

1.7.5 ESTÁNDARES PARA LA DESCRIPCIÓN DE LOS FACTORES DE CALIDAD

Son una serie de recomendaciones a seguir para garantizar que el software, además de tener calidad, cumpla con los requisitos del cliente. El estándar es de adopción voluntaria y no garantiza el resultado final, pero su uso significa que el proceso cumple con unos mínimos.

1.7.6 ISO-9126

ISO 9126 es un estándar internacional para la evaluación de la calidad del software. Está dividido en cuatro partes, las cuales dirigen en realidad las métricas externas, las métricas internas y la calidad en las métricas de uso y expendido. El modelo de calidad establecido en la primera parte del estándar, ISO 9126-1, clasifica la calidad del software en un conjunto estructurado de características y sub-características de la siguiente manera:

- Funcionalidad: el conjunto de atributos que se relacionan con la existencia de un conjunto de funciones y sus propiedades específicas. Las funciones son aquellas que satisfacen las necesidades implícitas o explícitas. Entre estos atributos, se evalúa la adecuación, la exactitud, la "interoperabilidad", la seguridad y el cumplimiento funcional.
- Fiabilidad: el conjunto de atributos relacionados con la capacidad del software de mantener su nivel de prestación bajo condiciones establecidas durante un período establecido. Entre los atributos se evalúa la madurez, la capacidad de recuperación, la tolerancia a fallos y el cumplimiento de fiabilidad.
- Usabilidad: conjunto de atributos relacionados con el esfuerzo necesario para su uso y con la valoración individual de tal uso, por un establecido o implicado conjunto de usuarios. Los atributos que se evalúan son el aprendizaje, la comprensión, la operatividad y la capacidad de atracción.
- Eficiencia: conjunto de atributos relacionados con la relación entre el nivel de desempeño del software y la cantidad de recursos necesitados bajo unas condiciones establecidas. Se evalúa el comportamiento en el tiempo y el comportamiento de los recursos.
- Mantenibilidad: conjunto de atributos relacionados con la facilidad de extender, modificar o corregir errores en un sistema software. Los atributos evaluados en la mantenibilidad son la estabilidad, facilidad de análisis, la facilidad de cambio y la facilidad de pruebas.
- Portabilidad: conjunto de atributos relacionados con la capacidad de un sistema software para ser transferido desde una plataforma a otra. Se evalúa la capacidad de instalación y la capacidad de reemplazamiento.
- Calidad en uso: Conjunto de atributos relacionados con la aceptación por parte del usuario final y del departamento de Seguridad. Entre otros, evaluamos la eficacia, la productividad, la seguridad y la satisfacción.

1.7.7 OTROS ESTÁNDARES. COMPARATIVA

Estándar	Órgano Normalizador	Descripción	Ventajas
SPICE	Software Process Improvement and Capability Etermination ISO	Apoya la implementación de una norma internacional para la evaluación de procesos de software	Proporciona los principios requeridos para realizar la evaluación de la calidad, e implementación de dicho modelo de procesos en una organización
PSP	SEI Software Engineering ISO	Permite estimar el tiempo y el tamaño del software para llevar a cabo una buena administración de la calidad	Estima el tiempo de las actividades
ISO 9001:2000	ISO	Promover el desarrollo de la estandarización de las actividades relacionadas al mundo para facilitar el intercambio internacional	Realización del producto
CMME	Content Management Made Easy	Contenido web, sistema de gestión	Su utilización es muy sencilla
TSP	SEI Software Engineering	Método de establecimiento y mejora de trabajo en equipo	Mejora el trabajo del software
CMMI	Capability Maturity Model Integration	Ayuda a mejorar los procesos de realización de software	Mejora los procesos de construcción de software

1.8 HERRAMIENTAS DE USO COMÚN PARA EL DESARROLLO DEL SOFTWARE

Actualmente, existen herramientas que ayudan a los desarrolladores en el proceso de creación de software. Estas herramientas pueden ser clasificadas de acuerdo a la finalidad de la misma.

