Ingeniería en Desarrollo de software
3er semestre

Programa de la asignatura
Introducción a la ingeniería de software

Unidad 3.
Diseño, codificación, pruebas y mantenimiento

Clave:

Ingeniería:	TSU:
15142318	**16142318**

Universidad Abierta y a Distancia de México

Índice

Unidad 3. Diseño, codificación, pruebas y mantenimiento

Presentación de la unidad

En la pasada unidad abarcamos los temas de análisis y modelado de requerimientos; conociste el concepto de requerimiento y su clasificación. Respecto al modelado se abarcaron dos clasificaciones de diagramas UML: los del modelado del dominio y los del modelado de la interacción. Para continuar con el desarrollo de software, es necesario retomar los diagramas elaborados en la etapa de análisis y modelado de requerimientos para el diseño de la interfaz del sistema, de las bases de datos, procedimientos y de los posibles reportes del sistema.

En la presente unidad podrás conocer aspectos importantes del diseño del software: los lineamientos para el diseño del sistema, la interfaz y el código. Además, identificarás los principales tipos de pruebas y comprenderás en qué consiste la etapa de mantenimiento del ciclo de desarrollo de software.

Al término de la presente unidad, habrás abarcado las principales fases del desarrollo del software y sus características para que puedas identificar cómo se conforma un producto de software: desde la idea, hasta su instalación y mantenimiento.

Propósito

Al término de esta unidad lograrás:

- Analizar lineamientos del diseño de la interfaz.
- Analizar lineamientos de la codificación.
- Analizar tipos de pruebas y el proceso de mantenimiento.

Competencia específica

Seleccionar el desarrollo de las etapas de diseño, codificación, pruebas y mantenimiento para resolver un caso de estudio, analizando sus características.

3.1. Diseño

El diseño forma parte de las etapas de desarrollo de software, en el cual se utilizan técnicas y principios para bosquejar la estructura de los componentes de un software con suficiente detalle y permitir su interpretación y construcción. La base del diseño son los requerimientos del software y su análisis. A continuación verás diferentes elementos que se deben diseñar.

Es importante puntualizar que este tema abarca el diseño de sistemas –considerando que un sistema es un conjunto de partes o elementos organizados y relacionados que interactúan entre sí con objetivo específico–. Éstos reciben principalmente datos de entrada y proveen información de salida. Los sistemas de software son abstractos y cada uno puede contener subsistemas que pueden ser parte de uno más grande. A continuación se explicará cómo se diseña un sistema de software con diferentes vistas, iniciando con el diseño del contexto.

Diseño de contexto

Además de diseñar el sistema se identifican las fronteras del mismo y de otros sistemas con los que esté vinculado. Para realizar esto, los modelos de contexto y de interacción cuentan con diferentes vistas para el diseño del sistema en su entorno. Por lo tanto, se puede definir un modelo de contexto del sistema como: un modelo estructural que incluye a los otros sistemas con los que se relaciona y, un modelo de interacción es un modelo dinámico que muestra el comportamiento del sistema con su entorno.

Cuando se modelan las interacciones de un sistema con su entorno, se utiliza un enfoque abstracto sin muchos detalles. Puede representarse con un caso de uso –el cual se representa con una elipse que equivale a una interacción con el sistema. El actor puede ser un usuario o un sistema de información, por ejemplo:

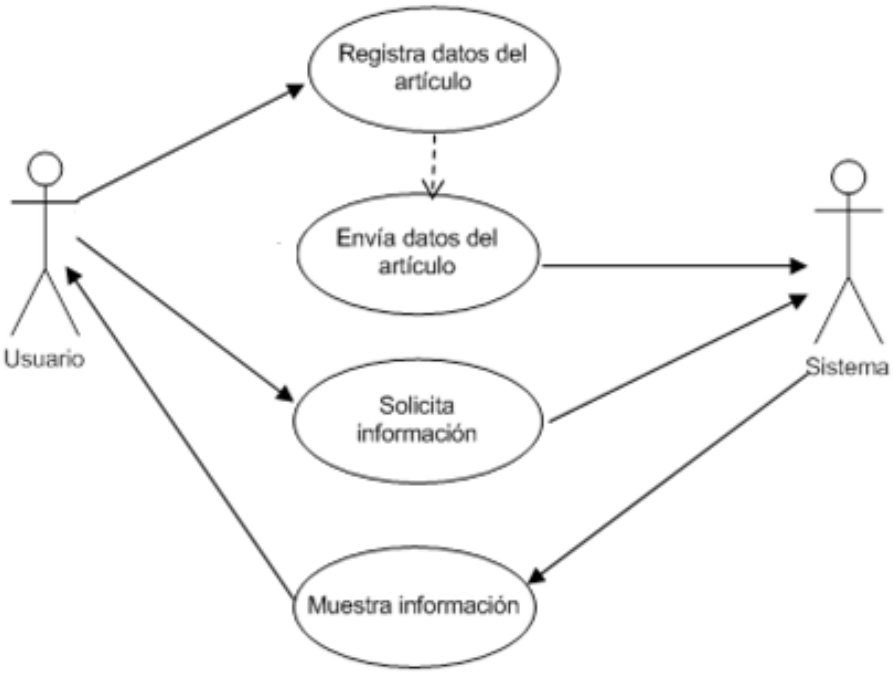


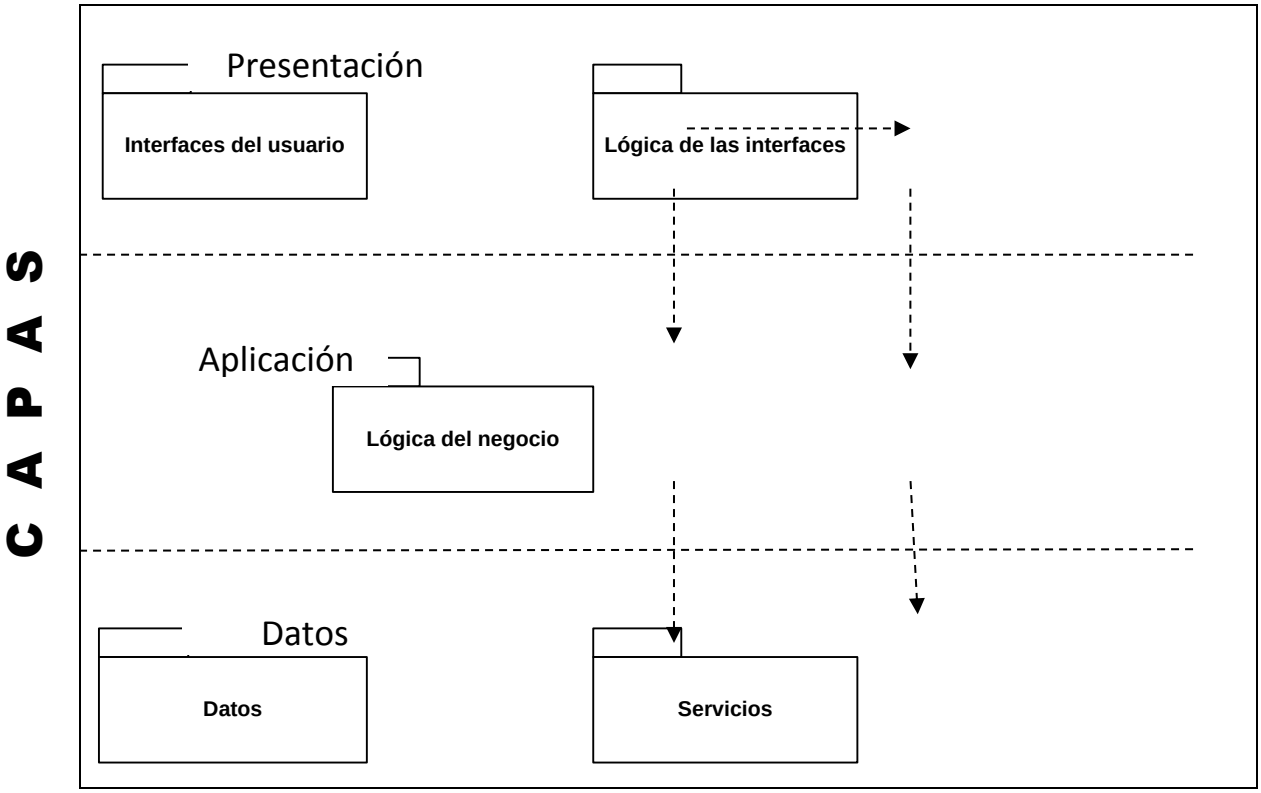
Diagrama de casos de uso de registro y consulta de información.

