Universidad Nacional del Nordeste
Facultad de Ciencias Exactas y Naturales y Agrimensura

FACENA – UNNE
Licenciatura en Sistemas de Información

TALLER DE PROGRAMACION II

MATERIAL TEORICO

AÑO 2014

INDICE

TEMA I: INTRODUCCION

Introducción. Tipos de errores: errores de sintaxis, errores en tiempo de ejecución, errores lógicos. Tratamiento de excepciones. Herramientas de depuración. Caso práctico.

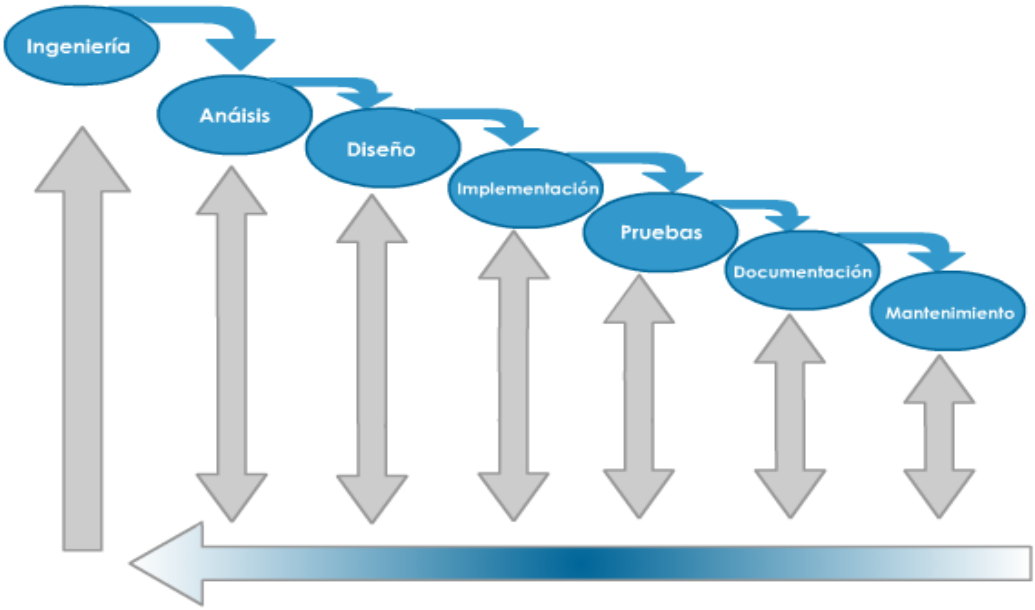
Ciclo de vida del software. Introducción. Etapas.

La ISO, International Organization for Standardization, en su norma 12207, define al **ciclo de vida de un software** como un marco de referencia que contiene las actividades y las tareas involucradas en el desarrollo, la explotación y el mantenimiento de un producto software, abarcando desde la definición hasta la finalización de su uso.

Existen varias versiones del ciclo de vida del software entre las cuales se destacan: el ciclo de vida clásico o en cascada, el modelo en espiral, el desarrollo de prototipos, el modelo por incrementos y el modelo extremo.

Etapas del ciclo de vida del software

El *ciclo de vida clásico* del software siendo uno de los más utilizados tal como lo plantean diferentes autores, está conformado en su versión ampliada por siete etapas que se pueden representar mediante un modelo en cascada así:



- INGENIERÍA DE SISTEMAS: En esta etapa el analista luego de un minucioso y detallado estudio de los sistemas de una organización, detecta un problema o una necesidad que para su solución y/o satisfacción es necesario realizar un desarrollo de software.

- ANÁLISIS: En esta etapa se debe entender y comprender de forma detallada cual es la problemática a resolver, verificando el entorno en el cual se encuentra dicho problema, de tal manera que se obtenga la información necesaria y suficiente para afrontar su respectiva solución. Esta etapa es conocida como la del QUÉ se va a solucionar.

Ejemplo

Sin entrar en el campo de la informática, para hacer la nómina de los mejores alumnos de una carrera, se necesita saber:
ENTRADA: Los datos de cada uno de los alumnos y si estos datos están en papel o en un fichero donde está toda la información de los alumnos.

PROCESO: La fórmula matemática para calcular el promedio de notas es:

(nota 1 + nota 2 + nota 3 + ...+ nota n) / cantidad de notas

SALIDA: El modelo del informe donde se desean imprimir el promedio de los alumnos.

- DISEÑO: Una vez que se tiene la suficiente información del problema a solucionar, es importante determinar la estrategia que se va a utilizar para resolver el problema. Esta etapa es conocida bajo el CÓMO se va a solucionar.

- IMPLEMENTACIÓN: partiendo del análisis y diseño de la solución, en esta etapa se procede a desarrollar el correspondiente programa que solucione el problema mediante el uso de una herramienta computacional determinada.

- PRUEBAS: Los errores humanos dentro de la programación de los computadores son muchos y aumentan considerablemente con la complejidad del problema. Cuando se termina de escribir un programa de computador, es necesario realizar las debidas pruebas que garanticen el correcto funcionamiento de dicho programa bajo el mayor número de situaciones posibles a las que se pueda enfrentar.

- DOCUMENTACIÓN: Es la guía o comunicación escrita en sus diferentes formas, ya sea en enunciados, procedimientos, dibujos o diagramas que se hace sobre el desarrollo de un programa. La importancia de la documentación radica en que a menudo un programa escrito

por una persona, es modificado por otra. Por ello la documentación sirve para ayudar a comprender o usar un programa o para facilitar futuras modificaciones (mantenimiento).

- MANTENIMIENTO: una vez instalado un programa y puesto en marcha para realizar la solución del problema previamente planteado o satisfacer una determinada necesidad, es importante mantener una estructura de actualización, verificación y validación que permitan a dicho programa ser útil y mantenerse actualizado según las necesidades o requerimientos planteados durante su vida útil. Para realizar un adecuado mantenimiento, es necesario contar con una buena documentación del mismo.

Análisis y Diseño de algoritmos

Es importante tener unos conceptos claros y precisos de lo que es el Análisis y el Diseño de Algoritmos:

1. Partiendo de un conocimiento básico de informática, se podría afirmar que el **Análisis de Algoritmos** es el conjunto de reglas y estrategias que permiten verificar el correcto y eficiente desarrollo de un algoritmo; Es importante tener claro la diferencia que hay entre el análisis del software y el análisis de un algoritmo. Como se presentó en el ciclo de vida del software, el objetivo de la etapa de análisis es comprender un problema bajo el entorno en que se desenvuelve para buscar su solución, mientras que analizar un algoritmo está relacionado con evaluar su desempeño y eficiencia con respecto a los datos de entrada. Teniendo claros estos conceptos se puede proceder a enmarcar el entorno en el cual se desenvuelve el análisis de algoritmos dentro del proceso de desarrollo de software.

   Dentro del ciclo de vida del software en la etapa de diseño es cuando se determina la estrategia y los algoritmos mediante los cuales se solucionará el problema, siendo allí el lugar apropiado para que en el momento de desarrollar o seleccionar un algoritmo, este cumpla con las necesidades planteadas en el problema y además haga un uso racional de los recursos computacionales con que dispone.

   Es importante tener en cuenta que desde el punto de vista cronológico el diseño de algoritmos es una etapa previa al análisis de los mismos; esto se debe a que es necesario primero tener el algoritmo (llegar a la solución de un problema) antes de poder evaluar el desempeño y uso de recursos del mismo.

2. De forma análoga al análisis de algoritmos, el **Diseño de Algoritmos** se puede definir como los diferentes pasos de una metodología que se deben aplicar para encontrar la solución a un problema. Esta característica es la que permite diferenciar el proceso de hacer programas de una manera empírica, a su desarrollo sistematizado bajo un conjunto de pautas, guías y principios claros (por esto se le denomina Ingeniería del Software), siendo uno de los objetivos del diseño de algoritmos el degarantizar que mediante un conjunto de estrategias pueda encontrarse una solución al problema planteado. Bajo estos conceptos, e ldiseño de algoritmos se enmarca también plenamente en la etapa de diseño de software bajo la premisa de servir como

estrategia básica para el desarrollo de los algoritmos que se requieren para el diseño de la solución global del problema.

Documentación de los programas.

El objetivo de la documentación de programas es familiarizar a analistas y programadores con lo que hace cada programa en particular.
La documentación de programas es una extensión de la documentación del sistema. El programador convierte las especificaciones de programas en lenguaje de computador. El programador deberá trabajar conjuntamente con las especificaciones de programas y asegurarse que el programa cumpla con las mismas. Cualquier cambio que surja como resultado de la programación, deberá ser expuesto y aceptado antes de aplicar el cambio.

La documentación se compone de *tres partes*:

a. **Documentación Interna**: Son los comentarios o mensajes que se añaden al código fuente para hacer más claro el entendimiento de los procesos que lo conforman, incluyendo las precondiciones y las poscondiciones de cada función.

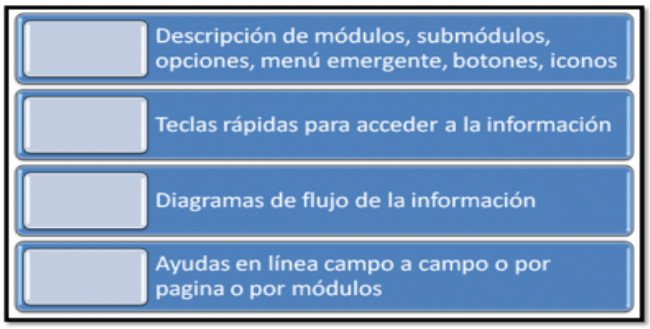
b. **Documentación Externa**: Se define en un documento escrito con los siguientes puntos:
Descripción del Problema
Datos del Autor
Algoritmo (diagrama de flujo o Pseudocódigo)
Diccionario de Datos
Código Fuente (programa)

c. **Manual de Usuario**: Describe paso a paso la manera como funciona el programa, con el fin de que el usuario lo pueda manejar para que obtenga el resultado deseado.

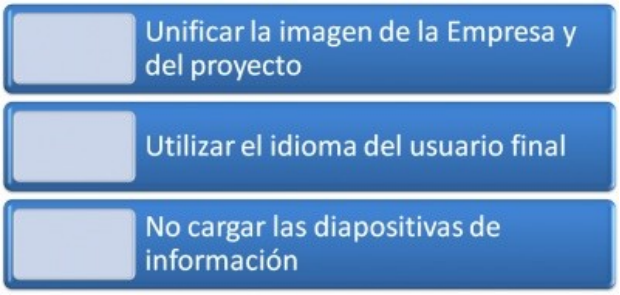
Escenarios de Pruebas. Documentos funcionales y de capacitación

Manual de Usuario:

Manual de Capacitación:



Fase Cierre del Proyecto: Acta de cierre:



Métodos de desarrollo del software

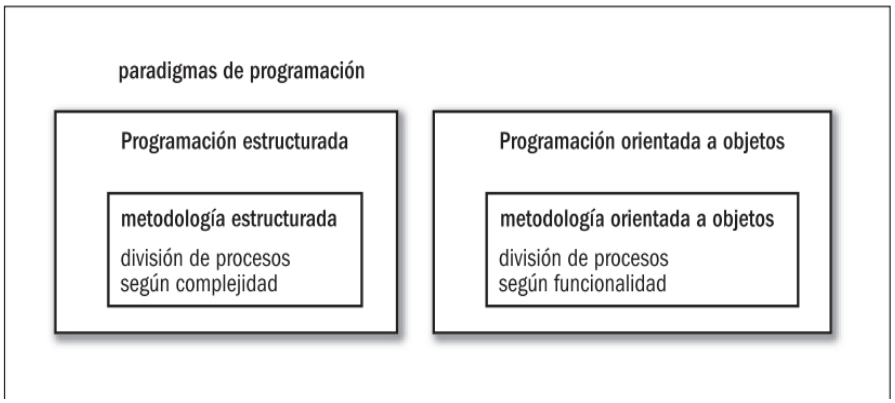
La finalidad de una metodología es prolijidad, corrección y control en cada etapa del desarrollo de un programa, lo que nos permitirá una forma sistemática para poder obtener un producto correcto y libre de errores.

Clasificación de las metodologías

Existen dos metodologías que tienen analogía en la práctica con los **paradigmas de programación**. Metodología estructurada y metodología orientada a objetos.

Metodología estructurada: la orientación de esta metodología se dirige hacia los procesos que intervienen en el sistema a desarrollar, es decir, cada función a realizar por el sistema se descompone en pequeños módulos individuales. Es más fácil resolver problemas pequeños, y luego unir cada una de las soluciones, que abordar un problema grande.

Metodología orientada a objetos: a diferencia de la metodología mencionada anteriormente, ésta no comprende los procesos como funciones, sino que arma módulos basados en componentes, es decir, cada componente es independiente del otro. Esto nos permite que el código sea reutilizable. Es más fácil de mantener porque los cambios están localizados en ada unos de estos componentes.



Diagramas del análisis estructurado

- Modelado de las Funciones del Sistema.
  - Diagrama de Flujo de Datos.
- Modelado de Datos Almacenados.
  - Diagrama Entidad - Relación.
- Modelado del Comportamiento Dependiente del Tiempo.
  - El Diagrama de Transición de Estados.
- Modelado de la Estructura de los Programas.
  - El Diagrama de Estructura.

Diagramas del análisis orientado a objetos

- Modelo de Objetos
  - Detección de clases candidatas
  - Diccionario
  - Detección de relaciones entre clases

- Modelo Dinámico
  - Determinar los Escenarios
  - Traza de sucesos de evento en particular.

- Modelo funcional: describe los cálculos existentes del sistema sin describir ni cómo ni cuándo se calculan.

  El *modelo funcional* especifica lo que sucede, el *modelo dinámico* especifica cuándo sucede y el *modelo de objetos* especifica a qué le sucede.

Lenguajes de programación

Los diferentes lenguajes de programación para la web.

Actualmente existen diferentes lenguajes de programación para desarrollar en la web, estos han ido surgiendo debido a las tendencias y necesidades de las plataformas. Desde los inicios de Internet, fueron surgiendo diferentes demandas por los usuarios y se dieron soluciones mediante lenguajes estáticos. A medida que paso el tiempo, las tecnologías fueron desarrollándose y surgieron nuevos problemas a dar solución. Esto dio lugar a desarrollar lenguajes de programación para las web dinámicas, que permitieran interactuar con los usuarios y utilizaran Bases de Datos. A continuación los lenguajes de programación para la web más utilizados.