1.8.1 EDITORES ORIENTADOS A LENGUAJES DE PROGRAMACIÓN

El editor es la principal herramienta de trabajo de un programador. Un editor poderoso e inteligente, puede hacer una gran diferencia en el nivel de eficiencia con el cual se desempeña un desarrollador. Las opciones a la hora de elegir el compañero óptimo para programar son muy variadas y, en muchos casos, están optimizadas para diferentes tipos de proyectos o lenguajes.

Son tan importantes como un compilador -imagínate como programar sin editores- y actualmente suelen incluir funciones específicamente dedicadas a la programación, como resaltado de sintaxis, auto-identificación, etc. Grandes editores de texto son GNU Emacs, Vim, Scite, Notepad++.

1.8.2 COMPILADORES Y ENLAZADORES

Los compiladores son programas que "traducen" un fichero de código fuente de cualquier lenguaje al lenguaje ensamblador, y llama cuando sea necesario al ensamblador y al *linker* (enlazador). Los más importantes son GCC (para C), G++ (para C++), G77 (para Fortran 77), Microsoft Visual C++, etc.

Un compilador es un programa que permite traducir el código fuente de un programa en lenguaje de alto nivel a otro lenguaje de nivel inferior (típicamente lenguaje de máquina). De esta manera, un programador puede diseñar un programa en un lenguaje mucho más cercano a cómo piensa un ser humano, para luego compilarlo a un programa más manejable por una computadora. Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente.

Los enlazadores, son programas que enlazan varios ficheros objeto en lenguaje binario para crear un único fichero, el ejecutable del programa. Un enlazador es un programa que toma los objetos generados en los primeros pasos del proceso de compilación y la información de todos los recursos necesarios (biblioteca), quita aquellos recursos que no necesita y enlaza el código objeto con su(s) biblioteca(s) con lo que finalmente produce un fichero ejecutable o una biblioteca. En el caso de los programas enlazados dinámicamente, el enlace entre el programa ejecutable y las bibliotecas se realiza en tiempo de carga o ejecución del programa.

1.8.3 GENERADORES DE PROGRAMAS

Los generadores de programas son herramientas que, a partir de ciertas entradas como datos y determinados elementos, generan el ejecutable del programa que queremos desarrollar. Uno de los más utilizados es EMF (*Eclipse Modelling Framework*), el *plug-in* de Eclipse, que permite a partir de una serie de modelos que el desarrollador va especificando, crear el código que ejecutará el programa.

1.8.4 DEPURADORES

Como su nombre indica, sirve para depurar o corregir errores. Se encargan de ejecutar, paso a paso, el programa para detectar los errores que puedan existir. Son particularmente útiles cuando el programa parece estar bien pero no da el resultado esperado (se cuelga, da resultados erróneos, etc.). El más importante es GDB. Actualmente, casi todas las IDE incluyen uno o deberían incluirlo.

1.8.5 DE PRUEBA Y VALIDACIÓN DE SOFTWARE

Son herramientas que permiten automatizar la etapa de prueba y validación del software. Ayudan al desarrollador a definir y ejecutar las diferentes pruebas a realizar. Entre las más utilizadas podemos mencionar:

- Selenium. Compuesto por dos herramientas, Selenium IDE y SeleniumWebDriver. La primera permite crear casos de prueba para aplicaciones web y la segunda los ejecuta. Utiliza los siguientes lenguajes: Python, Ruby, Java y C#; y ejecuta pruebas de aplicaciones para Android y iOS.
- JMeter. Permite realizar pruebas funcionales y de rendimiento para aplicaciones web.
- Testlink. Permite crear y gestionar casos de prueba, organizarlos en planes de pruebas, realizar un seguimiento de los resultados, establecer trazabilidad con los requisitos, generar informes etc.

1.8.6 OPTIMIZADORES DE CÓDIGO

Son herramientas, que a partir del análisis del código generado por el programador, crean un nuevo código más compacto y eficiente, eliminando por ejemplo sentencias que no se ejecutan nunca, simplificando expresiones aritméticas, etc. La profundidad con que se realiza esta optimización varía mucho de unos optimizadores a otros.