En el diagrama anterior podemos ver al actor **Usuario** interactuando con varios casos de uso y con el actor **Sistema**. El **Usuario** puede *registrar los datos del artículo* y *solicitar información*. Por otro lado el actor **Sistema** *muestra la información* solicitada. Vemos

también a los casos de uso interactuando entre ellos, tal situación del caso de uso *registra datos del artículo* y *envía datos del artículo*.

Diseño arquitectónico

Ya que se definieron las interacciones entre el sistema y su entorno, se utiliza esta información como base para diseñar la arquitectura del sistema. Se deben identificar los componentes del sistema y sus interacciones, después se organizan en un patrón arquitectónico como un modelo en capas o cliente – servidor.

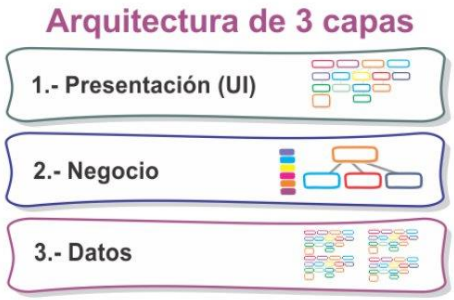


Diseño arquitectónico de capas

En este diagrama se puede apreciar un diseño arquitectónico de la programación por capas que son las siguientes:

Capa de presentación: es la que ve el usuario, también se le puede llamar capa de usuario o interfaz gráfica. Permite la interacción del usuario con el software dándole información y permitiéndole introducir datos. Contiene el código que soporta la lógica de la funcionalidad de las interfaces. Y se comunica con la capa de la aplicación utilizando la lógica del negocio.

Capa de negocio: contiene todos los programas que se ejecutan, recibe las peticiones de la capa del usuario y se le regresan los resultados de la operación; además se comunica con la capa de datos para hacer peticiones al gestor de base de datos (por ejemplo instrucciones SQL).



Capa de datos: es la que contiene todos los datos los cuales son almacenados y administrados en uno o más gestores de bases de datos. Mantiene comunicación con la capa de negocio atendiendo las peticiones almacenamiento o recuperación de datos que ésta le demande.

A continuación se presenta otra vista del diseño con un enfoque orientado a objetos.

Identificación de la clase objeto

El diseño orientado a objetos es diferente al diseño estructurado, ya que no se realiza un problema como en tareas (subrutinas) ni en términos de datos, sino que se analiza el problema como un sistema de objetos que interactúan entre sí. Lo primero que debe realizarse es identificar los objetos del programa y, por lo tanto, de las clases con sus atributos y comportamientos, así como las relaciones entre éstas y su posterior implementación.

Los sistemas orientados a objetos se deben diseñar considerando las 4 capas siguientes:

La capa del subsistema: en esta capa se describen los subsistemas en que fue dividido el software y la solución técnica que los soporta.

La capa de clase y objetos: contiene clases, generalizaciones de clases y representaciones de diseño de cada objeto.

La capa de mensajes: contiene los detalles que le permiten a cada objeto comunicarse con sus colaboradores. Esta capa establece las interfaces externas e internas para el sistema.

La capa de responsabilidades: contiene estructuras de datos y diseño algorítmico para todos los atributos y operaciones de cada objeto.

Para iniciar todo proyecto es necesario conocer los diferentes métodos para el diseño de sistemas, tales como los métodos: Booch, Coad y Yourdon, los cuales son los métodos de mayor importancia en el ámbito del análisis y el diseño orientado a objetos. El método fue realizado por Grady Booch –quien además es reconocido por participar en la creación del lenguaje unificado de modelado (UML) –. El enfoque de este método es la representación

de la vista lógica del software. Existen otros métodos con diferentes enfoques, tal es el caso del método de Coad y Yourdon cuyo énfasis es en la aplicación e infraestructura. A continuación veremos en qué consisten:

Método Booch: consiste en un proceso evolutivo basado en "micro-desarrollos" y "macro-desarrollos". Abarca las siguientes fases:

- **Planificación arquitectónica** que consiste en agrupar y distribuir objetos por niveles, crear y validar un prototipo.
- **Diseño táctico** para definir reglas de uso de operaciones y atributos, definir políticas y sus escenarios, refinar prototipo.
- **Planificación de la realización** por medio de la organización de escenarios asignando prioridades, diseño de la arquitectura de escenarios, construir cada escenario, ajustar objetos y el plan de la realización incremental.

Método de Coad y Yourdon: su enfoque de diseño se dirige a la aplicación y a la infraestructura. Su proceso consiste en:

- *Componente del dominio del problema*, que consiste en agrupar las clases y su jerarquía, simplificar la herencia, refinar diseño para mejorar rendimiento, desarrollo de la interfaz, generación de los objetos necesarios y revisión del diseño.
- *Componente de interacción humana*, abarca la definición de actores humanos, desarrollo de escenarios, diseño de la jerarquía de órdenes, refinar la secuencia de interacciones, diseño de clases y su respectiva jerarquía.
- *Componente para gestión de tareas*, para identificar tipos de tareas, las prioridades, identificar la tarea coordinadora, diseño de objetos para cada tarea.
- *Componentes para la gestión de datos,* nos permite diseñar la estructura de datos, los servicios, seleccionar herramientas para la gestión de datos y diseño de clases y jerarquías.

Éstos y otros métodos nos ayudan para el análisis y diseño orientado a objetos con el enfoque que cada uno propone. A continuación veremos un modelo genérico del diseño orientado a objetos.

Modelo genérico del diseño orientado a objetos

Mientras que el Análisis Orientado a Objetos (AOO) es una actividad de clasificación, en otras palabras, se analiza un problema identificando sus objetos y de qué clase son, así como sus relaciones y comportamiento del objeto. El Diseño Orientado a Objetos (DOO) permite indicar los objetos que se derivan de cada clase y su relación, además muestra cómo se desarrollan las relaciones entre objetos, cómo se debe implementar su comportamiento y cómo implementar la comunicación entre objetos. En general define cuatro componentes:

- **Dominio del problema**: subsistemas que implementan los requerimientos.
- **Interacción humana**: subsistemas reutilizables de interfaz gráfica.
- **Gestión de tareas**: subsistemas responsables del control de tareas concurrentes.
- **Gestión de datos**: subsistema responsable del almacenamiento y recuperación de datos (Pressman, 2010, pp. 407- 412).

Ya que has visto diferentes modelos para el análisis y el diseño de un software a continuación observa cómo es su proceso del diseño.

3.1.1. Diseño del sistema

El diseño es un proceso iterativo (que se repite) que aplica diversas técnicas y principios para definir un dispositivo, proceso o sistema detalladamente para permitir su realización física. A continuación se presentan las principales actividades involucradas en este proceso de acuerdo a Pressman (2010, pp. 413-412).

- Dividir el modelo analizado en subsistemas con las siguientes características:

- Definir una adecuada interfaz.
- Establecer la comunicación entre clases.
- Mantener un número pequeño de subsistemas.
- Los subsistemas pueden dividirse internamente para reducir la complejidad.

- Identificar concurrencia y asignar de alguna de estas dos maneras:
  - Asignar cada subsistema a un procesador independiente.
  - Asignar los subsistemas al mismo procesador y ofrecer soporte de concurrencia a través de las capacidades del sistema operativo
- Asignar subsistemas a procesadores y tareas con la siguiente estrategia:
  - Determinar las características de la tarea.
  - Definir una tarea coordinadora y objetos asociados.
  - El coordinador y las otras tareas que lo integran.

Tarea: el nombre del objeto. *Descripción:* descripción del propósito del objeto. *Prioridad:* baja, media, alta *Operaciones:* una lista de servicios del objeto. *Coordinado por:* como invocar el comportamiento del objeto. Comunicación*:* datos de entrada – salida relevantes.

La plantilla básica de la tarea (Diseño estándar).