Lenguaje HTML

Desde el surgimiento de internet se han publicado sitios web gracias al lenguaje HTML. Es un lenguaje estático para el desarrollo de sitios web (acrónimo en inglés de HyperText Markup Language, en español Lenguaje de Marcas Hipertextuales). Desarrollado por el World Wide Web Consortium (W3C). Los archivos pueden tener las extensiones (htm o html).

Sintaxis:

<html> (Inicio del documento HTML)

<head> (Cabecera)

</head>

<body>(Cuerpo)

</body>

</html>

  Negrita

  Definir párrafo

<etiqueta> Apertura de la etiqueta

</etiqueta> Cierre de la etiqueta

Ventajas:

- Sencillo que permite describir hipertexto.
- Texto presentado de forma estructurada y agradable.
- No necesita de grandes conocimientos cuando se cuenta con un editor de páginas web o WYSIWYG.
- Archivos pequeños.
- Despliegue rápido.
- Lenguaje de fácil aprendizaje.
- Lo admiten todos los exploradores.

Desventajas:

- Lenguaje estático.
- La interpretación de cada navegador puede ser diferente.
- Guarda muchas etiquetas que pueden convertirse en "basura" y dificultan la corrección.
- El diseño es más lento.

- Las etiquetas son muy limitadas.

Lenguaje Javascript

Este es un lenguaje interpretado, no requiere compilación. Fue creado por Brendan Eich en la empresa Netscape Communications. Utilizado principalmente en páginas web. Es similar a Java, aunque no es un lenguaje orientado a objetos, el mismo no dispone de herencias. La mayoría de los navegadores en sus últimas versiones interpretan código Javascript.

El código Javascript puede ser integrado dentro de nuestras páginas web. Para evitar incompatibilidades el World Wide Web Consortium (W3C) diseño un estándar denominado DOM (en inglés Document Object Model, en su traducción al español Modelo de Objetos del Documento).

Sintaxis:

<script type="text/javascript"> ... </script>

Ventajas:

- Lenguaje de scripting seguro y fiable.
- Los script tienen capacidades limitadas, por razones de seguridad.
- El código Javascript se ejecuta en el cliente.

Desventajas:

- Código visible por cualquier usuario.
- El código debe descargarse completamente.
- Puede poner en riesgo la seguridad del sitio, con el actual problema llamado XSS (significa en inglés Cross Site Scripting renombrado a XSS por su similitud con las hojas de estilo CSS).

Lenguaje PHP

Es un lenguaje de programación utilizado para la creación de sitio web. PHP es un acrónimo recursivo que significa “PHP Hypertext Pre-processor”, (inicialmente se llamó Personal Home Page). Surgió en 1995, desarrollado por PHP Group. PHP es un lenguaje de script interpretado en el lado del servidor utilizado para la generación de páginas web dinámicas, embebidas en páginas HTML y ejecutadas en el servidor. PHP no necesita ser compilado para ejecutarse. Para su funcionamiento necesita tener instalado Apache o IIS con las librerías de PHP. La mayor parte de su sintaxis ha sido tomada de C, Java y Perl con algunas características específicas. Los archivos cuentan con la extensión (php).

Sintaxis: La sintaxis utilizada para incorporar código PHP es la siguiente:

```
<?
$mensaje = “Hola”;
echo $mensaje;
?>
```

También puede usarse:

```
<?php
$mensaje = “Hola”;
echo $mensaje;
?>
```

Ventajas:

- Muy fácil de aprender.
- Se caracteriza por ser un lenguaje muy rápido.
- Soporta en cierta medida la orientación a objeto. Clases y herencia.
- Es un lenguaje multiplataforma: Linux, Windows, entre otros.
- Capacidad de conexión con la mayoría de los manejadores de base de datos: MysSQL, PostgreSQL, Oracle, MS SQL Server, entre otras.
- Capacidad de expandir su potencial utilizando módulos.
- Posee documentación en su página oficial la cual incluye descripción y ejemplos de cada una de sus funciones.
- Es libre, por lo que se presenta como una alternativa de fácil acceso para todos.
- Incluye gran cantidad de funciones.
- No requiere definición de tipos de variables ni manejo detallado del bajo nivel.

Desventajas:

- Se necesita instalar un servidor web.
- Todo el trabajo lo realiza el servidor y no delega al cliente. Por tanto puede ser más ineficiente a medida que las solicitudes aumenten de número.
- La legibilidad del código puede verse afectada al mezclar sentencias HTML y PHP.
- La programación orientada a objetos es aún muy deficiente para aplicaciones grandes.
- Dificulta la modularización.
- Dificulta la organización por capas de la aplicación.

Seguridad:

PHP es un poderoso lenguaje e intérprete, ya sea incluido como parte de un servidor web en forma de módulo o ejecutado como un binario CGI separado, es capaz de acceder a archivos, ejecutar comandos y abrir conexiones de red en el servidor. Estas propiedades hacen que cualquier cosa que sea ejecutada en un servidor web sea insegura por naturaleza. PHP está diseñado específicamente para ser un lenguaje más seguro para escribir programas CGI que Perl o C, y con la selección correcta de opciones de configuración en tiempos de compilación y ejecución, y siguiendo algunas prácticas correctas de programación.

Lenguaje ASP

Es una tecnología del lado de servidor desarrollada por Microsoft para el desarrollo de sitio web dinámicos. ASP significa en inglés (Active Server Pages), fue liberado por Microsoft en 1996. Las páginas web desarrolladas bajo este lenguaje es necesario tener instalado Internet Information Server (IIS). ASP no necesita ser compilado para ejecutarse. Existen varios lenguajes que se pueden utilizar para crear páginas ASP. El más utilizado es VBScript, nativo de Microsoft. ASP se puede hacer también en Perl and Jscript (no JavaScript). El código ASP puede ser insertado junto con el código HTML. Los archivos cuentan con la extensión (asp).

Sintaxis:

<% %>

Ventajas:

- Usa Visual Basic Script, siendo fácil para los usuarios.
- Comunicación óptima con SQL Server.
- Soporta el lenguaje JScript (Javascript de Microsoft).

Desventajas:

- Código desorganizado.
- Se necesita escribir mucho código para realizar funciones sencillas.
- Tecnología propietaria.
- Hospedaje de sitios web costosos.

Lenguaje ASP.NET

Este es un lenguaje comercializado por Microsoft, y usado por programadores para desarrollar entre otras funciones, sitios web. ASP.NET es el sucesor de la tecnología ASP, fue lanzada al mercado mediante una estrategia de mercado denominada .NET.

El ASP.NET fue desarrollado para resolver las limitantes que brindaba tu antecesor ASP. Creado para desarrollar web sencillas o grandes aplicaciones. Para el desarrollo de ASP.NET se puede utilizar C#, VB.NET o J#. Los archivos cuentan con la extensión (aspx). Para su funcionamiento de las páginas se necesita tener instalado IIS con el Framework .Net. Microsoft Windows 2003 incluye este framework, solo se necesitará instalarlo en versiones anteriores.

Ventajas:

- Completamente orientado a objetos.
- Controles de usuario y personalizados.
- División entre la capa de aplicación o diseño y el código.
- Facilita el mantenimiento de grandes aplicaciones.
- Incremento de velocidad de respuesta del servidor.
- Mayor velocidad.
- Mayor seguridad.

Desventajas:

- Mayor consumo de recursos.

Lenguaje JSP

Es un lenguaje para la creación de sitios web dinámicos, acrónimo de Java Server Pages. Está orientado a desarrollar páginas web en Java. JSP es un lenguaje multiplataforma. Creado para ejecutarse del lado del servidor. JSP fue desarrollado por Sun Microsystems. Comparte ventajas similares a las de ASP.NET, desarrollado para la creación de aplicaciones web potentes. Posee un motor de páginas basado en los servlets de Java. Para su funcionamiento se necesita tener instalado un servidor Tomcat.

Características:

- Código separado de la lógica del programa.
- Las páginas son compiladas en la primera petición.
- Permite separar la parte dinámica de la estática en las páginas web.
- Los archivos se encuentran con la extensión (jsp).
- El código JSP puede ser incrustado en código HTML.

Elementos de JSP

Los elementos que pueden ser insertados en las páginas JSP son los siguientes:

- Código: se puede incrustar código “Java”.

- Directivas: permite controlar parámetros del servlet
- Acciones: permite alterar el flujo normal de ejecución de una página.

Ventajas:

- Ejecución rápida del servlets.
- Crear páginas del lado del servidor.
- Multiplataforma.
- Código bien estructurado.
- Integridad con los módulos de Java.
- La parte dinámica está escrita en Java.
- Permite la utilización se servlets.

Desventajas:

- Complejidad de aprendizaje.

Lenguaje Python

Es un lenguaje de programación creado en el año 1990 por Guido van Rossum, es el sucesor del lenguaje de programación ABC. Python es comparado habitualmente con Perl. Los usuarios lo consideran como un lenguaje más limpio para programar. Permite la creación de todo tipo de programas incluyendo los sitios web. Su código no necesita ser compilado, por lo que se llama que el código es interpretado. Es un lenguaje de programación multiparadigma, lo cual fuerza a que los programadores adopten por un estilo de programación particular:

- Programación orientada a objetos.
- Programación estructurada.
- Programación funcional.
- Programación orientada a aspectos.

Ejemplo de una clase en Phyton:

```
def dibujar_muneco(opcion):
   if opcion == 1:
      C.create_line(580, 150, 580, 320, width=4, fill="blue")
      C.create_oval(510, 150, 560, 200, width=2, fill='PeachPuff')
```

Ventajas:

- Libre y fuente abierta.
- Lenguaje de propósito general.

- Gran cantidad de funciones y librerías.
- Sencillo y rápido de programar.
- Multiplataforma.
- Licencia de código abierto (Opensource).
- Orientado a Objetos.
- Portable.

Desventajas:

- Lentitud por ser un lenguaje interpretado.

Lenguaje Ruby

Es un lenguaje interpretado de muy alto nivel y orientado a objetos. Desarrollado en el 1993 por el programador japonés Yukihiro "Matz" Matsumoto. Su sintaxis está inspirada en Phyton, Perl. Es distribuido bajo licencia de software libre (Opensource).

Ruby es un lenguaje dinámico para una programación orientada a objetos rápida y sencilla. Para los que deseen iniciarse en este lenguaje pueden encontrar un tutorial interactivo de ruby. Se encuentra también a disposición de estos usuarios un sitio con informaciones y cursos en español.

Sintaxis:

puts "hola"

Características:

- Existe diferencia entre mayúsculas y minúsculas.
- Múltiples expresiones por líneas, separadas por punto y coma ";".
- Dispone de manejo de excepciones.
- Ruby puede cargar librerías de extensiones dinámicamente si el (Sistema Operativo) lo permite.
- Portátil.

Ventajas:

- Permite desarrollar soluciones a bajo Costo.
- Software libre.
- Multiplataforma.

Visual Studio

Visual Studio es un conjunto completo de herramientas de desarrollo para la generación de:

- Aplicaciones web ASP.NET.
- Aplicaciones de escritorio.
- Aplicaciones móviles.

Visual Basic, Visual C# utilizan todos el mismo entorno de desarrollo integrado (IDE), que habilita el uso compartido de herramientas y hace más sencilla la creación de soluciones en varios lenguajes, se pueden incorporar otros lenguajes al entorno de desarrollo. Asimismo, dichos lenguajes utilizan las funciones de .NET Framework, las cuales ofrecen acceso a tecnologías clave para simplificar el desarrollo de aplicaciones web ASP.

La gama de productos de Visual Studio comparte un único entorno de desarrollo integrado (IDE) que se compone de varios elementos: el Menú, barra de herramientas Estándar, varias ventanas de herramientas que se acoplan u ocultan automáticamente a la izquierda, parte inferior y a la derecha, así como en el espacio del editor. Las ventanas de herramientas, menús y barras de herramientas disponibles dependen del tipo de proyecto o archivo en el que esté trabajando.

Selección del lenguaje de programación más apropiado al proyecto a desarrollar.

- Según las especificaciones del sistema se puede optar por lenguajes para el desarrollo de:
  - Sistemas de escritorio (control de stock),
  - Entorno web (sistema comercial, sitio académico, etc.).
  - Existen dos ramas de los lenguajes, que pueden ser:
    - Software libre y gratuito (JAVA, PHP).
    - Con licencia.

Por ejemplo:Si nuestro sistema requiere en su mayor parte de la aplicación de fórmulas matemáticas, los lenguajes aconsejables podrían ser: JAVA, Matlab o el Mathemática.

TEMA II: Especificación de Requerimientos

Especificación de requerimientos. Estudio y análisis de una aplicación informática. Comprensión del dominio del problema. Técnicas de especificación de requerimientos. Casos de usos. Diagramas UML. Validación de requisitos. Aplicaciones prácticas

- Introducción:

Los **requerimientos para un sistema de software** determinan lo que hará el sistema y definen las restricciones de su operación e implementación.

El **análisis de requisitos** es una de las tareas más importantes en el ciclo de vida del desarrollo de software, puesto que en ella se determinan los "planos" de la nueva aplicación.

En cualquier proyecto software los **requisitos** son **las necesidades del producto** que se debe desarrollar. Por ello, en la fase de análisis de requisitos se deben identificar claramente estas necesidades y documentarlas.

Como resultado de esta fase se debe producir un documento de especificación de requisitos en el que se describa lo que el futuro sistema debe hacer. Por tanto, no se trata simplemente de una actividad de análisis, sino también de síntesis.

El análisis de requisitos se puede definir como el proceso del estudio de las necesidades de los usuarios para llegar a una definición de los requisitos del sistema, hardware o software, así como el proceso de estudio y refinamiento de dichos requisitos, definición proporcionada por el IEEE [Piattini, 1996]. Asimismo, se define requisito como una condición o capacidad que necesita el usuario para resolver un problema o conseguir un objetivo determinado [Piattini, 1996]. Esta definición se extiende y se aplica a las condiciones que debe cumplir o poseer un sistema o uno de sus componentes para satisfacer un contrato, una norma o una especificación.

En la determinación de los requisitos no sólo deben actuar los analistas, es muy importante la participación de los propios usuarios, porque son éstos los que mejor conocen el sistema que se va a automatizar. Analista y cliente se deben poner de acuerdo en las necesidades del nuevo sistema, ya que el cliente no suele entender el proceso de diseño y desarrollo del software como para redactar una **especificación de requisitos software** (ERS) y los analistas no suelen entender completamente el problema del cliente, debido a que no dominan su área de trabajo.

Así pues, el documento de **especificación de requisitos** debe ser legible por el cliente, con lo que se evita el malentendido de determinadas situaciones, ya que el cliente participa activamente en la extracción de dichos requisitos.

Basándose en estos requisitos, el ingeniero de software procederá al modelado de la futura aplicación. Para ello, se pueden utilizar diferentes tipos de metodologías entre las que se destacan la *metodología estructurada* y la *metodología orientada a objetos* (por ejemplo DFDs y UML respectivamente).