Un compilador es una herramienta que trata de minimizar ciertos atributos de un programa informático con el fin de aumentar la eficiencia y rendimiento. Las optimizaciones del compilador se aplican generalmente mediante una secuencia de transformaciones de optimización, algoritmos que transforman un programa para producir otro con una salida semánticamente equivalente pero optimizada.

1.8.7 EMPAQUETADORES

Un *packer* o empaquetador es un programa que se utiliza para reducir el tamaño de un archivo ejecutable, generalmente, mediante la compresión del mismo. Se utiliza empaquetadores para poder hacer la distribución del código ejecutable del programa.

1.8.8 GENERADORES DE DOCUMENTACIÓN DE SOFTWARE

Son herramientas que dan soporte a la etapa de análisis, diseño y codificación del ciclo de vida del software. Permiten que el equipo de desarrollo genere la documentación relacionada con la etapa a medida que va trabajando.

1.9 GESTIÓN DE PROYECTOS

La construcción de software es una actividad compleja que involucra a mucha gente que trabaja durante mucho tiempo. Para conseguir un proyecto de software exitoso, se debe comprender el ámbito del trabajo a realizar, los riesgos en los que se puede incurrir, los recursos requeridos, las tareas a llevar a cabo, el esfuerzo (coste) a consumir y el plan a seguir. Este conjunto de actividades es complejo y debe ser gestionado.

La gestión de proyectos involucra la planificación, la supervisión y el control del personal, el proceso y los eventos que ocurren, mientras que el software evoluciona desde una idea preliminar hasta una implementación operativa.

1.9.1 PLANIFICACIÓN DE PROYECTOS

El objetivo de la planificación del proyecto de software es proporcionar un marco de trabajo que permita al gestor hacer estimaciones razonables de recursos, coste y planificación temporal. Estas estimaciones se hacen dentro de un marco de tiempo limitado al comienzo de un proyecto de software y deberían actualizarse regularmente a medida que progresa el proyecto.

La planificación la realizan los gestores del software, utilizando la información solicitada a los clientes y a los ingenieros de software y los datos de las métricas de software obtenidos de proyectos anteriores.

La estimación comienza con una descripción del ámbito del producto. Hasta que no se delimita el ámbito no es posible realizar una estimación con sentido. El problema es entonces descompuesto en un conjunto de problemas

de menor tamaño y cada uno de estos se estima guiándose con datos históricos y con la experiencia. Es aconsejable realizar las estimaciones utilizando al menos dos métodos diferentes (como comprobación). La complejidad y el riesgo del problema se consideran antes de realizar una estimación final.

¿Cuál es el producto obtenido? Se obtiene una tabla que indica las tareas a desarrollar, las funciones a implementar y el coste, esfuerzo y tiempo necesarios para la realización de cada una. También se obtiene una lista de recursos necesarios para el proyecto.

El principal objetivo de la planificación es la configuración del calendario del proyecto o también denominado programa de tiempos. Este consiste en una representación gráfica de todas las actividades del proyecto necesarias para producir el resultado final que permite al jefe de proyecto coordinar de una forma efectiva al equipo de desarrollo durante el transcurso del mismo. El calendario debería ser dinámico, es decir, debería variar a medida que avanza el proyecto si surgen cambios no previstos en su extensión, sus plazos, etc. Sin este calendario, el control del proyecto se hace casi imposible. La dirección del proyecto podría ser extremadamente difícil si no se han identificado previamente las actividades individuales y sus interrelaciones. El control del proyecto se basa en la supervisión periódica y en la comparación de los resultados con los previstos en el calendario. Si no existe este calendario, es imposible estimar el estado del proyecto acertadamente.