- Elegir una de las dos estrategias para la gestión de datos: (1) la gestión de datos para la aplicación, y (2) la creación de infraestructura para el almacenamiento y recuperación de objetos.
- Identificar recursos y mecanismos para acceder a ellos.
  Los recursos globales del sistema pueden ser unidades externas por ejemplo torres de discos, procesadores, líneas de comunicación, etc.
- Diseño de control apropiado para el sistema.
  Definir el componente interfaz hombre-máquina (IHM), se pueden aplicar los casos de uso como datos de entrada. Ya que se definió el actor y su escenario de uso, se

identifica la jerarquía de órdenes o comandos lo cual define las principales categorías de menús del sistema. Esto se va refinando con cada interacción hasta que cada caso de uso pueda implementarse navegando a través de la jerarquía de funciones.

- Definir como manipular condiciones límite.
  Ya que se definieron los subsistemas es necesario definir las colaboraciones que existen entre ellos, para esto es útil un diagrama de colaboración como se vio en el capítulo anterior.

Ya que has conocido el proceso del diseño del sistema, es importante conocer los aspectos importantes de la arquitectura de un software, subtema que verás a continuación.

3.1.2. Tipos de Arquitecturas

La arquitectura de software es una abstracción que muestra un marco de referencia que sirve de guía en la construcción de un software. Ésta se construye con diferentes diagramas que ayudan a mostrar diferentes vistas del sistema.

La arquitectura de un software puede ser muy simple como un módulo de un programa o puede ser tan complejo como incluir bases de datos, comunicarse con componentes de software o aplicaciones que puedan intercambiar datos entre sí. Incluso, aplicaciones distribuidas que incluyan servidores web, servidores de aplicaciones o gestores de contenido y actualmente la mayoría de los sistemas de cómputo que se construyen son distribuidos.

A los sistemas distribuidos se les define como:

“Una colección de computadoras independientes que aparecen al usuario como un solo sistema coherente” (Sommerville, 2011, p. 480).

Los sistemas distribuidos presentan las siguientes ventajas:

- Comparten recursos de hardware y software.

- Diseñados de manera estándar que permiten la combinación de hardware y software de diferentes proveedores.
- Permiten que grandes procesos puedan ejecutarse al mismo tiempo y en computadoras independientes en red, a esta característica se le llama concurrencia.
- Permiten agregar nuevos recursos o aumentar las capacidades del sistema, lo cual se conoce como escalabilidad.
- Pueden existir computadoras con fallas, pero la pérdida completa de servicio sólo ocurre cuando hay una falla de red. Cuando ocurre esto se dice que el sistema es tolerante a las fallas.

Estos sistemas son más complejos que los sistemas que se ejecutan en un solo procesador. Algunos de los problemas de diseño más importantes que se deben considerar son:

- *Transparencia*: el reto que se tiene en este atributo es lograr que cada nodo trabaje como si fuese independiente, unificando todos los recursos y sistemas para la aplicación y por lo tanto para el usuario.
- *Apertura*: en los sistemas distribuidos se debe dar la apertura a los nuevos servicios pero sin perjudicar a los ya existentes.
- *Escalabilidad*: poder incrementar las capacidades de trabajo sin perder las capacidades y usos anteriores.
- *Seguridad:* el reto de la seguridad es mantener la *confidencialidad* por medio de la protección contra individuos no autorizados. *Integridad*, con la protección contra la alteración o corrupción y la *disponibilidad* protegerse ante la posibilidad de interferencia.
- *Calidad en el servicio:* identificar la configuración que mejor se adapte y sea aceptable por los usuarios.
- *Gestión de fallas:* Tiene el reto de que el sistema continúe funcionando ante fallos de algún componente de manera independiente, para lo cual debe existir un plan de mitigación ante cualquier riesgo latente.

Modelos de interacción

El modelo de interacción tiene que ver con el rendimiento y la dificultad de poner límites temporales en un sistema distribuido, hay varios tipos de interacción.

- Paso de mensajes.
- Radiado (multicast).
- Llamada a procedimiento remoto (RPC).
- Cita multiflujo (multithreaded).
- Publicación/subscripción (publish/subscribe)

Y pueden ser representados con los diagramas de interacción como el que se muestra a continuación:

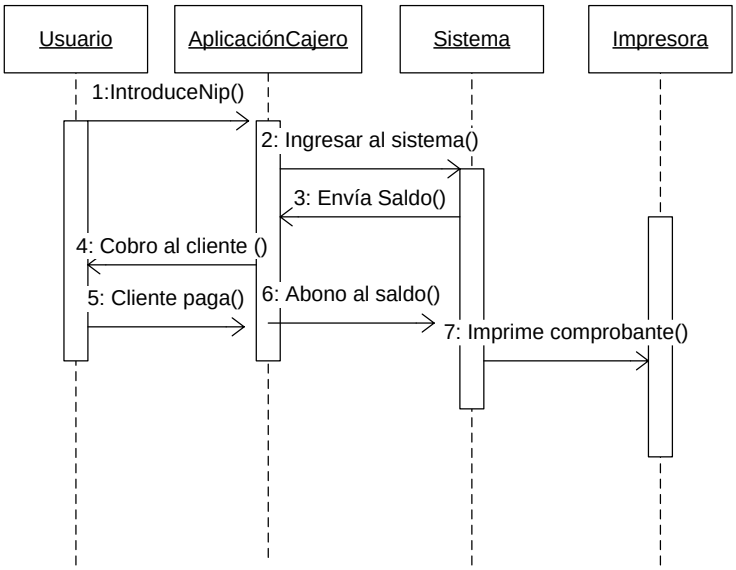
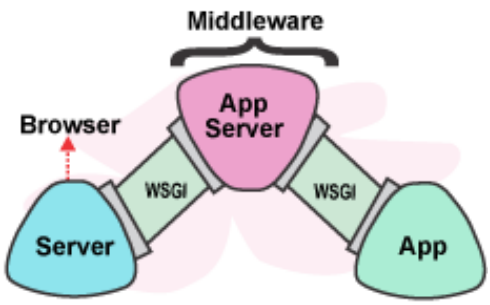


Diagrama de interacción

Middleware

Los sistemas de información se construyen con diferentes lenguajes de programación, la base de datos se genera en algún software gestor específico, además pueden ser ejecutados en distintos procesadores, por lo tanto un sistema distribuido necesita software para gestionar todas



estas partes y asegurarse que puedan comunicarse e intercambiar datos. El término middleware se usa para referirse a este software que se encuentra en el centro y se compone de un conjunto de librerías que se instalan en cada computadora distribuida.

Computación cliente – servidor

Los sistemas distribuidos a los que se accede por Internet normalmente son del tipo cliente-servidor. En esta arquitectura, una aplicación se modela como un conjunto de servicios que proporciona el servidor. Los clientes pueden acceder a dichos servicios y presentar los resultados a los usuarios. Los clientes desconocen la existencia de otros clientes, pero deben identificar a los servidores que les otorgaran el servicio.

Los sistemas cliente-servidor dependen de que esté bien estructurada la información que solicitan y que aparte se ejecuten los cálculos o procesos para generar esa información. Para esto se deben diseñar de manera lógica las diferentes capas lógicas:

- Capa de presentación: se ocupa de presentar información al usuario y gestionar todas sus interacciones.
- Capa de gestión de datos: gestiona los datos que pasan hacia y desde el cliente. Puede comprobar datos y generar páginas Web, etc.
- Capa de procesamiento: para implementar la lógica de la aplicación y proporcionar la funcionalidad requerida a los usuarios finales.
- Capa de base de datos: almacena datos y ofrece servicios de gestión de transacción, etc.

Los diseñadores de sistemas distribuidos deben organizar sus diseños de sistema para encontrar un equilibrio entre rendimiento, confiabilidad, seguridad y manejabilidad del sistema. No existe un modelo universal de organización de sistemas adecuado para todas las circunstancias. Por ejemplo los siguientes **tipos de patrones de arquitectura**:

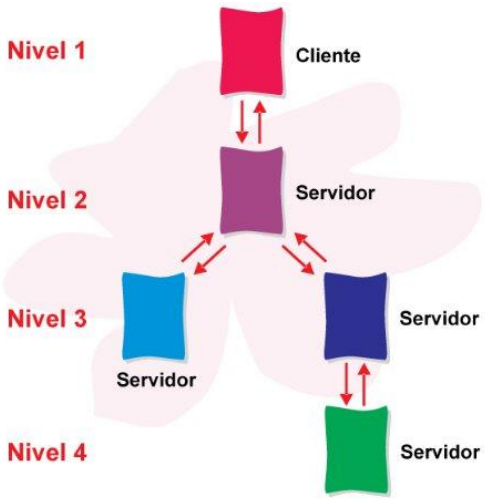
1. Arquitectura maestro-esclavo: se usan en un sistema de tiempo real donde puede haber procesadores separados asociados con la adquisición de datos del entorno

del sistema, procesamiento de datos y computación y actuador de gestión. Este modelo se usa en situaciones donde es posible predecir el procesamiento distribuido que se requiere, y en el procesamiento puede asignarse a procesadores esclavos que pueden usarse para operaciones de cómputo intensivo, como el procesamiento de señales y la administración de equipo controlado por el sistema.