La *metodología estructurada* está basada en la representación de las funciones que debe realizar el sistema y los datos que fluyen entre ellas.

En la *metodología orientada a objetos* se utiliza el UML [Pierre-Alain, 1997], mediante el cual podemos representar diagramas (casos de uso) que permiten definir el sistema desde el punto de vista del usuario estableciendo las relaciones entre el futuro sistema y su entorno. Estas relaciones se establecen en forma de acciones del usuario y reacciones del sistema.

La **ingeniería de requisitos** facilita el mecanismo apropiado para comprender lo que quiere el cliente, analizando necesidades, confirmando su viabilidad, negociando una solución razonable, especificando la solución sin ambigüedad, validando la especificación y gestionando los requisitos para que se transformen en un sistema operacional.

El **proceso** de ingeniería de requisitos puede ser descrito en 5 pasos distintos:

1. **Identificación de requisitos,**
2. **Análisis de requisitos y negociación,**
3. **Especificación de requerimientos,**
4. **Modelado del sistema,**
5. **Validación y gestión de requisitos**.

1.- IDENTIFICACION DE REQUISITOS.

La ERS (especificación de requisitos de software), es una descripción que debe decir ciertas cosas y al mismo tiempo debe decirlas de una determinada manera. Una de las formas viene especificada por el estándar IEEE 830.

Una ERS forma parte de la documentación asociada al software que se está desarrollando, por tanto debe definir correctamente todos los requerimientos, pero no más de los necesarios. Esta documentación no debería describir ningún detalle de diseño, modo de implementación o gestión del proyecto, ya que los requisitos se deben describir de forma que el usuario pueda entenderlos. Al mismo tiempo, se da una mayor flexibilidad a los desarrolladores para la implementación.

Así pues, el grado y el lenguaje utilizado para la documentación de los requisitos estarán en función del nivel que el usuario tenga para entender dichas especificaciones.

Christel y Kang identifican una serie de problemas que nos ayudan a comprender por qué la obtención de requisitos es costosa.

• *Problemas de alcance.* El límite del sistema está mal definido o los detalles técnicos innecesarios, que han sido aportados por los clientes/usuarios, pueden confundir más que clarificar los objetivos del sistema.

• *Problemas de comprensión*. Los clientes no están completamente seguros de lo que necesitan, tienen una pobre comprensión de las capacidades y limitaciones de su entorno de computación, no existe un total entendimiento del problema, etc.

• *Problemas de volatilidad.* Los requisitos cambian con el tiempo.

Para ayudar a solucionar estos problemas, los ingenieros de sistemas deben aproximarse de una manera organizada a través de reuniones para definir requisitos. El resultado

alcanzado como consecuencia de la identificación de requisitos variará dependiendo del tamaño del sistema o producto a construir.

2.- ANÁLISIS Y NEGOCIACIÓN DE REQUISITOS.

Una vez recopilados los requisitos, el producto obtenido configura la base del análisis de requisitos. Los requisitos se agrupan por categorías y se organizan en subconjuntos, se estudia cada requisito en relación con el resto, se examinan los requisitos en su consistencia, completitud y ambigüedad, y se clasifican en base a las necesidades de los clientes/usuarios.

Es corriente en clientes y usuarios solicitar más de lo que puede realizarse, consumiendo recursos de negocios limitados. También es relativamente común en clientes y usuarios el proponer requisitos contradictorios, argumentando que su versión es “esencial por necesidades especiales”.

El ingeniero del sistema debe resolver estos conflictos a través de un proceso de negociación. Los clientes, usuarios y el resto de intervinientes deberán clasificar sus requisitos y discutir los posibles conflictos según su prioridad. Los riesgos asociados con cada requisito serán identificados y analizados. Se efectúan estimaciones del esfuerzo de desarrollo que se utilizan para valorar el impacto de cada requisito en el costo del proyecto y en el plazo de entrega. Utilizando un procedimiento iterativo, se irán eliminando requisitos, se irán combinando y/o modificando para conseguir satisfacer los objetivos planteados.

3.- ESPECIFICACION DE REQUERIMIENTOS.

El termino **requerimiento** no se utiliza de forma consistente en la industria del software. En algunos casos, un requerimiento se visualiza como una declaración abstracta de alto nivel de un **servicio** que debe proveer el sistema o como una **restricción** de éste. Por otro lado, es una definición matemática detallada y formal de una **función del sistema.**

❖ REQUERIMIENTOS FUNCIONALES Y NO FUNCIONALES

A menudo los requerimientos de sistemas de software se clasifican en funcionales y no funcionales, o como requerimientos del dominio.

Requerimientos funcionales

Son declaraciones de los servicios que proveerá el sistema, de la manera en que éste reaccionará a entradas particulares. En algunos casos, los requerimientos funcionales de los sistemas también declaran explícitamente lo que el sistema no debe hacer.

Los **requerimientos funcionales** de un sistema describen la funcionalidad o los servicios que se espera que éste provea. Estos dependen del tipo de software y del sistema que se

desarrolle y de los posibles usuarios del software. Cuando se expresan como requerimientos del usuario, habitualmente se describen de forma general mientras que los requerimientos funcionales del sistema describen con detalle la función de éste, sus entradas y salidas, excepciones, etc.

Muchos de los problemas de la ingeniería de software provienen de la imprecisión en la especificación de requerimientos. Para un desarrollador de sistemas es natural dar interpretaciones de un requerimiento ambiguo con el fin de simplificar su implementación. Sin embargo, a menudo no es lo que el cliente desea. Se tienen que estipular nuevos requerimientos y se deben hacer cambios al sistema, retrasando la entrega de éste e incrementando el costo.

En principio, la especificación de requerimientos funcionales de un sistema debe estar completa y ser consistente. La *compleción* significa que todos los servicios solicitados por el usuario están definidos. La *consistencia* significa que los requerimientos no tienen definiciones contradictorias. En la práctica, para sistemas grandes y complejos, es imposible cumplir los requerimientos de consistencia y compleción. La razón de esto se debe parcialmente a la complejidad inherente del sistema y parcialmente a que los diferentes puntos de vista tienen necesidades inconsistentes. Estas inconsistencias son obvias cuando los requerimientos se especifican por primera vez. Los problemas emergen después de un análisis profundo. Una vez que éstos se hayan descubierto en las diferentes revisiones o en las fases posteriores del ciclo de vida, se deben corregir en el documento de requerimientos.

Requerimientos no funcionales

Son restricciones de los servicios o funciones ofrecidos por el sistema. Incluyen restricciones de tiempo, sobre el proceso de desarrollo, estándares, etc.

Son aquellos requerimientos que no se refieren directamente a las funciones específicas que entrega el sistema, sino a las *propiedades emergentes de éste como la fiabilidad,* la respuesta en el tiempo y la capacidad de almacenamiento. De forma alternativa, definen las restricciones del sistema como la capacidad de los dispositivos de entrada/salida y la representación de datos que se utiliza en la interface del sistema.

Muchos requerimientos no funcionales se refieren al sistema como un todo más que a rasgos particulares del mismo. Esto significa que a menudo con más críticos que los requerimientos funcionales particulares. Mientras que el incumplimiento de este último degradará el sistema, una falla en un requerimiento no funcional del sistema lo inutiliza.

Los requerimientos no funcionales surgen de la necesidad del usuario, debido a las restricciones en el presupuesto, a las políticas de la organización, a la necesidad de interoperabilidad con otros sistemas de software o hardware o a factores externos como los reglamentos de seguridad, las políticas de privacidad, etcétera.

Estos diferentes tipos de requerimientos se clasifican de acuerdo con sus implicaciones.

• *Requerimientos del producto*. Especifican el comportamiento del producto; como los requerimientos de desempeño en la rapidez de ejecución del sistema y cuánta memoria se requiere; los de fiabilidad que fijan la tasa de fallas para que el sistema sea aceptable; los de portabilidad y los de usabilidad.

• *Requerimientos organizacionales*. Se derivan de las políticas y procedimientos existentes en la organización del cliente y en la del desarrollador: estándares en los procesos que deben utilizarse; requerimientos de implementación como los lenguajes de programación o el método de diseño a utilizar, y los requerimientos de entrega que especifican cuándo se entregará el producto y su documentación.

• *Requerimientos externos*. Se derivan de los factores externos al sistema y de su proceso de desarrollo. Incluyen los requerimientos de interoperabilidad que definen la manera en que el sistema interactúa con los otros sistemas de la organización; los requerimientos legales que deben seguirse para asegurar que el sistema opere dentro de la ley, y los requerimientos éticos. Estos últimos son impuestos al sistema para asegurar que será aceptado por el usuario y por el público en general.

Un problema común con los **requerimientos no funcionales** es que algunas veces son difíciles de verificar. Se redactan para reflejar las metas generales del usuario, como la facilidad de uso, la capacidad del sistema para recuperarse de las fallas o la respuesta rápida al usuario. Estos requerimientos causan problemas a los desarrolladores del sistema puesto que dejan abierta la posibilidad a la interpretación, lo que provoca discusiones subsecuentes una vez que el sistema se entregue.

De forma ideal, los requerimientos no funcionales no se deben expresar de manera cuantitativa utilizando métricas que se puedan probar de forma objetiva.

En la práctica, la especificación cuantitativa de requerimientos es difícil. A los clientes no les es posible traducir sus metas en requerimientos cuantitativos; para algunas de éstas, como las de mantenimiento, no existen métricas que se puedan utilizar; el costo de verificar de forma objetiva los requerimientos no funcionales cuantitativos es muy alto.

En principio, ***los requerimientos funcionales y no funcionales*** se diferencian en el documento de requerimientos. En la práctica, esto es difícil. Si un requerimiento no funcional se declara de forma separada a los funcionales, algunas veces es difícil ver la relación entre ellos. Si se declaran con los requerimientos funcionales, es difícil separar las condiciones funcionales y no funcionales e identificar los requerimientos que se refieren al sistema como un todo. Se debe hallar un balance apropiado que dependa del tipo de sistema a especificar. Sin embargo, los requerimientos que claramente se refieren a las propiedades emergentes el sistema se deben resaltar. Esto se hace colocándolos en una sección aparte o diferenciándolos, de alguna forma, de los otros requerimientos del sistema.

Requerimientos del dominio

Son requerimientos que provienen del dominio de aplicación del sistema y que reflejan las características de ese dominio. Éstos pueden ser funcionales o no funcionales.

Se derivan del dominio del sistema más que de las necesidades especificas de los usuarios. Pueden ser requerimientos funcionales nuevos, restringir los existentes o establecer cómo se deben ejecutar cálculos particulares. Los requerimientos del dominio son importantes debido a que a menudo reflejan los fundamentos del dominio de aplicación. Si estos requerimientos no se satisfacen, es imposible hacer que el sistema trabaje de forma satisfactoria.

❖ REQUERIMIENTOS DEL USUARIO Y DEL SISTEMA

Algunos de los problemas que surgen durante el proceso de ingeniería de requerimientos son resultado de no hacer una clara separación entre los diferentes niveles de descripción. Esto se hace utilizando requerimientos del usuario para determinar los requisitos abstractos de alto nivel, y requisitos del sistema, para designar la descripción detallada de lo que el sistema debe hacer. De igual forma que en estos dos niveles de detalle, se puede producir una descripción más detallada para establecer un puente entre la ingeniería de requerimientos y las actividades de diseño. Los requerimientos del usuario, los del sistema y la especificación del diseño de software se definen de la siguiente manera:

Requerimientos del usuario

Son **declaraciones** en lenguaje natural y en diagramas de los servicios que se espera que el sistema provea y de las restricciones bajo las cuales debe operar.

Describen los requerimientos funcionales y no funcionales de tal forma que sean comprensibles por los usuarios del sistema que no posean un conocimiento técnico detallado. Únicamente especifican el comportamiento externo del sistema y evitan, tanto como sea posible, las características de diseño del sistema. Por consiguiente, los requerimientos del usuario no se deben definir utilizando un modelo de implementación.

Deben redactarse utilizando el lenguaje natural, representaciones y diagramas intuitivos sencillos.

Sin embargo, pueden surgir diversos problemas cuando se redactan en lenguaje natural: falta de claridad, confusión de requerimientos y conjunción de requerimientos.

Los requerimientos del sistema

Establecen con detalle los servicios y restricciones del sistema. El documento de requerimientos del sistema, algunas veces denominado especificación funcional, debe ser preciso. Éste sirve como un contrato entre el comprador del sistema y el desarrollador del software.

Son descripciones más detalladas de los requerimientos del usuario. Sirven como base para definir el contrato de la especificación del sistema y, por lo tanto, debe ser una especificación completa y consistente del sistema. Son utilizados por los ingenieros de software como el punto de partida para el diseño del sistema.

La especificación de requerimientos del sistema incluye diferentes modelos del sistema como el de objetos o el de flujo de datos.

En principio, los requerimientos del sistema deberán establecer lo que éste hará y no la manera en que se implementará. Sin embargo, en el nivel de detalle requerido para especificar el sistema completamente, es casi imposible excluir toda la información de diseño.

Una especificación del diseño del software

Es una descripción abstracta del diseño del software, que es una base para un diseño e implementación detallados; agrega detalle a la especificación de requerimientos del sistema.

❖ **EL DOCUMENTO DE REQUERIMIENTOS DEL SOFTWARE**

Este es la declaración oficial de qué es lo que requieren los desarrolladores del sistema. Incluye tanto los requerimientos del usuario para el sistema como una especificación detallada de los requerimientos del sistema. En algunos casos, los dos tipos de requerimientos se integran en una única descripción. En otros, los del usuario se definen en una introducción de la especificación de los del sistema. Si existe un gran número de requerimientos, los detalles de los requerimientos del sistema se pueden presentar como documentos separados.

El documento de requerimientos tiene un conjunto diverso de usuarios que va desde los administradores principales de la organización, quienes pagan por el sistema, hasta los ingenieros responsables del software.

Una gran variedad de organizaciones han definido estándares para los documentos de requerimientos.

El IEEE sugiere la siguiente estructura para los documentos de requerimientos.

1. Introducción

• Propósito del documento de requerimientos

• Alcance del producto

• Definiciones, acrónimos y abreviaturas

• Referencias

• Resumen del resto del documento

2. Descripción general

• Perspectiva del producto

• Funciones del producto

• Características del usuario

• Restricciones generales

• Suposiciones y dependencias

3. Requerimientos Específicos. Cubren los requerimientos funcionales, no funcionales y de interfaz. Obviamente, ésta es la parte más sustancial del documento, pero debido a la amplia variabilidad en la práctica organizacional, no es apropiado definir una estructura estándar para esta sección. Los requerimientos pueden documentar las interfaces externas, describir la funcionalidad y el desempeño del sistema, especificar los requerimientos lógicos de la base de datos, las restricciones de diseño, las propiedades emergentes del sistema y las características de calidad.