Para que un programa de tiempos sea efectivo, debe tener las siguientes características:

- Comprensible por todas aquellas personas que van a utilizarlo.
- Suficientemente detallado para servir de base para medir y controlar el progreso del proyecto.
- Capaz de señalar las actividades críticas.
- Flexible, fácilmente modificable.
- Basado en estimaciones de tiempos fidedignas según los compromisos adquiridos por los responsables de cada tarea.
- Ajustable a los recursos disponibles.
- Compatible con los planes de otros proyectos que compartan los mismos recursos.

El primer cometido del jefe de proyecto es la realización del plan de proyecto. Es un documento que describe los trabajos que se van a realizar y la forma en que el jefe de proyecto va a dirigir su desarrollo. Debe definir un conjunto de tareas, coordinadas en el tiempo, así como los recursos necesarios para cumplir los objetivos marcados a cada tarea con el propósito de satisfacer los requisitos contractuales (o los compromisos adquiridos en desarrollos internos de la compañía).

Independientemente del tamaño del proyecto, este debe contener un plan que especifique aquello que hay que hacer, cuándo, dónde y quién lo realizará, así como el coste asociado. El plan no necesita estar muy elaborado, pero debe definir de una manera precisa el trabajo del que se responsabiliza el jefe del proyecto y los métodos que aseguren el éxito del desarrollo. Es muy importante que, en su elaboración, el jefe de proyecto haya negociado y acordado los compromisos de esfuerzo, calidad y plazo que cada persona del equipo (o su superior) deberá cumplir para lograr el éxito del proyecto en las tareas que tiene asignadas.

1.9.2 CONTROL DE PROYECTOS

Una vez realizada la planificación del proyecto y comenzada la ejecución del proyecto es necesario realizar el control del mismo.

El seguimiento y control de los proyectos de desarrollo de software tiene como objetivo fundamental la vigilancia de todas las actividades de desarrollo del sistema que se está construyendo. Es una de las labores más importantes en todo desarrollo del producto, pues un adecuado control hace posible evitar desviaciones en costes y plazos, o al

menos detectarlas cuanto antes. Para poder ejercer un correcto seguimiento y control del proyecto es necesario que el Jefe de Proyecto dedique todo el tiempo que sea preciso a vigilar el estado de cada una de las tareas que se están desarrollando, prestando especial interés a aquellas que están sufriendo algún retraso. En el momento en que se detecta cualquier desviación hay que analizar las causas para poder efectuar las correcciones oportunas y recuperar el tiempo perdido. Las actividades de seguimiento y control de un proyecto se llevan a cabo desde la asignación de las tareas hasta su aceptación interna por parte del equipo de proyecto, previa a la aceptación del Cliente.

1.9.3 EJECUCIÓN DE PROYECTOS

Esta es la etapa de desarrollo del trabajo en sí la cual es responsabilidad del jefe de proyecto con la supervisión del cliente. Durante la ejecución del proyecto, se debe poner énfasis en la comunicación para tomar decisiones lo más rápido posible en caso de que surjan problemas. Así, es posible acelerar el proyecto estableciendo un plan de comunicación, por ejemplo, usando tableros que muestren gráficamente los resultados del trabajo, permitiendo que el jefe del proyecto arbitre en caso de variaciones e informes de progreso que permitan a todas las personas involucradas en él estar informadas sobre las acciones en progreso y aquellas terminadas. Generalmente, "informar" incluye la preparación completa y la presentación de informes sobre las actividades. Además, se deberán organizar regularmente reuniones para administrar el equipo del proyecto, es decir, discutir regularmente el progreso del proyecto y determinar las prioridades para las siguientes semanas.

1.9.4 HERRAMIENTAS DE USO COMÚN PARA LA GESTIÓN DE PROYECTOS

Para trabajar de forma profesional en un proyecto, muchas veces necesitamos algo más que una lista de tareas. Si los clientes son varios, entonces ya es imprescindible encontrar algún buen programa de gestión de proyectos, estable pero también flexible. Existen herramientas de pago como puede ser el Microsoft Project que es un software de administración de proyectos diseñado, desarrollado y comercializado por Microsoft para asistir a administradores de proyectos en el desarrollo de planes, asignación de recursos a tareas, dar seguimiento al progreso, administrar presupuesto y analizar cargas de trabajo. Es útil para la gestión de proyectos, aplicando procedimientos descritos en el PMBoK del Project Management Institute.