2. Arquitecturas cliente – servidor de dos niveles: es la forma más simple de este tipo de arquitecturas. El sistema se implementa como un solo servidor lógico más un número indefinido de clientes que usan dicho servidor.

    a. Cliente ligero: La capa de presentación se implementa en el cliente y todas las otras capas (gestión de datos, la aplicación y base de datos) se implementan en el servidor. Generalmente se utiliza un navegador Web para presentar los datos en la computadora cliente.
    b. Cliente pesado: el procesamiento de la aplicación se realiza en el cliente. Las funciones de gestión de datos y base de datos se implementan en el servidor.

3. Arquitecturas cliente – servidor multinivel: en esta arquitectura las diferentes capas del sistema (presentación, gestión de datos, procesamiento de aplicación y base de datos) son procesos separados que se pueden ejecutar en diferentes procesadores. Por ejemplo una arquitectura cliente–servidor de tres niveles permite optimizar la transferencia de información entre el servidor Web y el servidor de base de datos. La comunicación entre éstos puede usar rápidos protocolos de intercambio de datos. Para manejar la recuperación de información de la base de datos, se usa el middleware que soporta consultas de base de datos (SQL).



El modelo cliente – servidor de tres niveles puede extenderse a una variante multinivel, donde se agregan servidores adicionales al sistema. Haciendo uso del servidor Web para los datos y servidores separados para la aplicación y servicios de base de datos. O bien cuando se requiere tener acceso a datos de diferentes bases de datos, en este caso conviene agregar un servidor de integración al sistema, para recolectar los datos distribuidos y presentarlos al servidor de aplicación como si fuera una sola base de datos.

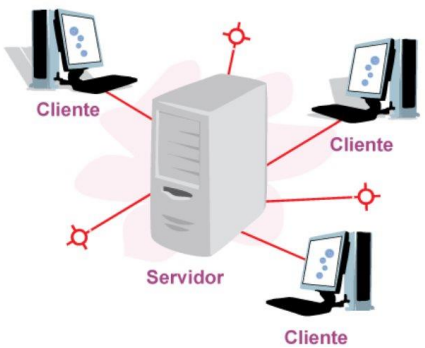
Observa la siguiente tabla que se contiene el tipo de aplicaciones que soporta cada tipo de arquitectura.

Arquitectura	Aplicación
Cliente – servidor de dos niveles con clientes ligeros	Aplicaciones de cómputo intensivo, como compiladores con poca o ninguna gestión de datos y aplicaciones web con gestión de datos.
Cliente – servidor de dos niveles con clientes pesados	Aplicaciones con software comercial por ejemplo Microsoft Excel en el cliente, aplicaciones que requieran procesamiento de datos intensivo y aplicaciones móviles con aplicaciones por Internet o red local.
Cliente – servidor multinivel	Aplicaciones con miles de clientes, con datos y aplicaciones volátiles. Se integran los datos provenientes de diferentes fuentes.

Tabla de tipos de patrones de arquitecturas

Existen otros tipos de arquitecturas que no se basan en capas, por ejemplo las arquitecturas de componentes distribuidos y arquitecturas entre pares (peer – to – peer) (Sommerville, 2011, pp. 485-495).

El diseño de la arquitectura del software es una de las vistas que describen cómo deberá ser construido. Otros aspectos del diseño de un software son los que tienen que ver con la interacción hombre–máquina y el diseño de la interfaz, estos conceptos serán abarcados en los siguientes temas.



3.1.3. La interacción hombre-máquina

En el contexto del diseño de software podemos entender a la interacción como la relación dada entre el ser humano y la máquina a través de una interface. Esta relación produce en el ser humano una extensión de sus capacidades por medio de las máquinas, gracias a esta ventaja el ser humano logra realizar tareas que antes le ocasionaban rutina y le hacían el trabajo más complicado. El término máquina dentro de este tema se refiere a computadoras, dispositivos, ordenadores incluso móviles, robots o cualquier equipo que el sistema requiera.

Entonces podemos definir a la interacción humano–máquina como el estudio de la interacción entre la gente, los ordenadores y las tareas. De esta manera comprenderás como la gente y las computadoras interactúan para lograr hacer tareas específicas. Además se atiende a la forma en que deben ser diseñados y para lograr esto observa algunos enfoques de esta área:

- Interacción hardware–software
- Diseño orientado al usuario
- Modelos mentales del usuario y su relación con el sistema.
- Funciones del sistema y su adaptación a las necesidades del usuario.
- Su impacto en la organización.

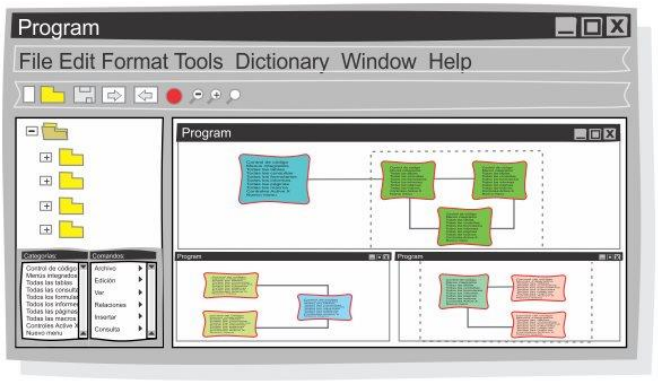
La interacción entre el humano y la computadora puede darse de varios modos (multimodal); por ejemplo la mayoría de los sistemas interactúan con un usuario a través del monitor, teclado, ratón, bocinas, etc. De esta manera se abarcan varios canales de comunicación por medio de los sentidos humanos como lo son el tacto, la vista, el oído, etc. Los sistemas actuales tienden a tener múltiples canales de comunicación de entrada/salida. Los humanos procesan información por varios canales de manera simultánea. Y dadas estas características se logra una interacción dimensional por el concepto de múltiples funciones del ser humano.

Los **estilos de interacción** más importantes son (Sabatini, 2010, p. 1):

- **Interfaz por línea de órdenes:** puede ser con teclas de función, abreviaciones cortas, palabras completas, etc.

- **Menús y formularios**: Los formularios contienen un grupo de elementos que permiten al usuario introducir datos para que sea utilizada en la aplicación, ya sea que se almacene en alguna base de datos o se utilice directamente en funciones o cálculos del mismo software. Los menús son elementos agrupados como un conjunto de opciones que el usuario puede seleccionar y al seleccionarse se espera la ejecución de alguna funcionalidad del sistema. Cuando los menús ocupan mucho espacio se puede incluir menús desplegables o menús pop-up.

- **Manipulación directa:** sistemas con acciones rápidas que provocan efectos visibles y claramente identificables en el objeto seleccionado. Por ejemplo ventanas, iconos, cursores, menús, etc.



- **Interacción asistida:** con asistentes o programas tipo agente que ayudan, leen la entrada que el usuario presenta en la interface y pueden hacer cambios en los objetos que el usuario ve en la pantalla. Los agentes son autónomos porque pueden trabajar en un segundo plano, si así se les configura. Tienen inteligencia por tener iniciativa y capacidad de adaptación a múltiples situaciones. Son de uso personal, ya que aprenden del usuario, sugieren sin imponer soluciones.

Éstos son aspectos generales de la interacción humano–computadora, a continuación verás las características que se deben tomar en cuenta para su diseño. Al desarrollar la etapa del diseño del software también se debe considerar como debe darse la interacción de los usuarios y el software para lo cual se debe diseñar su interacción, estudia el siguiente tema:

3.1.4. Diseño de la interacción

Considerando que la interacción entre el usuario y un software se realiza principalmente por medio de la interfaz, le llamamos mutuamente dependientes. Por lo tanto se abarcará este tema realzando el diseño de la interfaz.