4**. Apéndices.**

5**. Índice.**

Aunque el estándar IEEE no es ideal, contiene una buena ayuda de cómo tratar los requerimientos y evitar problemas. Es muy general para que pueda ser estándar de una organización. Sin embargo, se puede transformar y adaptar para definir un estándar que se ajuste a las necesidades de una organización en particular.

Por supuesto, la información que se incluya en un documento de requerimientos debe depender del tipo de software a desarrollar y del enfoque de desarrollo que se utilice.

4.- MODELADO DEL SISTEMA

Es importante evaluar los componentes del sistema y sus relaciones entre sí, determinar cómo están reflejados los requisitos, y valorar como se ha concebido la “estética” en el **sistema.**

5.- VALIDACIÓN DE REQUISITOS

El resultado del trabajo realizado es una consecuencia de la ingeniería de requisitos (especificación del sistema e información relacionada) y es evaluada su calidad en la fase de validación. La **validación de requisitos** examina las especificaciones para asegurar que

todos los requisitos del sistema han sido establecidos sin ambigüedad, sin inconsistencias, sin omisiones, que los errores detectados hayan sido corregidos, y que el resultado del trabajo se ajusta a los estándares establecidos para el proceso, el proyecto y el producto.

GESTIÓN DE REQUISITOS

Es un conjunto de actividades que ayudan al equipo de trabajo a identificar, controlar y seguir los requisitos y los cambios en cualquier momento.

FACTORES EN LA CALIDAD DEL SOFTWARE.

La construcción de software de calidad requiere el cumplimiento de numerosas características:

- ✓ **Eficiencia**: La eficiencia del software es su capacidad para hacer un buen uso de los recursos que manipula.
- ✓ **Portabilidad**: La portabilidad es la facilidad con la que un software puede ser transportado sobre diferentes sistemas físicos o lógicos.
- ✓ **Verificabilidad**: La verificabilidad es facilidad de verificación de un software; es su capacidad para soportar los procedimientos de validación y de aceptar juegos de test o ensayo de programas.
- ✓ **Integridad:** La integridad es la capacidad de un software para proteger sus propios componentes contra los procesos que no tengan derecho de acceso.
- ✓ Fácil de utilizar: Un software es fácil de utilizar si se puede comunicar con él de manera cómoda.
- ✓ **Corrección:** Capacidad de los productos software de realizar exactamente las tareas definidas por su especificación.
- ✓ **Robustez:** Capacidad de los productos software de funcionar incluso en situaciones anormales.
- ✓ **Extensibilidad:** Facilidad que tienen los productos de adaptarse a cambios en su especificación. Existen dos principios fundamentales para conseguir esto: diseño simple y descentralización.
- ✓ **Reutilización**: Capacidad de los productos para ser reutilizados, en su totalidad o en parte, en nuevas aplicaciones.
- ✓ **Compatibilidad:** Facilidad de los productos para ser combinados con otros.

ESTUDIO Y ANALISIS DE UNA APLICACIÓN INFORMATICA

El **análisis y diseño de sistemas o aplicación informática** se refiere al proceso de examinar la situación de una empresa con el propósito de mejorar con métodos y procedimientos más adecuados.

El desarrollo de sistemas tiene dos componentes: Análisis y Diseño.

Análisis: Es el proceso de clasificación e interpretación de hechos, diagnostico de problemas y empleo de la información para recomendar mejoras al sistema. Especifica que es lo que el sistema debe hacer.

Diseño: Especifica las características del producto terminado. Establece como alcanzar el objetivo.

LO QUE NO ES EL ANÁLISIS DE SISTEMAS

- ✓ El estudio de una empresa para buscar procesos ya existentes con el propósito de determinar cuáles deberían, ser llevados a cabo por una computadora y cuáles por métodos manuales. La finalidad del análisis está en comprender los detalles de una situación y decir si es deseable o factible una mejora. La selección del método, ya sea utilizando o no una computadora, es un aspecto secundario.
- ✓ Determinar los cambios que deberían efectuarse.
- ✓ Determinar la mejor forma de resolver un `problema de sistemas de información. Sin importar cuál sea la organización, el analista trabaja en los problemas de ésta. Es un error hacer una distinción entre los problemas de la empresa y los de sistemas ya que estos últimos no existirían sin los primeros. Cualquier sugerencia debe primero considerarse a la luz de si beneficiará o perjudicará a la organización. No se debe ir tras ideas técnicamente atractivas a menos que estas mejoren el sistema de la organización.

ELEMENTOS DE UN SISTEMA DE INFORMACION

SOFWARE. Los programas de computadoras, las estructuras de datos y la documentación asociada, que sirve para realizar el método lógico.

HARWARE: Los dispositivos electrónicos que proporcionan la capacidad de computación y que proporcionan las funciones del mundo exterior.

USUARIOS: Los individuos que son usuarios y operadores del software y del hardware.

BASES DE DATOS: Una colección grande y organizada de información a la que se accede mediante el software y que es una parte integral del funcionamiento del sistema.

DOCUMENTACION: Los manuales, los impresos y otra información descriptiva que explica el uso y / o la operación.

PROCESAMIENTOS: Los pasos que definen el uso especifico de cada elemento del sistema o el contexto procedimental en que reside el sistema.

CONTROL: Los sistemas trabajan mejor cuando operan dentro de niveles de control tolerables de rendimiento por ejemplo: el sistema de control de un calentador de agua.

CLASIFICACION DE LOS ISTEMAS DE INFORMACION

- **ABIERTOS**. Son los que intercambian información, materiales y energía con su ambiente.
- **CERRADOS**. Son auto-contenidos, no interactúan con el medio ambiente.
- **PROBABILISTICO**. No se conoce con certeza su comportamiento.
- **DETERMINISTICO.** Cualquier estado futuro que adopten puede preciarse con antelación.

UML y la Especificación de Requisitos.

Para determinar la funcionalidad de un sistema a desarrollar, UML señala el uso de dos elementos: el **actor** y el **caso de uso**.

El **actor** representa una entidad externa que interactúa con el sistema. Las entidades externas podrían ser personas u otros sistemas. Es importante resaltar que los actores son abstracciones de papeles o roles y no necesariamente tienen una correspondencia directa con personas.

A diferencia del actor, el **caso de uso** hace referencia al sistema a construir, detallando su comportamiento, el cual se traduce en resultados que pueden ser observados por el actor. Los **casos de uso** describen las cosas que los actores quieren que el sistema haga, por lo que un caso de uso deberá ser una tarea completa desde la perspectiva del actor.

Los **actores y los casos de uso** forman un modelo al que se le denomina *"modelo de casos de uso"*. Dicho modelo muestra el comportamiento del sistema desde la perspectiva del usuario y servirá como producto de entrada para el análisis y diseño del sistema. La **figura 1** muestra la notación que se debe utilizar para representar un actor y un caso de uso.

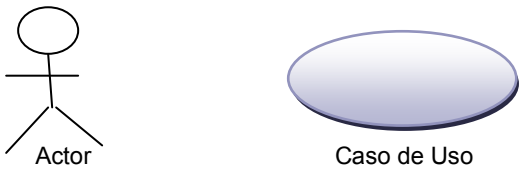
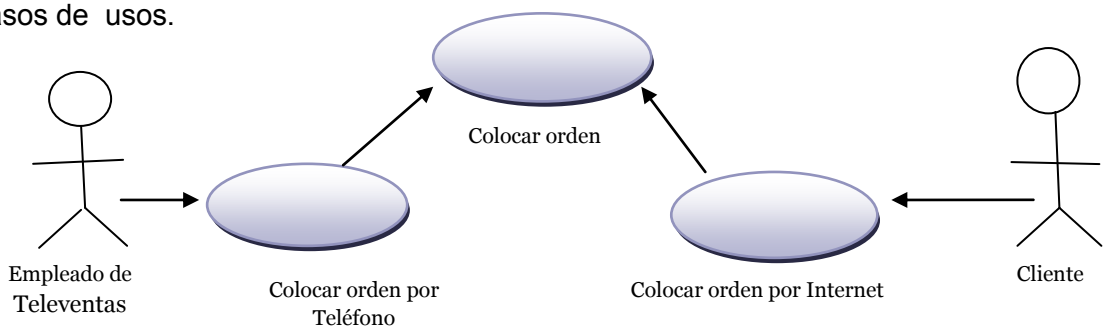


Figura 1. Representación grafica de actor y caso de uso

UML especifica que para representar gráficamente la relación entre un actor y caso de uso se debe trazar una línea que los una a la que se le denomina "*relación de comunicación*". Además, **UML** señala que los casos de uso pueden tener relaciones entre sí. Los tipos de relaciones que pueden existir son: **"include", "extends" y "generalization".**

La figura 2 muestra un ejemplo de casos de uso con relaciones de tipo **"generalization"** entre casos de usos.



Actividades para la Especificación de Requisitos con Casos de Uso

Los resultados de la especificación de requisitos son dos productos: el **catalogo de requisitos** y el **documento de especificación de requisitos de software**. El primero de ellos contiene la lista de requisitos de software clasificada por tipo y prioridad; y el segundo de ellos, especifica el comportamiento del sistema a un grado de detalle mayor al del catalogo de requisitos. La **figura 3**, indica el contenido de los productos de la especificación de requisitos de software mediante un diagrama de clases.

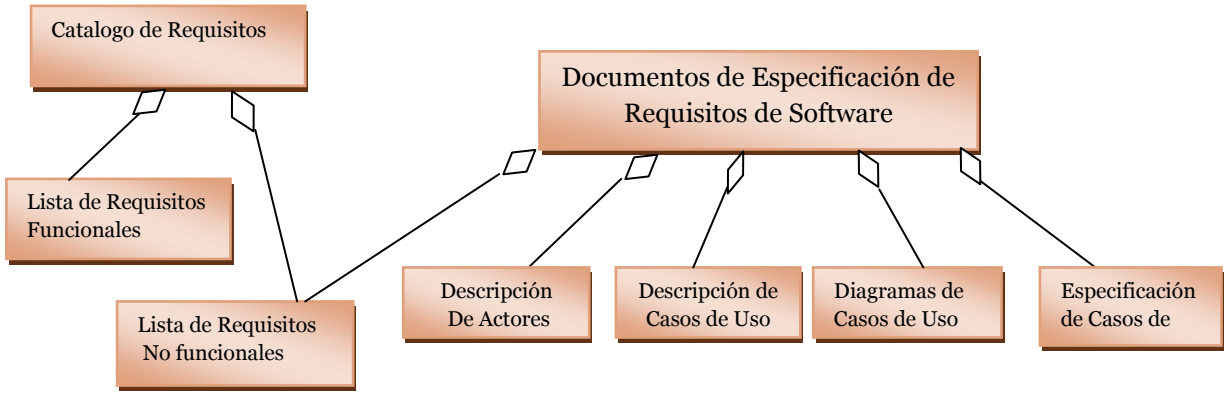


Figura 3. Diagrama de clases que representan los productos de la especificación de requisitos

Las actividades para la especificación de requisitos de software usando casos de uso son las siguientes:

- identificar y clasificar requisitos, identificar actores, identificar escenarios, identificar casos de uso, especificar casos de uso e identificar relaciones entre casos de uso. La secuencia de actividades, que se detallan en este parte, es el resultado de la adaptación de las propuestas metodológicas de Bernd Bruegge, Rational Unified Process y Métrica Versión 3 para la especificación de requisitos.
  La **figura 4** muestra la secuencia en que se deben realizar las actividades y sus resultados.

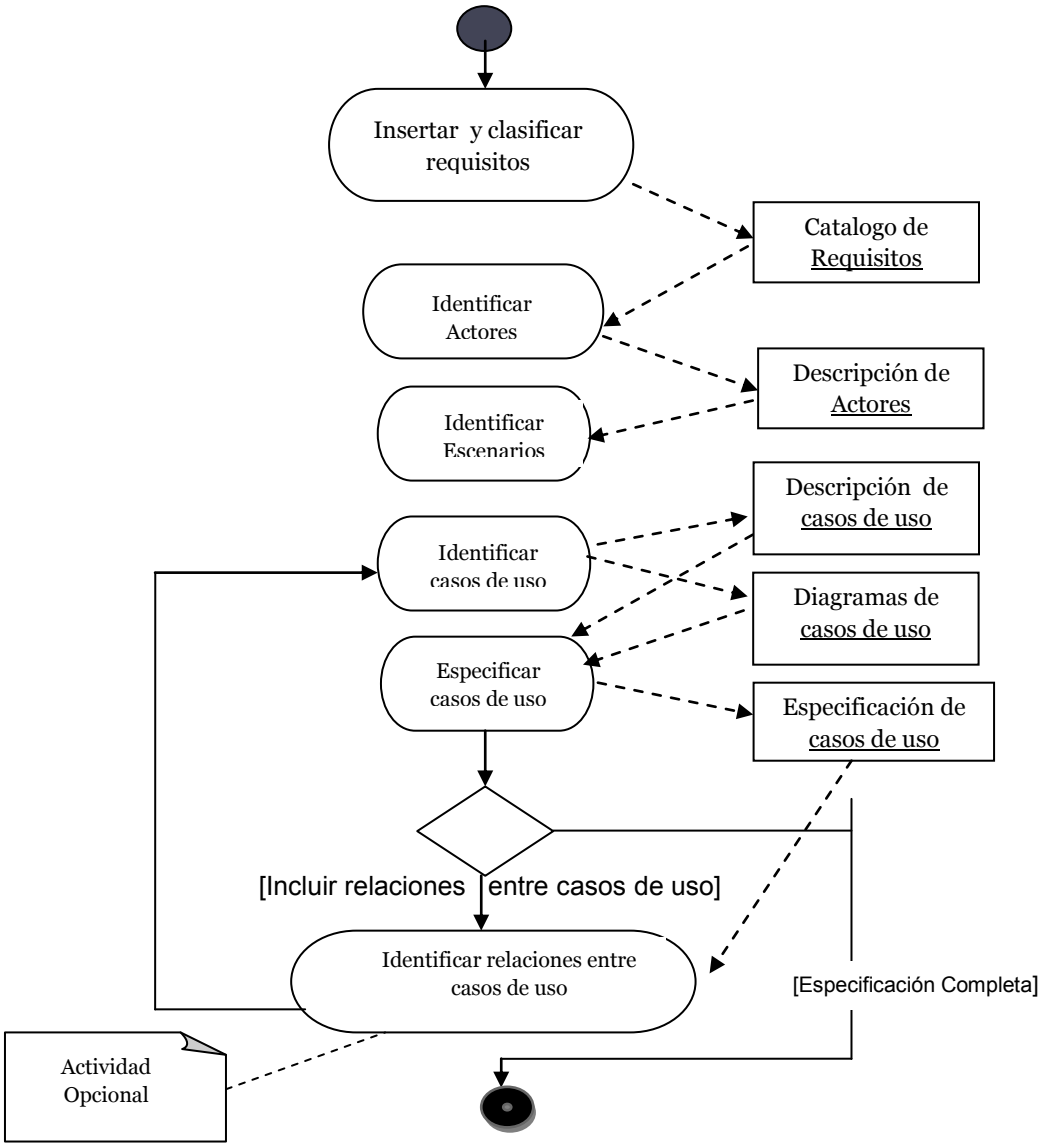


Figura 4: Actividades para la especificación de requisitos usando casos de uso.