Por otro lado podemos encontrar programas de software libre que además de ser potentes, cuentan detrás con una comunidad de desarrolladores y pueden hacernos ahorrar bastante en costes.

- Colabtive: es la alternativa *open source* a herramientas propietarias como Basecamp. Permite importar desde Basecamp e incluye funciones similares como la gestión de diferentes proyectos, los *milestones* y las listas de tareas. También mide el tiempo dedicado a las tareas, emite informes y cuenta con varios *plug-ins* para extender sus funciones.
- Project HQ: también similar a Basecamp, Project HQ está construido sobre Python, Pylons y SQLAlchemy, y su base de datos es totalmente independiente. Gestiona distintas compañías, miembros y proyectos y cuenta con *milestones* y listas de tareas. Es configurable visualmente usando CSS.
- Gantt PV: gratuito, es un programa simple, sin complicaciones, que se basa en diagramas de Gantt para planificaciones de proyectos y seguimiento de tareas. Está disponible para Windows, Mac OS y Linux.
- Clocking IT: tiene diagramas de Gantt interactivos, más otras utilidades como contador de tiempo, varias formas de comunicación, seguimiento e indexación de los cambios y unos muy buenos informes de avance.

- TeamWork: una excelente interfaz para una herramienta online que permite hacer un seguimiento de distintos proyectos y equipos de trabajo, con una versión optimizada para acceder desde móviles. Tienen licencias gratuitas para organizaciones sin ánimo de lucro y *bloggers*. Disponible para Mac OS X, Linux y Windows.
- iceScrum: tiene la misma interfaz para todos los roles. Incluye registros de historias de usuario (*backlogs*), de asuntos, de problemas y pruebas, chat en línea, *timeline* e indicadores de producto.
- Achievo: además de la utilidad de gestión de proyecto, que divide según el tiempo de su ejecución, incluye calendarios, estadísticas, plantillas y notas. No hay tarifas de licencia o limitaciones para su uso.
- dotProject.net: otra herramienta basada en la web dotProject. Lleva un tiempo funcionando y no hay ninguna empresa detrás de ella, está sostenida por voluntarios y usuarios. Permite la gestión para múltiples clientes, con herramientas para gestión de tareas, agendas y comunicaciones.
- GanttProject: un programa de escritorio multiplataforma que corre sobre Windows, Mac OS X y Linux, totalmente gratuito. Incluye diagramas de Gantt, asignación de las personas que trabajarán en el proyecto, y permite exportar los diagramas como imágenes, mientras genera informes en PDF y HTML. Permite interactuar con Microsoft Project, importando y exportándolos a sus formatos.
- TaskJuggler: un gestor de proyectos realmente potente y superior a otros que usan herramientas para editar diagramas de Gantt. Cubre todos los aspectos de desarrollo de un proyecto, desde la primera idea hasta su fin. Ayuda a medir su campo de alcance, asignación de recursos, esquema de costos y ganancias, riesgo y gestión de las comunicaciones.

1.10 TEST DE CONOCIMIENTOS

1 Señale la respuesta correcta con respecto a los modelos de ciclo de vida del software:
a) En el modelo en cascada no es necesario esperar a la finalización de una etapa para comenzar con la siguiente.
b) En el modelo iterativo el cliente evalúa el producto y lo corrige o propone mejoras.
c) En el modelo "incremental" aumenta el riesgo en el desarrollo de sistemas largos y complejos.
d) El desarrollo rápido (RAD) es una metodología basada en el paradigma orientado a objetos.

2 Las "declaraciones en lenguaje natural y en diversos diagramas de los servicios del sistema y de las restricciones bajo las que debe operar" se denominan:
a) Requisitos de usuario.
b) Requisitos del sistema.
c) Requisitos funcionales.
d) Requisitos no funcionales.