A continuación se presentan una serie de principios relevantes para el diseño de interfaces gráficas efectivas. Se dice que una interfaz es efectiva cuando ésta oculta al usuario el funcionamiento del sistema (Tognazzini, 2003, p. 1).

- Las aplicaciones deben anticiparse a las necesidades del usuario, no esperar a que el usuario busque o recuerde información, la aplicación debe ofrecer todas las herramientas necesarias para cada etapa de su trabajo.
- Se debe evitar llenar de restricciones al usuario, dejarle cierto nivel de autonomía para que trabaje con confianza y logre el control del sistema. Pero es importante mantenerle informado del estado del sistema y tenerlo siempre visible y actualizado.
- Considerar a las personas que padecen el defecto genético del daltonismo, por lo cual no debe diseñar la transmisión de la información basada únicamente en el color.
- Diseñar una aplicación que funcione como lo espera el usuario, por ejemplo, sí un icono muestra cierta imagen, se espera que la acción realice algo relacionado con la imagen. La única manera de comprobar la consistencia es revisar las expectativas del usuario y hacer pruebas de interfaz con ellos.
- Considerar el diseño de campos de texto que contengan valores por defecto o sugeridos, éstos deben mostrarse seleccionados para que el usuario sólo tenga que teclear, borrar o escribir. Los valores que aparezcan por defecto deben tener sentido con el dominio del campo.
- Buscar la productividad del usuario y no de la máquina. El gasto más alto de un negocio es el trabajo humano. Cada vez que el usuario tiene que esperar la respuesta del sistema, es dinero perdido. Escribe mensajes de ayuda concisos y que ayuden a resolver el problema: un buen texto ayuda mucho en comprensión y eficacia. Los menús y etiquetas deben comenzar con la palabra más importante.

- No se debe encerrar a usuario en una sola ruta, dejar varias posibilidades para que ellos elijan como realizar su tarea. Permitir y facilitar que siempre pueda regresar al inicio. Diseñar las acciones reversibles para que el usuario pueda experimentar, en otras palabras permitir siempre la opción “deshacer”.
- Reducir el tiempo de espera con acciones como: efectos visuales al dar clic al botón, mostrar reloj de arena animado para acciones entre medio y dos segundos, mostrar mensaje o barra de estado en procesos que tarden más de dos segundos, indicar con alarmas o pitidos cuando termine el proceso y el usuario pueda volver a tomar el control del sistema.
- Realizar diseños intuitivos cuyo tiempo de aprendizaje por parte del usuario sea mínimo aproximándose al ideal de que los usuarios se sentaran frente al software y supieran como utilizarlo. La usabilidad y facilidad de uso no son excluyentes, debes decidir cuál es la más importante y luego implementa ambas.
- El uso de metáforas que evoquen lo familiar, pero con un nuevo punto de vista.
- Utilizar texto con alto contraste. Por ejemplo negro sobre blanco o amarillo pálido. Evitar fondos grises cuando haya texto. El tamaño de letra que sea legible en los monitores más comunes tomando en cuenta a los usuarios mayores, cuya visión suele ser peor que la de los jóvenes.
- Administrar y mostrar los estados del sistema, por ejemplo cuando el usuario es la primera vez que entra, ubicación de la navegación del usuario, a donde quiere ir, histórico de navegación en el sistema, cuando abandonó la sesión el usuario.

Después de haber conocido los principios relevantes para el diseño de interfaces conocerás la fase de codificación, la cual lleva el peso de la construcción del software, pues es donde se deben aplicar todos los modelos y principios generados en el análisis y diseño que hemos visto hasta este punto. Cualquier inconsistencia en las etapas previas a la codificación, se verán reflejadas en ésta y cualquier cambio o nuevo requerimiento identificado en esta fase involucrará modificar o añadir a lo ya previamente generado (Requerimientos definidos, diagramas de análisis, diseño, etcétera).

3.2. Codificación

Ahora veremos la fase de codificación como parte del desarrollo de software. Esta etapa implica desarrollar los programas que integrarán el software para resolver los requerimientos que el cliente nos solicitó. La manera de realizarlo normalmente es a través de módulos que al término serán integrados en una sola unidad, el principal entregable del proceso de desarrollo del software.

En esta etapa las acciones del algoritmo tienen que convertirse en instrucciones. Para codificar un algoritmo tiene que conocerse la sintaxis del lenguaje al que se va a traducir la lógica del programa que indique las acciones y el orden en que se debe ejecutar. Por lo tanto, es conveniente que todo programador aprenda a diseñar algoritmos antes de pasar a la fase de codificación.

En este tema encontrarás aspectos importantes sobre el proceso de crear código y organizarlo de tal manera que vaya obteniendo el aspecto esperado por el usuario y trazado en la etapa de diseño. Así mismo que resuelva la funcionalidad solicitada en los requerimientos del software. Comenzaremos con la traducción del diseño a código.

3.2.1. Traducción de diseño a código

En el proceso de traducción y codificación pueden aparecer inconsistencias de muchas maneras y, la interpretación equivocada de las especificaciones del diseño detallado, puede conducir a un código fuente erróneo. La complejidad o las restricciones de un lenguaje de programación pueden conducir a un código que sea difícil de probar y mantener, es decir, las características de un lenguaje de programación pueden influir en la forma de pensar.

Para poder desarrollar código se necesita un **lenguaje de programación** que es un lenguaje artificial que permite escribir instrucciones para indicar a una computadora lo que tiene que hacer. Existen diversos tipos de lenguajes que se pueden clasificar en: máquina, de bajo nivel y de alto nivel.

El lenguaje máquina es el que está basado en ceros y unos (binario) es el lenguaje natural de la computadora. El lenguaje de bajo nivel está basado en palabras reservada llamadas mnemotécnicos que ayudan a generar operaciones que serán ejecutadas por el procesador. A este tipo de lenguajes se les llama **ensambladores** y necesita un traductor para convertirlo al lenguaje máquina. Y por último, los lenguajes de alto nivel son los que se basan en un conjunto de palabras, sintaxis y reglas muy parecidas al lenguaje natural del ser humano, más fácil de programar, sin embargo también necesitará ser traducido al lenguaje máquina. De los lenguajes de alto nivel se desprenden una gran variedad de lenguajes de programación que nos llevaría otro curso comentar sus características, por lo tanto sólo se mencionan algunos: C, Java, C++, Lisp, Python, Ruby, html, javascript, prolog, etc. Para escoger seleccionar el lenguaje de programación es necesario considerar los siguientes principios (Álvarez y Arias, 2002. p. 1):

- Respecto a los aspectos generales debemos buscar un lenguaje que: sea de alto nivel, permita el uso de nombres significativos y creación de módulos y funciones, utilice estructuras de control, se puedan declarar variables y tenga métodos de manejo de error.
- Respecto a la funcionalidad, evaluar: el tamaño del proyecto, conocimiento del lenguaje por parte de los programadores, disponibilidad del software y documentación de ayuda, que funcione en varios sistemas operativos (en el caso de que el cliente lo solicite), el tipo de aplicación a desarrollar, complejidad de la lógica y estructuras de datos necesarias.

Para definir un estilo de programación principalmente se hace énfasis en la buena escritura del código, que sea auto descriptivo, lo cual se logra por medio de un adecuado uso de nombres para variables, objetos, clases, arreglos y todo elemento que necesite ser nombrado. Utilizar palabras que se relacionen con el contenido que almacenará dicho elemento, por ejemplo un arreglo que contenga los días de la semana puede llamarse "*díasDeLaSemana*" o una variable que almacenará temporalmente el resultado del total de la venta puede llamarse *"ventaTotal"*. Además debemos respetar las restricciones que las reglas del lenguaje nos impongan para nombrar a los elementos que utilicemos.