Actividad 1: Identificar y clasificar requisitos.
Esta actividad es el punto de partida para las siguientes actividades del proceso de obtención de requisitos y se refiere a la **identificación de los requisitos** del sistema de software a desarrollar.
En esta actividad, deberemos responder a los siguientes cuestionamientos: 'que le permitirá hacer, el sistema de software, al usuario? y 'el cliente o usuario me solicita alguna restricción para construir el sistema de software?
Contestando a esas preguntas se deberá realizar una lista que contendrá los requisitos del sistema.
Luego de haber obtenido la lista de requisitos, estos deberán ser clasificados en dos grupos: requisitos

Funcionales y requisitos no funcionales. Los requisitos funcionales son declaraciones de los servicios que proveerá el sistema, de la manera en que este reaccionará entradas particulares y de cómo se comportara en situaciones particulares. En algunos casos, los requisitos funcionales declaran explícitamente lo que el sistema no debe hacer.
A diferencia de los requisitos funcionales, los no funcionales no se refieren directamente a las funciones específicas que entrega el sistema, sino a sus propiedades como fiabilidad, respuesta en el tiempo y capacidad de almacenamiento. Los requisitos funcionales son restricciones de los servicios o funciones ofrecidos por el sistema.

La tabla 1 muestra una relación de requisitos funcionales y no funcionales

Tabla 1. Ejemplo de Requisitos funcionales y no funcionales

Requisitos Funcionales	Requisitos No Funcionales
1.El sistema permitirá registrar los clientes de la Empresa.	3. La interfaz de usuario del sistema se implementara sobre un navegador Web.
2. El sistema permitirá a los usuarios realizar una búsqueda de los clientes por DNI, nombre o apellido.	4. El sistema deberá soportar al menos 20 transacciones por segundo.
	5. El sistema permitirá que los nuevos usuarios se familiaricen con su uso en menos de 15 minutos.

Luego, estos requisitos se clasificaran según su importancia, obteniéndose, de esta manera, una lista que contendrá los requisitos clasificados por dos criterios: tipo de requisito (funcional y no funcional) e importancia. A esta lista se le conoce como catalogo de requisitos.

Actividad 2: Identificar actores.

Luego de haber identificado los requisitos funcionales y no funcionales se procederá a identificar los **actores** del sistema. Para encontrar actores del sistema se puede buscar en las categorías de personas, otro software, dispositivos de hardware o redes de computadoras.
Para un **sistema de biblioteca,** los actores podrían ser: bibliotecario y cliente (si es que hay módulos de consulta de libros). En el caso de un sistema de ventas, los actores podrían ser: el cliente (si se realiza ventas por Internet), el vendedor y el sistema de facturación.
En un sistema, un usuario del sistema puede actuar como muchos actores; por ejemplo, en un banco, Juan Pérez podría ser cliente y operador dependiendo el momento y el uso que haga del sistema.

Actividad 3: Identificar escenarios.

Un escenario, segun Bruegge ***"es una descripción concreta, enfocada e informal de una sola característica del sistema desde el punto de vista de un solo actor";*** es decir, un escenario muestra la secuencia de pasos que se produce cuando un actor interactúa con el sistema en una situación específica y un tiempo determinado.

Un ejemplo de escenario para un sistema de biblioteca es el siguiente: “Juan Perez se conecta al sistema de la Biblioteca Nacional a través de Internet. Juan Perez selecciona realizar búsqueda y cuando aparece el formulario ingresa en **título** de libros la frase especificación de requisitos. El sistema encuentra un único libro y lo muestra, el libro de la biblioteca es Especificación de Requisitos de Software de Alan Davis y código B 73-825”
Cabe resaltar que no es necesario documentar los escenarios de manera formal. Esto quiere decir que carece de importancia la creación de documentos que describan todos los escenarios posibles del sistema, ya que su propósito es servir en la identificación de los casos de uso del sistema.

Actividad 4: Identificar casos de uso.

La diferencia entre **escenarios** y **casos de uso** radica en que un escenario es una **instancia** de un caso de uso. El **caso de uso** es el que especifica todos los escenarios posibles para una parte de funcionalidad dada; es decir, todos los escenarios similares se agrupan en un solo caso de uso.
Por ejemplo, en el **sistema de biblioteca** las consultas de bibliografía por Internet por parte de Juan Perez y cualquier otro cliente se puede agrupar en un solo caso de uso, al que se le puede denominar “Consultar bibliografía” (Ver figura 5).

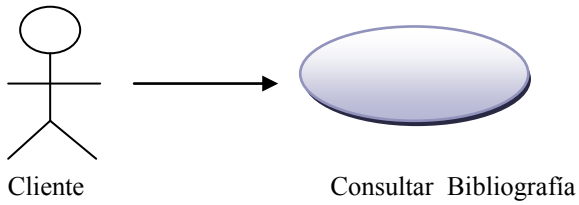


Figura 5. Diagrama de casos de uso para un sistema de biblioteca

Actividad 5: Especificar casos de uso

Luego de haber identificado los casos de uso, se tienen que indicar, detalladamente, la forma en la que el actor interactúa con el sistema. Esto se determina mediante la especificación y documentación de cada caso de uso.
RUP precisa que la especificación de cada caso de uso debe contener lo siguiente:

1. Precondiciones, que señalan los estados en que debe estar el sistema para que se pueda ejecutar el caso de uso.
2. Flujo básico, que señala la secuencia de pasos que se va a producir en la mayoría de las veces en que ese caso de uso se ejecute.
3. Flujos alternativos, que contienen las secuencias de pasos que se producirán como alternativas al flujo básico del caso de uso; es decir, especifican los pasos que se producirán en situaciones excepcionales.
4. Post condiciones, que señalan el estado en que el sistema quedara luego de haberse ejecutado el caso de uso.

Es importante resaltar que durante esta actividad se pueden producir las siguientes situaciones que deben tenerse en cuenta y que no deben ser motivo de preocupación:

1. Un caso de uso que debe partirse en dos casos de uso.
2. Un caso de uso que debe eliminarse, ya que debería formar parte de otro caso de uso.
3. Dos casos de uso que deben formar uno solo.

Actividad 6: Identificar relaciones entre casos de uso (opcional).

En esta actividad se identifican, en base a las especificaciones de casos de uso, las relaciones **"include", "extends" y "generalization"** entre casos de uso. Es importante resaltar que esta actividad es opcional.

La relación **"include"** se deberá determinar cuando la especificación de dos o más casos de uso contenga secuencias de acciones iguales. Para ello, la secuencia de pasos que se repite entre ellos, será extraída de esos **casos de uso** y se creara un nuevo caso de uso que los incluya.

La relación **"extend"** se produce cuando existe una secuencia de acciones que se producen en ocasiones excepcionales. Para ello, se creara un nuevo caso de uso que contenga dicha secuencia de pasos y dicho caso de uso extenderá la funcionalidad del caso uso original.

En cuanto a la relación **"generalization"** se identifica cuando existen casos de uso cuyo propósito es similar y contienen secuencias de acciones parecidas. En ese caso, se crea un caso de uso gene rico al cual se le denomina caso de uso padre, del cual heredan dos o más casos de uso. La relación **"generalization"** entre casos de uso es análoga a la relación de **"herencia"** entre clases.

4. Errores Comunes en la Especificación de Requisitos usando Casos de Uso

En cada una de las actividades especificadas anteriormente se producen y generan errores. A continuación se incluyen algunos de los más frecuentes.

4.1 Errores en la identificación de actores

Los errores introducidos en esta etapa se deben principalmente a no comprender quienes son los actores del sistema.

En algunos casos se incluyen actores que realmente no lo son; por ejemplo, en un sistema en el que se realizan pedidos de productos, se considera al cliente como un actor (Ver **figura 6**). Realmente quien ingresa los pedidos en el sistema es el vendedor y no el cliente, por lo tanto el vendedor sería el actor del sistema.

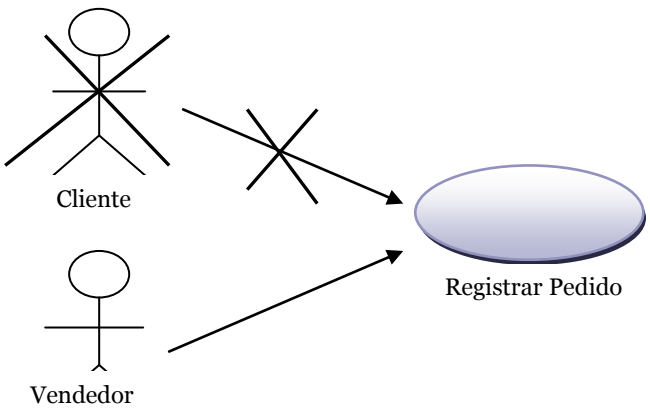


Figura 6. Diagrama de caso de uso para el registro de pedidos.

En el ejemplo de la **figura 6**, si el sistema permitiera registrar los pedidos por Internet, el cliente sí sería un actor del sistema.

4.2 Errores en la identificación de casos de uso.

Un error muy extendido, y que es cometido en la mayoría de la bibliografía sobre casos de uso, es considerar las opciones del menú o funciones del sistema como casos de uso (pueden revisar el libro de Larman y podrá encontrar este tipo de errores).
Kurt Bittner señala que los casos de uso deben mostrar lo que el usuario necesita del sistema y no mostrar las funciones u opciones del menú que permitirán realizar lo solicitado; por ejemplo, en un sistema donde se debe almacenar la información de los clientes, lo que al usuario le importa es actualizar la información de clientes. Esta actividad la podrá realizar accediendo a las opciones del menú agregar, modificar y eliminar clientes; por lo tanto la funcionalidad del sistema será representada con el caso de uso "Actualizar cliente" (ver **figura 7**).

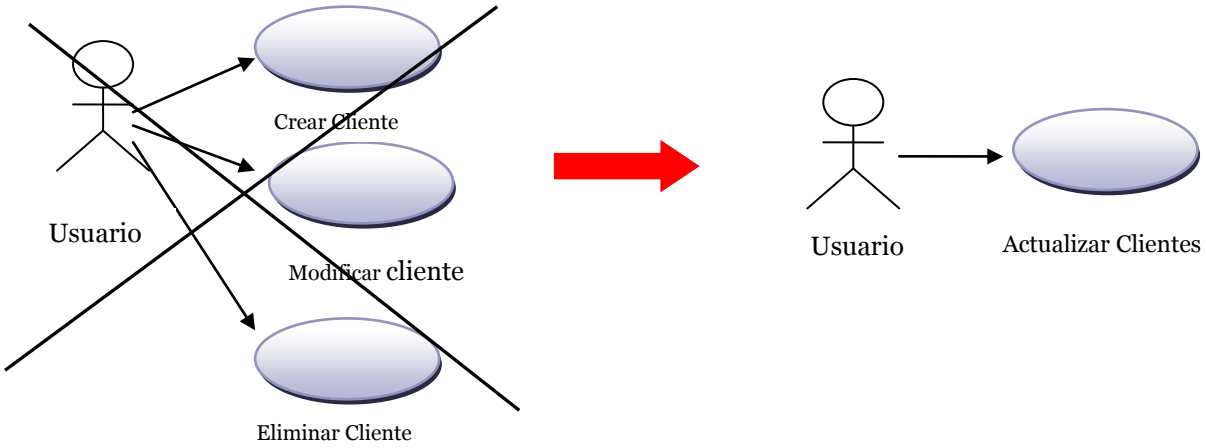


Figura 7. Considerar que los casos de uso son funciones del sistema.

4.3. Errores en la especificación de los casos de uso.

La técnica de casos de uso se debe utilizar para la especificación de requisitos del sistema, mas no para el diseño del sistema. Los errores que se producen en esta actividad se deben a la inclusión de cuestiones de diseño en la especificación de casos de uso. Algunos de los errores que se cometen son los siguientes:

- Introducir palabras que se refieran a componentes de ventanas como: botones, listas desplegables, opciones de menú, etc. En la especificación de casos de uso debe incluirse la información que será ingresada o será mostrada, pero no que componente de la ventana se va a utilizar para mostrar dicha información, sino se estaría realizando el diseño de pantallas en el proceso de especificación de requisitos, lo cual sería incorrecto.
- Mencionar elementos correspondientes al diseño de algoritmos o de base de datos en la especificación de casos de uso; por ejemplo, " grabar en la tabla clientes en la base de datos„ u " ordena los datos con el algoritmo de la burbuja„ son oraciones que no deben incluirse en una especificación; ya que son elementos que se determinan en la etapa de diseño.

Otro error es incluir “etc” o “así sucesivamente” cuando se indica la información que se debe ingresar o mostrar.
La especificación de **casos de uso** debe contener información exacta y precisa que permita realizar una buena estimación del esfuerzo requerido para realizar las etapas de análisis, diseño y codificación. Si la información no es exacta, se pueden producir retrasos debido a modificaciones de la base de datos, cambios en el diseño de las pantallas o en el código fuente producto de las especificaciones tardías de requisitos.

4.4 Errores en el uso de las relaciones entre casos de uso.

Los errores que se producen al incluir relaciones entre los casos de uso se deben principalmente a confundir los casos de uso con los procesos de los **diagramas de flujo de datos** (**DFD**) de Yourdon. Es por eso que se ven diagramas de casos de uso que parecen DFD, de manera similar al diagrama que se muestra en la **figura 8.**

Para evitar cometer este error, se aconseja que no haya más de dos niveles de relaciones de tipo **“include” o “extend”** en un diagrama de casos de uso.

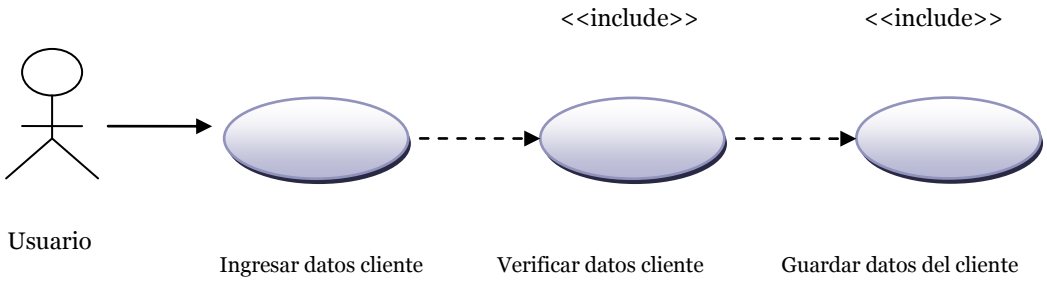


Figura 8. Error: casos de uso como DFD

Otro error frecuente es crear un caso de uso que es incluido por un solo caso de uso; por ejemplo, la **figura 9** muestra el caso de uso “Buscar Cliente”, el cual es incluido solo por el caso de uso “Actualizar clientes” . Se debe tener en cuenta que los casos de uso incluidos deben obtenerse luego de haber realizado las especificaciones de los casos de uso, ya que en ese momento es que se determinaran cuales son los pasos que se repiten entre los diferentes casos de uso y es allí donde se determinan las relaciones de tipo **“include” .**

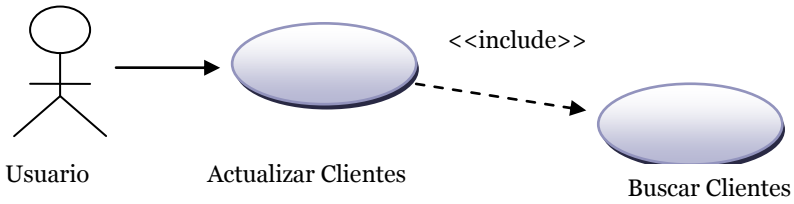


Figura 9. Error: el caso de uso que es incluido por uno solo.

Se puede encontrar bibliografía en la que se emplea la relación “use” entre casos de uso. Se debe tener en cuenta que dicha relación responde a versiones anteriores a UML versión 1.3, por lo que su utilización debe evitarse (ver **figura 10**).

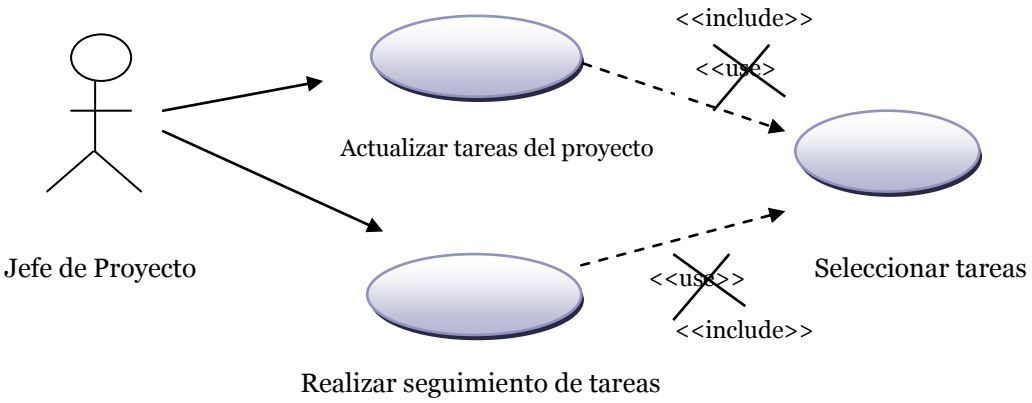


Figura 10. Error: uso de notación antigua de UML.

TEMA III: Modelado de Base de Datos y Consultas.

Tipos de Bases de datos. Bases de datos Relacionales. Diseño de una BD relacional. Repaso de conceptos de Normalización. Formas normales. Lenguaje de Consultas (SQL). Bases de datos orientadas a objetos (BDOO). Modelado de los datos. Persistencia de datos. Bases de datos objetos relacional (BDOR). Aplicaciones prácticas

Tipos de Bases de datos:

Los Sistema de gestión de bases de datos (SGDB) son un conjunto de datos relacionados entre sí y un grupo de programas para tener acceso a esos datos.

Las principales razones para emplear un SGBD son:

Tamaño: Cuando el volumen de información aumenta, es necesario algún sistema que nos facilite el intercambio de información con memoria secundaria, la búsqueda rápida, etc.

Concurrencia*:* Es necesario un mecanismo de control sobre la información cuando sobre ella pueden existir varias personas o programas interactuando, de modo que coordine sus accesos para que cada uno no invalide el trabajo realizado por el resto.

Recuperación e integridad: Cuando la información sobre la que se está trabajando es importante, es necesario algún mecanismo que se encargue de protegerla de pérdidas accidentales provocadas por fallos de energía, fallos de la propia aplicación, etc.

Persistencia: Representa la posibilidad de que la información permanezca aún después de desconectar el ordenador.

Además de lo anteriormente expuesto un SGBD suele proporcionar otras capacidades adicionales, tales como:

Distribución: o posibilidad de que la información esté almacenada en lugares diferentes. Los usuarios no necesitan conocer dónde está la información, el SGBD la encuentra para ellos.

Seguridad, que permite restringir el acceso a la información a usuarios no autorizados.

Administración: que permite a los usuarios o administradores de bases de datos examinar, controlar y ajustar el comportamiento del sistema. Permite por tanto mantener las restricciones de integridad definidas por el usuario.

Un SGBD necesita para su funcionamiento dar soporte a tres áreas básicas mediante un lenguaje para cada una de ellas:

Definición de datos (Data Definition Language). Debe permitir a los usuarios definir nuevas estructuras y tipos de información.

Manipulación de datos (Data Manipulation Language). Debe permitir acceder a las instancias de dichas estructuras de información, permitiendo incluso su modificación.

Consultas (Query). Debe permitir a los usuarios enunciar declarativamente la información que desean, siendo el propio SGBD quién se encargue de obtenerla. El soporte para la consulta suele incluir un motor que examinará la declaración de la información que se ha solicitado, elaborará un plan de ejecución para obtenerla, optimizará ese plan basándose en el conocimiento de la base de datos (índices, que información es local y cual es remota, etc.) y finalmente la ejecutará.

El criterio principal que se utiliza para clasificar los SGBD (Sistemas Gestores de Bases de Datos) es el modelo lógico en que se basan. Los modelos lógicos empleados con mayor frecuencia en los SGBD comerciales actuales son el relacional, aunque aún existe el modelo de red y el jerárquico. Algunos SGBD más modernos se basan en modelos orientados a objetos.

El ***modelo relacional*** **se** basa en el concepto matemático denominado "relación", que gráficamente se puede representar como una tabla. En el modelo relacional, los datos y las relaciones existentes entre los datos se representan mediante estas relaciones matemáticas, cada una con un nombre que es único y con un conjunto de columnas.

En el modelo relacional la base de datos es percibida por el usuario como un conjunto de tablas. Esta percepción es sólo a nivel lógico (en los niveles externo y conceptual de la arquitectura de tres niveles), ya que a nivel físico puede estar implementada mediante distintas estructuras de almacenamiento.

En el ***modelo de red*** los datos se representan como colecciones de registros y las relaciones entre los datos se representan mediante conjuntos, que son punteros en la implementación física. Los registros se organizan como un grafo: los registros son los nodos y los arcos son los conjuntos. El SGBD de red más popular es el sistema IDMS.

El ***modelo jerárquico*** es un tipo de modelo de red con algunas restricciones. De nuevo los datos se representan como colecciones de registros y las relaciones entre los datos se representan mediante conjuntos. Sin embargo, en el modelo jerárquico cada nodo puede tener un solo padre. Una base de datos jerárquica puede representarse mediante un árbol: los registros son los nodos, también denominados segmentos, y los arcos son los conjuntos. El SGBD jerárquico más importante es el sistema IMS.

La mayoría de los SGBD comerciales actuales están basados en el modelo relacional, mientras que los sistemas más antiguos estaban basados en el modelo de red o el modelo jerárquico. Estos dos últimos modelos requieren que el usuario tenga conocimiento de la estructura física de la base de datos a la que se accede, mientras que el modelo relacional proporciona una mayor independencia de datos. Se dice que el modelo relacional es declarativo (se especifica qué datos se han de obtener) y los modelos de red y jerárquico son navegacionales (se especifica cómo se deben obtener los datos).

El ***modelo orientado a objetos*** define una base de datos en términos de objetos, sus propiedades y sus operaciones. Los objetos con la misma estructura y comportamiento pertenecen a una clase, y las clases se organizan en jerarquías o grafos acíclicos. Las operaciones de cada clase se especifican en términos de procedimientos predefinidos denominados métodos. Algunos SGBD relacionales existentes en el mercado han estado

extendiendo sus modelos para incorporar conceptos orientados a objetos. A estos SGBD se les conoce como sistemas *objeto-relacionales*

Bases de datos Relacionales:

Es el modelo lógico en el que se basan la mayoría de los SGBD comerciales en uso hoy en día.

El modelo relacional se basa en dos ramas de las matemáticas: la teoría de conjuntos y la lógica de predicados de primer orden. El hecho de que el modelo relacional esté basado en la teoría de las matemáticas es lo que lo hace tan seguro y robusto. Al mismo tiempo, estas ramas de las matemáticas proporcionan los elementos básicos necesarios para crear una base de datos relacional con una buena estructura, y proporcionan las líneas que se utilizan para formular buenas metodologías de diseño.

Hay quien ofrece una cierta resistencia a estudiar complicados conceptos matemáticos para tan sólo llevar a cabo una tarea bastante concreta. Es habitual escuchar quejas sobre que las teorías matemáticas en las que se basa el modelo relacional y sus metodologías de diseño, no tienen relevancia en el mundo real o que no son prácticas. No es cierto: las matemáticas son básicas en el modelo relacional. Pero, por fortuna, no hay que aprender teoría de conjuntos o lógica de predicados de primer orden para utilizar el modelo relacional. Sería como decir que hay que saber todos los detalles de la aerodinámica para poder conducir un automóvil. Las teorías de la aerodinámica ayudan a entender cómo un automóvil puede ahorrar combustible, pero desde luego no son necesarias para manejarlo.

La teoría matemática proporciona la base para el modelo relacional y, por lo tanto, hace que el modelo sea predecible, fiable y seguro. La teoría describe los elementos básicos que se utilizan para crear una base de datos relacional y proporciona las líneas a seguir para construirla. El organizar estos elementos para conseguir el resultado deseado es lo que se denomina diseño."

El modelo relacional

En 1970, el modo en que se veían las bases de datos cambió por completo cuando E. F. Codd introdujo el modelo relacional. En aquellos momentos, el enfoque existente para la estructura de las bases de datos utilizaba punteros físicos (direcciones de disco) para relacionar registros de distintos ficheros. Si, por ejemplo, se quería relacionar un registro con un registro, se debía añadir al registro un campo conteniendo la dirección en disco del registro. Este campo añadido, un puntero físico, siempre señalaría desde el registro al registro. Codd demostró que estas bases de datos limitaban en gran medida los tipos de operaciones que los usuarios podían realizar sobre los datos. Además, estas bases de datos eran muy vulnerables a cambios en el entorno físico. Si se añadían los controladores de un nuevo disco al sistema y los datos se movían de una localización física a otra, se requería una conversión de los ficheros de datos. Estos sistemas se basaban en el modelo de red y el modelo jerárquico, los dos modelos lógicos que constituyeron la primera generación de los SGBD.

El modelo relacional representa la segunda generación de los SGBD. En él, todos los datos están estructurados a nivel lógico como tablas formadas por filas y columnas, aunque a nivel físico pueden tener una estructura completamente distinta. Un punto fuerte del modelo relacional es la sencillez de su estructura lógica. Pero detrás de esa simple estructura hay un fundamento teórico importante del que carecen los SGBD de la primera generación, lo que constituye otro punto a su favor.

Dada la popularidad del modelo relacional, muchos sistemas de la primera generación se han modificado para proporcionar una interfaz de usuario relacional, con independencia del modelo lógico que soportan (de red o jerárquico). Por ejemplo, el sistema de red IDMS ha evolucionado a IDMS/R e IDMS/SQL, ofreciendo una visión relacional de los datos.

Definiciones:

El modelo relacional se basa en el concepto matemático de *relación*, que gráficamente se representa mediante una tabla. Codd, que era un experto matemático, utilizó una terminología perteneciente a las matemáticas, en concreto de la teoría de conjuntos y de la lógica de predicados.

Una relación es una tabla con columnas y filas. Un SGBD sólo necesita que el usuario pueda percibir la base de datos como un conjunto de tablas. Esta percepción sólo se aplica a la estructura lógica de la base de datos (en el nivel externo y conceptual de la arquitectura de tres niveles ANSI-SPARC). No se aplica a la estructura física de la base de datos, que se puede implementar con distintas estructuras de almacenamiento.

Un atributo es el nombre de una columna de una relación. En el modelo relacional, las relaciones se utilizan para almacenar información sobre los objetos que se representan en la base de datos. Una relación se representa gráficamente como una tabla bidimensional en la que las filas corresponden a registros individuales y las columnas corresponden a los campos o atributos de esos registros. Los atributos pueden aparecer en la relación en cualquier orden.

Por ejemplo, la información de las oficinas de la empresa inmobiliaria se representa mediante la relación OFICINA, que tiene columnas para los atributos Onum (número de oficina), Calle, Area, Población, Teléfono y Fax. La información sobre la plantilla se representa mediante la relación PLANTILLA, que tiene columnas para los atributos Enum (número de empleado), Nombre, Apellido, Dirección, Puesto, Fecha_nac, Salario, DNI, Onum (número de la oficina a la que pertenece el empleado). A continuación se muestra una instancia de la relación OFICINA y una instancia de la relación PLANTILLA. Como se puede observar, cada columna contiene valores de un solo atributo. Por ejemplo, la columna Onum sólo contiene números de oficinas que existen.