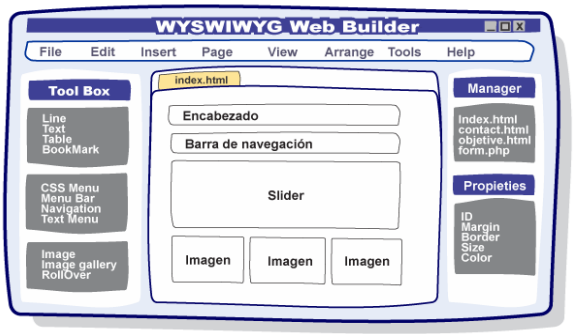
Una buena práctica de programación es aquella que utiliza los recursos críticos de forma eficiente. La eficiencia es un requisito de todo sistema y se debe planear considerando los siguientes aspectos: hacer expresiones lógicas y aritméticas más simples, revisar si las sentencias que están repitiéndose en los ciclos están correctas y deben repetirse. Evitar el uso excesivo de arreglos multidimensionales, de apuntadores y listas complejas. No mezclar tipos de datos, aunque el lenguaje lo permita, evitar declarar variables que no sean utilizadas, minimizar el uso de peticiones de entrada / salida (E/S) y considerar si algunos de los dispositivos de E/S puede afectar la calidad o velocidad.

Como te has dado cuenta este proceso del paso del diseño a la codificación debe ser debidamente organizado. Ahora conocerás las características necesarias para codificar la interfaz.

3.2.2. Codificación de la interfaz

La codificación de la interfaz se refiere a la implementación del diseño y a la aplicación de los estándares seleccionados para tal fin. El aprovechamiento de las nuevas tecnologías nos facilita la velocidad de desarrollo, ya sea por el uso de generadores automáticos de interfaces, lenguajes de POO que permiten construir plantillas padre y heredar sus atributos a diferentes plantillas hijo, permitiendo que los cambios sean más sencillos de realizar.

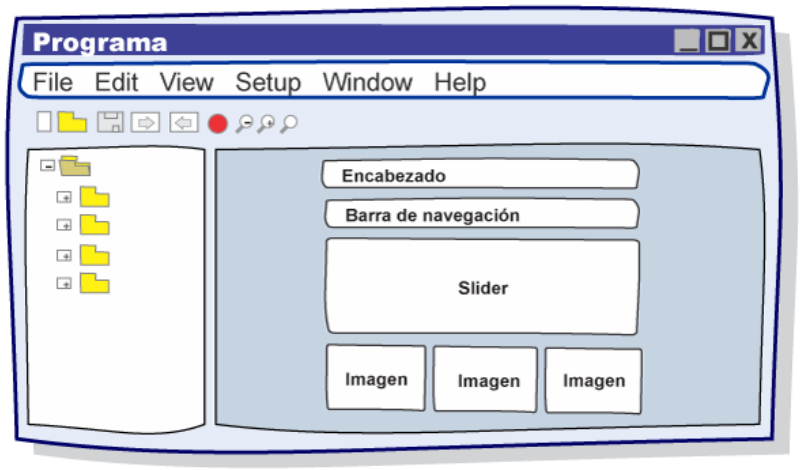
En el ámbito *web* encontramos herramientas de desarrollo tipo WYSIWYG es el acrónimo de *What You See Is What You Get* (en español, "lo que ves es lo que obtienes"); que ayudan al desarrollador a



construir dinámicamente las interfaces para internet (Sánchez, 2005, p. 1).

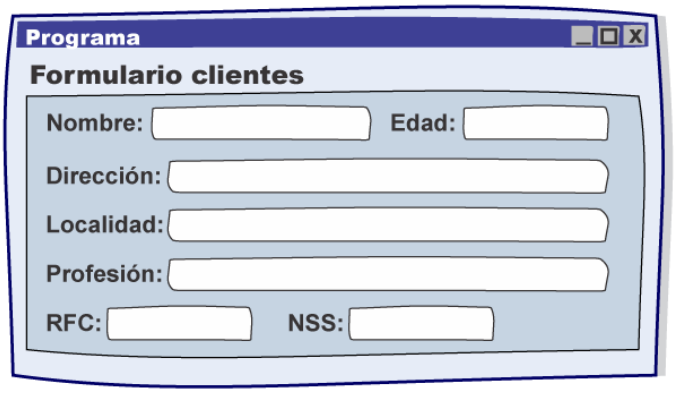
Cada organización establece su propio proceso de **construcción de interfaces** que en general podría ser similar al siguiente:
Codificación de pantallas: para facilitar la interacción entre el usuario y las funciones del sistema. Los elementos que deben crearse son los menús, encabezados, área de mensajes, sección del contenido, estándares de diseño.

Respecto al **contenido** de las pantallas éste debe organizar: la información que se mostrará, las validaciones ante posibles errores del usuario, los datos y sus relaciones, la navegación entre ventanas.



Generalmente se realizan formularios para introducir datos provenientes del usuario y por ello es importante definir de una manera más precisa a las **validaciones**. Éstas se refieren a la comparación de un valor esperado con el conjunto de valores que son permitidos para cada campo. Por ejemplo la validación de **tipo** revisará que el valor sea numérico, cadena o fecha, según el que se haya asignado para el dato.

Otra validación puede ser la **longitud**, no deberá permitir valores que sean menores o mayores a los esperados. Y por último la validación de **caracteres especiales** no permitir que se capturen caracteres que no sean los esperados. Si no activamos este proceso de validación estaremos permitiendo que el usuario introduzca datos imprecisos que se verán reflejados en la salida de información poco confiable.



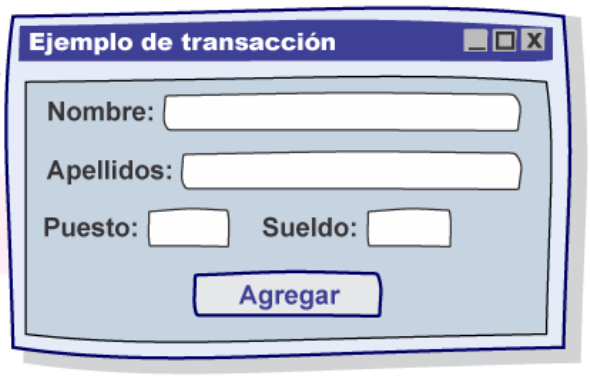
Y en el tema de **información**, ésta deberá ser presentada sólo para los usuarios que tengan la facultad para tenerla. Lo cual implica que deberá programarse niveles de seguridad en el software, habilitando y limitando funciones de acuerdo a cada tipo de usuario definido.

La **programación** abarcará también aspectos gráficos que contienen elementos que le darán una imagen estética al software y serán construidos con una adecuada distribución de colores, texto, imágenes, etc.

Y por último la manera de **navegar** a través del software generado, ésta y todos los elementos anteriores deben planearse estratégicamente en la etapa de diseño. En esta etapa de codificación únicamente se encargará de implementarlo con código añadiendo la funcionalidad previamente indicada.

Como podrás darte cuenta es necesario implementar un estándar para facilitar el proceso de codificación, tanto en el diseño de interfaces como en la forma de escribir el código. A continuación veremos estrategias para administrar los estándares dentro de la organización del desarrollo del software.

```
<script languaje "javascript" type= "text/javascript">
;(function() {
var oValidator = yahoo.extension.validator (´test-form´
notifyType:´tips´, // default tips
stopOnFirst: false,
imageBase: ´../src/img/´,
onSubmit: true, //default true, intervene the on submit
checkOnBlur: true, //default true, check an input
hideSuccess: false // default false
)};
oValidator.addRules({
´test-form1-name´ : {
desc: ´xx´,
maxLength: 15,
minLength: 6
```



3.2.3. Herramientas de desarrollo: gestión de la configuración

Como ya habrás notado hasta este punto, son muchos procesos y demasiada información que la que se genera durante las etapas del desarrollo de software, lo que hace necesario la gestión de la configuración, por lo que es fácil perderse cuando existe un cambio o se generan nuevas versiones por actualización (Sommerville, 2011, p. 682).

La administración de la configuración CM *(Configuration Management*) gestiona las políticas, procesos y las herramientas para administrar los elementos cambiantes del desarrollo de software. CM es útil para el trabajo individual y en equipo. Al implementar una administración configurada se obtendrán los siguientes alcances:

1. Administración del cambio: cuando se encuentra un cambio ya sea por parte del equipo o del cliente es necesario realizar el análisis de su impacto, estimar costos, tiempos y decidir si se implementan o no.
2. Gestión de versiones: se refiere al seguimiento de las versiones de los elementos del sistema y garantizar que los cambios o actualizaciones realizadas por diferentes miembros del equipo no interfieran entre sí.
3. Construcción del sistema: consiste en ensamblar los componentes del sistema, datos y librerías y luego compilarlos para generar el ejecutable.
4. Gestión de entregas: Implica preparar el software para la entrega y hacer un control de versiones de lo que se entregó al cliente.