OFICINA

Onum	Calle	Area	Población	Teléfono	Fax
O5	Enmedio, 8	Centro	Castellón	964 201 240	964 201 340
O7	Moyano, s/n	Centro	Castellón	964 215 760	964 215 670
O3	San Miguel, 1		Villarreal	964 520 250	964 520 255
O4	Trafalgar, 23	Grao	Castellón	964 284 440	964 284 420
O2	Cedre, 26		Villarreal	964 525 810	964 252 811

PLANTILLA

Enum	Nombre	Apellido	Dirección	Puesto	Fecha_nac	Salario	DNI	Onum
EL21	Amelia	Pastor	Magallanes, 15	Director	12/10/62	30000	39432212E	O5
			Castellón					
EG37	Pedro	Cubedo	Bayarri, 11	Supervisor	24/3/57	18000	38766623X	O3
			Villarreal					
EG14	Luis	Collado	Borriol, 35	Administ.	9/5/70	12000	24391223L	O3
			Villarreal					
EA9	Rita	Renau	Casalduch, 32	Supervisor	19/5/60	18000	39233190F	O7
			Castellón					
EG5	Julio	Prats	Melilla, 23	Director	19/12/50	24000	25644309X	O3
			Villarreal					
EL41	Carlos	Baeza	Herrero, 51	Supervisor	29/2/67	18000	39552133T	O5
			Castellón					

Un dominio es el conjunto de valores legales de uno o varios atributos. Los dominios constituyen una poderosa característica del modelo relacional. Cada atributo de una base de datos relacional se define sobre un dominio, pudiendo haber varios atributos definidos sobre el mismo dominio. La siguiente tabla muestra los dominios de los atributos de la relación OFICINA. Nótese que en esta relación hay dos atributos que están definidos sobre el mismo dominio, Teléfono y Fax.

Atributo	*Nombre del Dominio*	*Descripción*	*Definición*
Onum	NUM_OFICINA	Posibles valores de número de oficina	3 caracteres;
			rango O1-O99
Calle	NOM_CALLE	Nombres de calles de España	25 caracteres

Area	NOM_AREA	Nombres de áreas de las poblaciones de España	20 caracteres
Población	NOM_POBLACION	Nombres de las poblaciones de España	15 caracteres
Teléfono	NUM_TEL_FAX	Números de teléfono de España	9 caracteres
Fax	NUM_TEL_FAX	Números de teléfono de España	9 caracteres

El concepto de dominio es importante porque permite que el usuario defina, en un lugar común, el significado y la fuente de los valores que los atributos pueden tomar. Esto hace que haya más información disponible para el sistema cuando éste va a ejecutar una operación relacional, de modo que las operaciones que son semánticamente incorrectas, se pueden evitar. Por ejemplo, no tiene sentido comparar el nombre de una calle con un número de teléfono, aunque los dos atributos sean cadenas de caracteres. Sin embargo, el importe mensual del alquiler de un inmueble no estará definido sobre el mismo dominio que el número de meses que dura el alquiler, pero sí tiene sentido multiplicar los valores de ambos dominios para averiguar el importe total al que asciende el alquiler. Los SGBD relacionales no ofrecen un soporte completo de los dominios ya que su implementación es extremadamente compleja.

Una tupla es una fila de una relación. Los elementos de una relación son las tuplas o filas de la tabla. En la relación OFICINA, cada tupla tiene seis valores, uno para cada atributo. Las tuplas de una relación no siguen ningún orden.

El grado de una relación es el número de atributos que contiene. La relación OFICINA es de grado seis porque tiene seis atributos. Esto quiere decir que cada fila de la tabla es una tupla con seis valores. El grado de una relación no cambia con frecuencia.

La cardinalidad de una relación es el número de tuplas que contiene. Ya que en las relaciones se van insertando y borrando tuplas a menudo, la cardinalidad de las mismas varía constantemente.

Una base de datos relacional es un conjunto de relaciones normalizadas.

Propiedades de las relaciones

Las relaciones tienen las siguientes características:

- ✓ Cada relación tiene un nombre y éste es distinto del nombre de todas las demás.
- ✓ Los valores de los atributos son atómicos: en cada tupla, cada atributo toma un solo valor.

Se dice que las relaciones están *normalizadas*.

- ✓ No hay dos atributos que se llamen igual.
- ✓ El orden de los atributos no importa: los atributos no están ordenados.

- ✓ Cada tupla es distinta de las demás: no hay tuplas duplicadas.
- ✓ El orden de las tuplas no importa: las tuplas no están ordenadas.

Claves:

Ya que en una relación no hay tuplas repetidas, éstas se pueden distinguir unas de otras, es decir, se pueden identificar de modo único. La forma de identificarlas es mediante los valores de sus atributos.

Una *superclave* es un atributo o un conjunto de atributos que identifican de modo único las tuplas de una relación.

Una *clave candidata* es una superclave en la que ninguno de sus subconjuntos es una superclave de la relación. El atributo o conjunto de atributos de la relación es una clave candidata para si y sólo si satisface las siguientes propiedades:

· *Unicidad:* nunca hay dos tuplas en la relación con el mismo valor.

· *Irreducibilidad (minimalidad):* ningún subconjunto de tiene la propiedad de unicidad, es decir, no se pueden eliminar componentes de sin destruir la unicidad.

Cuando una clave candidata está formada por más de un atributo, se dice que es una *clave compuesta*. Una relación puede tener varias claves candidatas. Por ejemplo, en la relación OFICINA, el atributo Población no es una clave candidata ya que puede haber varias oficinas en una misma población. Sin embargo, ya que la empresa asigna un código único a cada oficina, el atributo Onum sí es una clave candidata de la relación OFICINA. También son claves candidatas de esta relación los atributos Teléfono y Fax.

La *clave primaria* de un relación es aquella clave candidata que se escoge para identificar sus tuplas de modo único. Ya que una relación no tiene tuplas duplicadas, siempre hay una clave candidata y, por lo tanto, la relación siempre tiene clave primaria. En el peor caso, la clave primaria estará formada por todos los atributos de la relación, pero normalmente habrá un pequeño subconjunto de los atributos que haga esta función.

Las claves candidatas que no son escogidas como clave primaria son denominadas *claves alternativas*. Por ejemplo, la clave primaria de la relación OFICINA es el atributo Onum, siendo Teléfono y Fax dos claves alternativas. En la relación VISITA sólo hay una clave candidata formada por los atributos Qnum e Inum, por lo que esta clave candidata es la clave primaria.

Una *clave ajena* es un atributo o un conjunto de atributos de una relación cuyos valores coinciden con los valores de la clave primaria de alguna otra relación (puede ser la misma). Las claves ajenas representan *relaciones entre datos*. El atributo Onum de PLANTILLA relaciona a cada empleado con la oficina a la que pertenece. Este atributo es una clave ajena cuyos valores hacen referencia al atributo Onum, clave primaria de OFICINA. Se dice que un valor de clave ajena representa una *referencia* a la tupla que contiene el mismo valor en su clave primaria (*tupla referenciada*).

Reglas de integridad

Una vez definida la estructura de datos del modelo relacional, pasamos a estudiar las reglas de integridad que los datos almacenados en dicha estructura deben cumplir para garantizar que son correctos.

Al definir cada atributo sobre un dominio se impone una restricción sobre el conjunto de valores permitidos para cada atributo. A este tipo de restricciones se les denomina *restricciones de dominios*. Hay además dos reglas de integridad muy importantes que son restricciones que se deben cumplir en todas las bases de datos relacionales y en todos sus estados o instancias (las reglas se deben cumplir todo el tiempo). Estas reglas son la *regla de integridad de entidades* y la *regla de integridad referencial*. Antes de definirlas, es preciso conocer el concepto de *nulo*.

Nulos

Cuando en una tupla un atributo es desconocido, se dice que es *nulo*. Un nulo no representa el valor cero ni la cadena vacía, éstos son valores que tienen significado. El nulo implica ausencia de información, bien porque al insertar la tupla se desconocía el valor del atributo, o bien porque para dicha tupla el atributo no tiene sentido.

Ya que los nulos no son valores, deben tratarse de modo diferente, lo que causa problemas de implementación. De hecho, no todos los SGBD relacionales soportan los nulos.

Regla de integridad de entidades

La primera regla de integridad se aplica a las claves primarias de las relaciones base: *ninguno de los atributos que componen la clave primaria puede ser nulo.*

Por definición, una clave primaria es un identificador irreducible que se utiliza para identificar de modo único las tuplas. Que es irreducible significa que ningún subconjunto de la clave primaria sirve para identificar las tuplas de modo único. Si se permite que parte de la clave primaria sea nula, se está diciendo que no todos sus atributos son necesarios para distinguir las tuplas, con lo que se contradice la irreducibilidad.

Nótese que esta regla sólo se aplica a las relaciones base y a las claves primarias, no a las claves alternativas.

Regla de integridad referencial

La segunda regla de integridad se aplica a las claves ajenas: *si en una relación hay alguna clave ajena, sus valores deben coincidir con valores de la clave primaria a la que hace referencia, o bien, deben ser completamente nulos*.

La regla de integridad referencial se enmarca en términos de estados de la base de datos: indica lo que es un estado ilegal, pero no dice cómo puede evitarse. La cuestión es ¿qué hacer si estando en un estado legal, llega una petición para realizar una operación que conduce a un estado ilegal? Existen dos opciones: *rechazar* la operación, o bien *aceptar* la operación y realizar operaciones adicionales compensatorias que conduzcan a un estado legal.

Por lo tanto, para cada clave ajena de la base de datos habrá que contestar a tres preguntas:

· ***Regla de los nulos****:* ¿Tiene sentido que la clave ajena acepte nulos?

· ***Regla de borrado****:* ¿Qué ocurre si se intenta borrar la tupla referenciada por la clave ajena?

Restringir: no se permite borrar la tupla referenciada.

Propagar: se borra la tupla referenciada y se propaga el borrado a las tuplas que la referencian mediante la clave ajena.

Anular: se borra la tupla referenciada y las tuplas que la referenciaban ponen a nulo la clave ajena (sólo si acepta nulos).

· ***Regla de modificación:*** ¿Qué ocurre si se intenta modificar el valor de la clave primaria de la tupla referenciada por la clave ajena?

Restringir: no se permite modificar el valor de la clave primaria de la tupla referenciada.
Propagar: se modifica el valor de la clave primaria de la tupla referenciada y se propaga la modificación a las tuplas que la referencian mediante la clave ajena.

Anular: se modifica la tupla referenciada y las tuplas que la referenciaban ponen a nulo la clave ajena (sólo si acepta nulos).

Diseño de una BD relacional.

"El diseño de una base de datos es el proceso por el que se determina la organización de la misma, incluidos su estructura, contenido y las aplicaciones que se han de desarrollar". Durante mucho tiempo, el diseño de bases de datos fue considerado una tarea para expertos: más un arte que una ciencia. Sin embargo, se ha progresado mucho en el diseño de bases de datos y éste se considera ahora una disciplina estable, con métodos y técnicas propios. Debido a la creciente aceptación de las bases de datos por parte de la industria y el gobierno en el plano comercial, y a una variedad de aplicaciones científicas y técnicas, el diseño de bases de datos desempeña un papel central en el empleo de los recursos de información en la mayoría de las organizaciones. El diseño de bases de datos ha pasado a constituir parte de la formación general de los informáticos, en el mismo nivel que la capacidad de construir algoritmos usando un lenguaje de programación convencional."

"Las últimas dos décadas se han caracterizado por un fuerte crecimiento en el número e importancia de las aplicaciones de bases de datos. Las bases de datos son componentes esenciales de los sistemas de información, usadas rutinariamente en todos los computadores". El diseño de bases de datos se ha convertido en una actividad popular, desarrollada no sólo por profesionales sino también por no especialistas.

A finales de la década de 1960, cuando las bases de datos entraron por primera vez en el mercado del software, los diseñadores de bases de datos actuaban como artesanos, con herramientas muy primitivas: diagramas de bloques y estructuras de registros eran los formatos comunes para las especificaciones, y el diseño de bases de datos se confundía frecuentemente con la implantación de las bases de datos. Esta situación ahora ha cambiado: los métodos y modelos de diseño de bases de datos han evolucionado paralelamente con el progreso de la tecnología en los sistemas de bases de datos. Se ha entrado en la era de los sistemas relacionales de bases de datos, que ofrecen poderosos lenguajes de consulta, herramientas para el desarrollo de aplicaciones e interfaces amigables con los usuarios. La tecnología de bases de datos cuenta ya con un marco teórico, que incluye la teoría relacional de datos, procesamiento y optimización de consultas, control de concurrencia, gestión de transacciones y recuperación, etc.

Según ha avanzado la tecnología de bases de datos, así se han desarrollado las metodologías y técnicas de diseño. Se ha alcanzado un consenso, por ejemplo, sobre la descomposición del proceso de diseño en fases, sobre los principales objetivos de cada fase y sobre las técnicas para conseguir estos objetivos."

"Desafortunadamente, las metodologías de diseño de bases de datos no son muy populares; la mayoría de las organizaciones y de los diseñadores individuales confía muy poco en las metodologías para llevar a cabo el diseño y esto se considera, con frecuencia, una de las principales causas de fracaso en el desarrollo de los sistemas de información. Debido a la falta de enfoques estructurados para el diseño de bases de datos, a menudo se subestiman el tiempo o los recursos necesarios para un proyecto de bases de datos, las bases de datos son inadecuadas o ineficientes en relación a las demandas de la aplicación, la documentación es limitada y el mantenimiento es difícil.

Muchos de estos problemas se deben a la falta de una claridad que permita entender la naturaleza exacta de los datos, a un nivel conceptual y abstracto. En muchos casos, los datos se describen desde el comienzo del proyecto en términos de las estructuras finales de almacenamiento; no se da peso a un entendimiento de las propiedades estructurales de los datos que sea independiente de los detalles de la realización."

Metodología de diseño de bases de datos

El diseño de una base de datos es un proceso complejo que abarca decisiones a muy distintos niveles. La complejidad se controla mejor si se descompone el problema en subproblemas y se resuelve cada uno de estos sub-problemas independientemente, utilizando técnicas

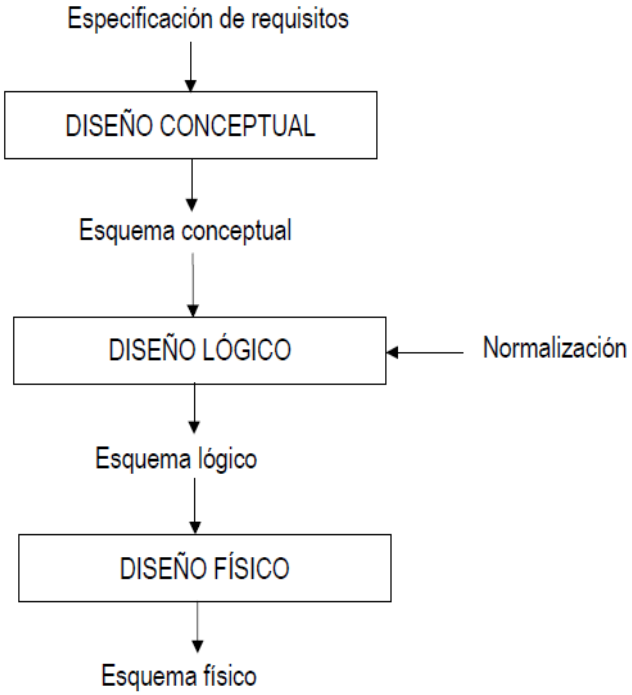
específicas. Así, el diseño de una base de datos se descompone en diseño conceptual, diseño lógico y diseño físico.