La **administración de la configuración** implica una gran cantidad de información, por lo cual se han generado muchas herramientas de gestión de documentos y administración de la configuración.

Las **herramientas de software** que existen en el mercado para administrar versiones son:

CVS: es una herramienta famosa por ser de uso libre, permite generar repositorios para administrar los documentos que se generen en los proyectos. Favorece la administración de versiones, por ejemplo cuando se ha creado un archivo no podrá renombrarse ni sobreescribirse, cualquier cambio se guarda en una nueva versión, conservando las versiones anteriores.

- **Subversión**: es una herramienta de uso libre, fue creada para reemplazar al CVS, y también funciona para llevar el control de las versiones de los documentos. Genera una versión a nivel proyecto.
- **ClearCase**: ayuda a facilitar el proceso del cambio, permite administrar los documentos de manera confiable, flexible, es óptimo para equipos de desarrollo grandes. Funciona en sistemas operativos Linux, Windows, Unix.
- **Darcs**: es una herramienta de control de versiones para ambientes distribuidos. Permite la generación de puntos de restauración para restablecer documentos con versiones anteriores. Es muy útil para los archivos que generan los desarrolladores de software,

mantienen las versiones y se registran a los autores de los cambios. Tiene una interfaz muy sencilla de usar.

- **Plastic SCM**: es una herramienta de control de versiones para desarrollo distribuido, funciona para sistemas operativos Windows, GNU/Linux Solaris, Mac OS X, .Net/Mono. Es útil para equipos grandes y ayudan a comprender el estado del proyecto. Su principal característica es que simplifica la ramificación y fusión.

Establecer una adecuada estructura de trabajo con los estándares, políticas y herramientas de gestión es muy importante para el logro del buen desempeño del equipo de desarrollo. Facilitar a los desarrolladores herramientas que hagan su trabajo más sencillo y les agilice su labor, permitirá que se puedan organizar la producción generada evitando pérdidas de información, duplicidad de archivos y otros errores.

Las etapas que faltan por abarcar son la de pruebas y las de mantenimiento. Es importante marcar la diferencia entre las etapas de codificación y de pruebas, cada etapa implica el desarrollo de código sin embargo se corre el riesgo de que la etapa de codificación se extienda absorbiendo el tiempo de pruebas, lo cual se convertirá un problema si se lanza un producto que no fue probado adecuadamente.

Asimismo la etapa de mantenimiento involucra codificar para realizar ajustes al software o la generación de código nuevo para atender nuevas solicitudes de requerimientos. Es importante que cada etapa tenga sus propios tiempos asignados y se respeten las actividades planeadas para cada fase. Observa los siguientes temas para que consideres las características de estas etapas.

3.3. Pruebas y mantenimiento

A continuación verás el tema de pruebas, sus características y tipos de pruebas que se pueden aplicar al desarrollo del software. Además se incluyen sugerencias de herramientas de software para administrar los documentos de pruebas, así como herramientas que se utilizan para probar el código generado.

Las pruebas tienen el objetivo de demostrar que un programa hace lo que se espera que haga y descubrir los defectos que el programa tenga antes de usarlo. Al probar el programa, éste se ejecuta con datos de ejemplo para verificar los resultados y revisar errores en la información que arroje el programa. El proceso de prueba tiene 2 metas:

1. Demostrar al desarrollador y al cliente que el software cumple con los requerimientos.
2. Encontrar situaciones vulnerables del software para evitar caídas del sistema, cálculos incorrectos o corrupción de datos.

Las **pruebas** no pueden demostrar que el software esté exento de defectos o que se comportará de alguna manera específica en un momento dado. No es posible dejar con cero defectos ya que es posible que el software no sea probado en todas las situaciones probables. Dentro del proceso de pruebas existen dos procesos que normalmente se confunden: **verificación y validación**. La validación contesta la pregunta de ¿construimos el producto correcto? Y la verificación responde a la pregunta ¿construimos bien el producto?

La finalidad de la verificación es revisar que el software cumpla con la funcionalidad esperada. La validación es un proceso más general, su meta es que el software cumpla con las expectativas del cliente. Además también existen los procesos de inspección que se enfocan principalmente al código fuente de un sistema. Cuando el sistema se inspecciona se utiliza el conocimiento del sistema, del dominio de aplicación y del lenguaje de programación para descubrir errores. Las inspecciones no substituyen las pruebas de software, ya que no son eficaces para descubrir defectos que surjan por interacciones entre diferentes partes de un programa (Sommerville, 2011, pp. 206-207).

A continuación verás una clasificación de los tipos de pruebas, así como las herramientas que se utilizan para administrarlas e incluso probar código, principalmente tenemos pruebas enfocadas a los componentes que conforman al software y pruebas enfocadas al software integrado.

3.3.1. Tipos de pruebas y herramientas

¿Por qué aplicar pruebas al código? Principalmente porque el código es un modelo basado en un modelo humano, que además es construido por humanos, por lo que está expuesto a la generación de errores y fallos imperceptibles. Inspeccionar adecuadamente el producto puede ser la diferencia entre un software con calidad y uno que no la tiene; por lo que realizar pruebas al software, es un proceso obligado para todo equipo de desarrollo; así como inspeccionar la calidad del producto durante el proceso de desarrollo, involucra diversas tareas que tienen relación con el tipo de pruebas que se hayan decidido aplicar.

Las características del software nos indicarán que pruebas y cuando deberán aplicarse. Sommerville (2011, pp. 210-230) lo define de la siguiente manera:

Tipos de prueba

Pruebas de desarrollo: son un proceso de prueba obligado, cuyo objetivo consiste en descubrir errores en el software, generalmente están entrelazadas con la depuración: localizar problemas con el código y cambiar programas para corregirlos.

Pruebas de unidad: para probar programas o clases. Sirven para comprobar la funcionalidad de objetos o métodos.

- Pruebas del componente: se prueban las interfaces de los componentes.
- Pruebas del sistema: se prueba la integración de todos los componentes.

Pruebas de unidad: se encarga de probar componentes del programa, como métodos o clases de objetos. Se tienen que diseñar pruebas para revisar todas las operaciones

asociadas, establecer y verificar el valor de todos los atributos y poner el objeto en todos los estados posibles.

Pruebas de componentes: se enfoca en demostrar que la interfaz de componente se comporte según la especificación. Existen diferentes tipos de interfaz entre componentes de programa y, en consecuencia, distintos tipos de error de interfaz. Por ejemplo:

- Uso incorrecto de la interfaz.
- Mala interpretación de la interfaz.
- Errores de temporización.

Pruebas del sistema: incluyen la integración de los componentes y luego probar el sistema completamente integrado. La prueba del sistema es un proceso colectivo más que individual. En algunas compañías, las pruebas del sistema implican un equipo de prueba independiente, sin incluir diseñadores ni programadores.

Pruebas de versión: sirven para poner a prueba una versión particular de un sistema que se pretende usar fuera del equipo de desarrollo. Generalmente esta versión es para clientes y usuarios, sin embargo en proyectos muy grandes, una versión podría ser para otros equipos que desarrollan sistemas relacionados. La principal meta de este tipo de pruebas es convencer al proveedor del sistema de que: éste es suficientemente apto para su uso. Si es así puede liberarse como producto o entregarse al cliente.

Pruebas basadas en requerimientos: Estas pruebas son de validación más que defectos, se intenta demostrar que el sistema implementó adecuadamente sus requerimientos. Para esto es necesario escribir muchas pruebas para garantizar que cubrió los requerimientos.

Pruebas de escenario: son similares a las de versión, donde se crean escenarios que generalmente suceden y se les utiliza en el desarrollo de casos de prueba para el sistema. Un escenario es una historia que describe cómo puede utilizarse el sistema. Éstos deben ser realistas con usuarios reales. Si los escenarios fueron parte de los elementos de tus requerimientos, entonces podría utilizarlos como escenarios de prueba.

Pruebas de rendimiento: Se aplican cuando el sistema ha sido integrado completamente, con ellas es posible probar el rendimiento y confiabilidad. Generalmente consisten en aumentar la carga hasta que el sistema se vuelve inaceptable. Una manera de descubrir defectos es diseñar pruebas sobre los límites del sistema. En las pruebas de rendimiento, significa estresar el sistema al hacer demandas que estén fuera de los límites del diseño del software. Esto se conoce como “prueba de esfuerzo”.

Pruebas de usuario: Este tipo de pruebas son necesarias por la influencia del entorno de trabajo que tiene gran efecto sobre la fiabilidad, el rendimiento, el uso y la robustez de un sistema. En la práctica existen tres tipos de pruebas de usuario:

- **Pruebas alfa:** las realiza el usuario en presencia de personal de desarrollo del proyecto haciendo uso de una máquina preparada para tal fin.
- **Pruebas beta:** las realiza el usuario después de que el equipo de desarrollo les entregue una versión casi definitiva del producto.
- **Pruebas de aceptación**: son las únicas pruebas que son realizadas por los usuarios, deciden si está o no listo para ser aceptado.

Dependiendo del tipo de prueba que se aplique, será la dificultad o sencillez (para lograr que las pruebas sean más sencillas se pueden utilizar herramientas para probar código). Si utilizas software para administrar las pruebas podrás llevar un control más preciso sobre ellas, pues existen herramientas para desarrollar pruebas de software en todas las etapas del desarrollo del sistema, éstas soportan las pruebas y el desarrollo a lo largo del ciclo de vida, desde la validación de requerimientos hasta el soporte del funcionamiento del sistema. Otras herramientas se enfocan en una sola parte del ciclo de vida. Por ejemplo a continuación veremos herramientas de software cuya función va orientada al tipo de aplicación (Fernández, 2012, p. 1).

HP *Unified Functional Testing* es útil para probar la interfaz gráfica de las aplicaciones de Java, Sap, Siebel, Visual Basic, .Net, Oracle, etc. Ayuda a la generación de las pruebas y su administración. Además de ser útil para probar interfaz gráfica, sirve para otros módulos. Se pueden crear escenarios de prueba. Ofrece un entorno de trabajo sencillo de utilizar.

Selenium es una herramienta útil para probar aplicaciones web, desarrolladas en C#, Java, Groovy, Perl, PHP, Python y Ruby. Es multiplataforma ya que se puede usar en Windows, Linux y MacOS, es gratis y opensource (código fuente abierto). Además se puede encontrar mucha documentación en Internet sobre cómo utilizarla.

Eggplant es una herramienta que se puede aplicar en diferentes plataformas como Windows, MacOS y Linux, en otras palabras, no depende de ninguna ya que convierte a la pantalla en imagen y aplicando la tecnología de reconocimiento óptico de caracteres (OCR) puede identificar imágenes y texto para su interpretación. Su ambiente de trabajo es fácil de utilizar y ayuda en la automatización de pruebas de pantalla por medio de la comparación.

Ranorex es exclusiva para plataformas Windows, es capaz de identificar todos los elementos de una aplicación escritorio o web. Se basa en el reconocimiento de objetos y tiene un ambiente de trabajo sencillo y fácil de aprender. Es un medio para desarrollar y administrar desde las pruebas unitarias hasta los proyectos de prueba. Y permite la depuración del código ya que cuenta con un editor.

TestComplete es una herramienta muy completa para hacer pruebas de software, pero únicamente para la plataforma Windows. Los tipos de pruebas que se pueden gestionar son de regresión, de cargas web, unitarias, etc. Su interfaz de desarrollo es muy completa y se integra con las herramientas de Microsoft Visual Studio. Se pueden probar las tecnologías de Visual Basic, Delphi, C++ y otras.

Microsoft Test Manager principalmente es una herramienta para la administración y automatización de pruebas y puede integrarse al kit de Microsoft Visual Studio. Los tipos de pruebas que puede gestionar son unitarias, de interfaz de usuario, de base de datos, de carga y genéricas. Se debe utilizar otra herramienta *Team Fundation Server* para almacenar casos de pruebas y requerimientos. Es para la plataforma Windows exclusivamente.

3.3.2. Mantenimiento

Haber concluido con un proyecto de desarrollo de software, no implica que el software haya llegado a su final, ya que como todo proceso, el paso del tiempo producirá mejoras ya sea por iniciativa, cambios tecnológicos o por situaciones externas (gobierno, instituciones, organizaciones), produciendo mejoras al software ya sea modificando lo existente o agregando nuevos requerimientos. En este tema analizaremos el proceso de mantenimiento al software.

El proceso de **mantenimiento** consiste en cambiar un sistema después de que éste se entregó; generalmente se aplica a software hecho a la medida y los cambios consisten desde la simple corrección de errores, hasta mejoras significativas para corregir errores de especificación o bien, agregar nuevos requerimientos. Existen tres tipos de mantenimiento de software:

1. **Reparación de fallas:** errores de codificación, diseño o requerimientos, siendo estos últimos los más costosos de corregir.
2. **Adaptación ambiental:** es necesario cuando algún aspecto del entorno del sistema como hardware, plataforma operativa o algún otro elemento hacer cambiar al software. El sistema debe modificarse para que se adapte a dichos cambios.
3. **Adición de funcionalidad:** este tipo de mantenimiento es necesario cuando los requerimientos funcionales cambian por algún factor organizacional o empresarial. Generalmente este tipo de cambios es más grande que los anteriores.

Como puedes darte cuenta, la **adaptación** es una palabra clave en el proceso del mantenimiento al software; ya que lo que se busca es hacer que éste se adapte cada vez más a la solución de las necesidades del ser humano, por ello desde la reparación de fallas, los ajustes al contexto ambiental en el que se encuentra o simplemente agregarle mayor funcionalidad, son actividades que convertirán al software en una verdadera herramienta de trabajo para la humanidad.

Cierre de la unidad

En el transcurso de la unidad has podido ver las etapas de diseño, codificación y pruebas que corresponden al proceso de desarrollo de software. También conociste aspectos que tienen relación con el mantenimiento al software y los tipos que existen.

Estos conocimientos te serán de ayuda para entender el proceso de desarrollo de software y podrás aplicarlos en tu ciclo personal de construcción de software, sí es que trabajas de manera independiente.

Si formas parte de un equipo de desarrollo, o en algún momento formarás parte de alguno, podrás ver reflejadas estas etapas del proceso del software y los roles que aplican para cada fase. Estos temas te dan la visión de lo que implica construir software de una manera organizada y más certera al entender los conceptos que la ingeniería de software propone.

Para saber más

Ejemplos de Interacción humano-máquina. Contiene ilustraciones y videos de ejemplos de la interacción humano-máquina:

- **http://www.interfacemindbraincomputer.wikifoundry.com/sitemap**
- **http://www.interfacemindbraincomputer.wikifoundry.com/**

Visita la siguiente página, encontrarás una plantilla de uso libre para realizar casos de prueba. Se explica detalladamente cómo se realiza su llenado:

- http://readyset.tigris.org/nonav/es/templates/test-case-format

Fuentes de consulta

Bibliografía básica

- Kendall, K. E. y Kendall, J. E. (2011). *Análisis y diseño de sistemas*. México: Pearson Educación.
- Pressman, R. (2010). *Ingeniería de software. Un enfoque práctico.* Madrid: McGraw-Hill Interamericana.
- Sommerville, I. (2011). *Ingeniería de software*. México: Pearson Educación.

Bibliografía complementaria

- Álvarez J. y Arias M. (2002). *Fase de implementación*. UNED. Recuperado de: http://www.ia.uned.es/ia/asignaturas/adms/GuiaDidactica.pdf

- Fernández, A. (2012). *Comparativa de herramientas para pruebas automáticas.* Globe Testing. España. Recuperado de: http://www.globetesting.com/2012/03/comparativa-de-herramientas-para-pruebas-automaticas/

- Sabatini, A. (2017). *Interacción humano-máquina*. Buenos Aires. Recuperado de: http://www.interfacemindbraincomputer.wikifoundry.com/sitemap

- Sánchez, J. (2005). *Metodología para el diseño y desarrollo de interfaces de usuario*. Recuperado de: http://pegasus.javeriana.edu.co/~fwj2ee/descargas/metodologia(v0.1).pdf

- Tognazzini, B. (2003). *First Principles of Interaction Design*. Etados Unidos: Tomado de: http://galinus.com/es/articulos/principios-diseno-de-interaccion.html