El diseño conceptual parte de las especificaciones de requisitos de usuario y su resultado es el esquema conceptual de la base de datos. Un *esquema conceptual* es una descripción de alto nivel de la estructura de la base de datos, independientemente del SGBD que se vaya a utilizar para manipularla. Un *modelo conceptual* es un lenguaje que se utiliza para describir esquemas conceptuales. El objetivo del diseño conceptual es describir el contenido de información de la base de datos y no las estructuras de almacenamiento que se necesitarán para manejar esta información.

El diseño lógico parte del esquema conceptual y da como resultado un esquema lógico. Un *esquema lógico* es una descripción de la estructura de la base de datos en términos de las estructuras de datos que puede procesar un tipo de SGBD. Un *modelo lógico* es un lenguaje usado para especificar esquemas lógicos (modelo relacional, modelo de red, etc.). El diseño lógico depende del tipo de SGBD que se vaya a utilizar, no depende del producto concreto.

El diseño físico parte del esquema lógico y da como resultado un esquema físico. Un *esquema físico* es una descripción de la implementación de una base de datos en memoria secundaria: las estructuras de almacenamiento y los métodos utilizados para tener un acceso eficiente a los datos. Por ello, el diseño físico depende del SGBD concreto y el esquema físico se expresa mediante su lenguaje de definición de datos.

Gráficamente:



Modelos de datos

Un *modelo de datos* es una serie de conceptos que puede utilizarse para describir un conjunto de datos y las operaciones para manipularlos. Hay dos tipos de modelos de datos: los modelos conceptuales y los modelos lógicos. Los *modelos conceptuales* se utilizan para representar la realidad a un alto nivel de abstracción. Mediante los modelos conceptuales se puede construir una descripción de la realidad fácil de entender. En los *modelos lógicos*, las descripciones de los datos tienen una correspondencia sencilla con la estructura física de la base de datos.

En el diseño de bases de datos se usan primero los modelos conceptuales para lograr una descripción de alto nivel de la realidad, y luego se transforma el esquema conceptual en un esquema lógico. El motivo de realizar estas dos etapas es la dificultad de abstraer la estructura de una base de datos que presente cierta complejidad. Un *esquema* es un conjunto de representaciones lingüísticas o gráficas que describen la estructura de los datos de interés.

Los modelos conceptuales deben ser buenas herramientas para representar la realidad, por lo que deben poseer las siguientes cualidades:

- *Expresividad*: deben tener suficientes conceptos para expresar perfectamente la realidad.
- *Simplicidad*: deben ser simples para que los esquemas sean fáciles de entender.
- *Minimalidad*: cada concepto debe tener un significado distinto.
- *Formalidad*: todos los conceptos deben tener una interpretación única, precisa y bien definida.

En general, un modelo no es capaz de expresar todas las propiedades de una realidad determinada, por lo que hay que añadir aserciones que complementen el esquema.

El modelo entidad-relación

El modelo entidad-relación es el modelo conceptual más utilizado para el diseño conceptual de bases de datos. Fue introducido por Peter Chen en 1976. El modelo entidad-relación está formado por un conjunto de conceptos que permiten describir la realidad mediante un conjunto de representaciones gráficas y lingüísticas.

Originalmente, el modelo entidad-relación sólo incluía los conceptos de entidad, relación y atributo. Más tarde, se añadieron otros conceptos, como los atributos compuestos y las jerarquías de generalización, en lo que se ha denominado *modelo entidad-relación extendido*.

Entidad

Cualquier tipo de objeto o concepto sobre el que se recoge información: cosa, persona, concepto abstracto o suceso. Por ejemplo: coches, casas, empleados, clientes, empresas, oficios, diseños de productos, conciertos, excursiones, etc. Las entidades se representan

gráficamente mediante rectángulos y su nombre aparece en el interior. Un nombre de entidad sólo puede aparecer una vez en el esquema conceptual.

Hay dos tipos de entidades: fuertes y débiles. Una *entidad débil* es una entidad cuya existencia depende de la existencia de otra entidad. Una *entidad fuerte* es una entidad que no es débil.

Relación (interrelación)

Es una correspondencia o asociación entre dos o más entidades. Cada relación tiene un nombre que describe su función. Las relaciones se representan gráficamente mediante rombos y su nombre aparece en el interior.

Las entidades que están involucradas en una determinada relación se denominan *entidades participantes*. El número de participantes en una relación es lo que se denomina *grado* de la relación. Por lo tanto, una relación en la que participan dos entidades es una relación *binaria*; si son tres las entidades participantes, la relación es *ternaria*; etc.

Una *relación recursiva* es una relación donde la misma entidad participa más de una vez en la relación con distintos papeles. El nombre de estos papeles es importante para determinar la función de cada participación.

La *cardinalidad* con la que una entidad participa en una relación especifica el número mínimo y el número máximo de correspondencias en las que puede tomar parte cada ocurrencia de dicha entidad. La participación de una entidad en una relación es *obligatoria (total)* si la existencia de cada una de sus ocurrencias requiere la existencia de, al menos, una ocurrencia de la otra entidad participante. Si no, la participación es *opcional (parcial)*. Las reglas que definen la cardinalidad de las relaciones son las *reglas de negocio*.

A veces, surgen problemas cuando se está diseñado un esquema conceptual. Estos problemas, denominados *trampas*, suelen producirse a causa de una mala interpretación en el significado de alguna relación, por lo que es importante comprobar que el esquema conceptual carece de dichas trampas. En general, para encontrar las trampas, hay que asegurarse de que se entiende completamente el significado de cada relación. Si no se entienden las relaciones, se puede crear un esquema que no represente fielmente la realidad.

Una de las trampas que pueden encontrarse ocurre cuando el esquema representa una relación entre entidades, pero el camino entre algunas de sus ocurrencias es ambiguo. El modo de resolverla es reestructurando el esquema para representar la asociación entre las entidades correctamente.

Otra de las trampas sucede cuando un esquema sugiere la existencia de una relación entre entidades, pero el camino entre una y otra no existe para algunas de sus ocurrencias. En este caso, se produce una pérdida de información que se puede subsanar introduciendo la relación que sugería el esquema y que no estaba representada.

Atributo

Es una característica de interés o un hecho sobre una entidad o sobre una relación. Los atributos representan las propiedades básicas de las entidades y de las relaciones. Toda la información extensiva es portada por los atributos. Gráficamente, se representan mediante bolitas que cuelgan de las entidades o relaciones a las que pertenecen.

Cada atributo tiene un conjunto de valores asociados denominado *dominio*. El dominio define todos los valores posibles que puede tomar un atributo. Puede haber varios atributos definidos sobre un mismo dominio.

Los atributos pueden ser simples o compuestos. Un *atributo simple* es un atributo que tiene un solo componente, que no se puede dividir en partes más pequeñas que tengan un significado propio. Un *atributo compuesto* es un atributo con varios componentes, cada uno con un significado por sí mismo. Un grupo de atributos se representa mediante un atributo compuesto cuando tienen afinidad en cuanto a su significado, o en cuanto a su uso. Un atributo compuesto se representa gráficamente mediante un óvalo.

Los atributos también pueden clasificarse en monovalentes o polivalentes. Un *atributo monovalente* es aquel que tiene un solo valor para cada ocurrencia de la entidad o relación a la que pertenece. Un *atributo polivalente* es aquel que tiene varios valores para cada ocurrencia de la entidad o relación a la que pertenece. A estos atributos también se les denomina *multivaluados*, y pueden tener un número máximo y un número mínimo de valores. La *cardinalidad* de un atributo indica el número mínimo y el número máximo de valores que puede tomar para cada ocurrencia de la entidad o relación a la que pertenece.

Por último, los atributos pueden ser derivados. Un *atributo derivado* es aquel que representa un valor que se puede obtener a partir del valor de uno o varios atributos, que no necesariamente deben pertenecer a la misma entidad o relación.

Repaso de conceptos de Normalización

La normalización es una técnica para diseñar la estructura lógica de los datos de un sistema de información en el modelo relacional, desarrollada por E. F. Codd en 1972. Es una estrategia de diseño de abajo a arriba: se parte de los atributos y éstos se van agrupando en relaciones (tablas) según su afinidad. Aquí no se utilizará la normalización como una técnica de diseño de bases de datos, sino como una etapa posterior a la correspondencia entre el esquema conceptual y el esquema lógico, que elimine las dependencias entre atributos no deseadas. Las ventajas de la normalización son las siguientes:

- Evita anomalías en inserciones, modificaciones y borrados.
- Mejora la independencia de datos.
- No establece restricciones artificiales en la estructura de los datos.

Uno de los conceptos fundamentales en la normalización es el de *dependencia funcional*. Una dependencia funcional es una relación entre atributos de una misma relación (tabla).

Las dependencias funcionales del sistema se escriben utilizando una flecha, de la siguiente manera:

FechaDeNacimiento $\rightarrow$ *Edad*

De la normalización (lógica) a la implementación (física o real) puede ser sugerible tener éstas dependencias funcionales para lograr la eficiencia en las tablas.

Sean *X, Y, Z* tres atributos (o grupos de atributos) de la misma entidad. Si *Y* depende funcionalmente de *X* y *Z* de *Y*, pero *X* no depende funcionalmente de *Y*, se dice entonces que *Z* depende transitivamente de *X*. Simbólicamente sería:

X $\rightarrow$ *Y* $\rightarrow$ *Z* entonces *X* $\rightarrow$ *Z*

FechaDeNacimiento $\rightarrow$ *Edad*

Edad $\rightarrow$ *Conducir*

FechaDeNacimiento $\rightarrow$ *Edad* $\rightarrow$ *Conducir*

Entonces tenemos que *FechaDeNacimiento* determina a *Edad* y la *Edad* determina a *Conducir*, indirectamente podemos saber a través de *FechaDeNacimiento* a *Conducir* (Una persona necesita ser mayor de cierta edad para poder conducir un automóvil, por eso se utiliza este ejemplo).

En el proceso de normalización se debe ir comprobando que cada relación (tabla) cumple una serie de reglas que se basan en la clave primaria y las dependencias funcionales. Cada regla que se cumple aumenta el grado de normalización. Si una regla no se cumple, la relación se debe descomponer en varias relaciones que sí la cumplan.

La normalización se lleva a cabo en una serie pasos. Cada paso corresponde a una forma normal que tiene unas propiedades. Conforme se va avanzando en la normalización, las relaciones tienen un formato más estricto (más fuerte) y, por lo tanto, son menos vulnerables a las anomalías de actualización. El modelo relacional sólo requiere un conjunto de relaciones en primera forma normal. Las restantes formas normales son opcionales. Sin embargo, para evitar las anomalías de actualización, es recomendable llegar al menos a la tercera forma normal.

Formas normales

Primera forma normal (1FN)

Una relación está en primera forma normal si, y sólo si, todos los dominios de la misma contienen valores atómicos, es decir, no hay grupos repetitivos. Si se ve la relación

gráficamente como una tabla, estará en 1FN si tiene un solo valor en la intersección de cada fila con cada columna.

Si una relación no está en 1FN, hay que eliminar de ella los grupos repetitivos. Un grupo repetitivo será el atributo o grupo de atributos que tiene múltiples valores para cada tupla de la relación. Hay dos formas de eliminar los grupos repetitivos. En la primera, se repiten los atributos con un solo valor para cada valor del grupo repetitivo. De este modo, se introducen redundancias ya que se duplican valores, pero estas redundancias se eliminarán después mediante las restantes formas normales. La segunda forma de eliminar los grupos repetitivos consiste en poner cada uno de ellos en una relación aparte, heredando la clave primaria de la relación en la que se encontraban.

Un conjunto de relaciones se encuentra en 1FN si ninguna de ellas tiene grupos repetitivos.

Segunda forma normal (2FN)

Una relación está en segunda forma normal si, y sólo si, está en 1FN y, además, cada atributo que no está en la clave primaria es completamente dependiente de la clave primaria.

La 2FN se aplica a las relaciones que tienen claves primarias compuestas por dos o más atributos. Si una relación está en 1FN y su clave primaria es simple (tiene un solo atributo), entonces también está en 2FN. Las relaciones que no están en 2FN pueden sufrir anomalías cuando se realizan actualizaciones.

Para pasar una relación en 1FN a 2FN hay que eliminar las dependencias parciales de la clave primaria. Para ello, se eliminan los atributos que son funcionalmente dependientes y se ponen en una nueva relación con una copia de su determinante (los atributos de la clave primaria de los que dependen).

Tercera forma normal (3FN)

Una relación está en tercera forma normal si, y sólo si, está en 2FN y, además, cada atributo que no está en la clave primaria no depende transitivamente de la clave primaria. La dependencia es transitiva si existen las dependencias , , siendo , , atributos o conjuntos de atributos de una misma relación.

Aunque las relaciones en 2FN tienen menos redundancias que las relaciones en 1FN, todavía pueden sufrir anomalías frente a las actualizaciones. Para pasar una relación de 2FN a 3FN hay que eliminar las dependencias transitivas. Para ello, se eliminan los atributos que dependen transitivamente y se ponen en una nueva relación con una copia de su determinante (el atributo o atributos no clave de los que dependen).

Forma normal de Boyce-Codd (BCFN)

Una relación está en la forma normal de Boyce-Codd si, y sólo si, todo determinante es una clave candidata.

La 2FN y la 3FN eliminan las dependencias parciales y las dependencias transitivas de la clave primaria. Pero este tipo de dependencias todavía pueden existir sobre otras claves candidatas, si éstas existen. La BCFN es más fuerte que la 3FN, por lo tanto, toda relación en BCFN está en 3FN.

La violación de la BCFN es poco frecuente ya que se da bajo ciertas condiciones que raramente se presentan. Se debe comprobar si una relación viola la BCFN si tiene dos o más claves candidatas compuestas que tienen al menos un atributo en común.

Lenguaje de Consultas (SQL)

Son varios los lenguajes utilizados por los SGBD relacionales para manejar las relaciones. Algunos de ellos son *procedurales*, lo que quiere decir que el usuario dice al sistema exactamente cómo debe manipular los datos. Otros son *no procedurales*, que significa que el usuario dice qué datos necesita, en lugar de decir cómo deben obtenerse.

El mas popular es el L**enguaje de Consulta Estructurado** o **SQL** (por sus siglas en inglés *structured query language*) es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones en estas. Una de sus características es el manejo del álgebra y el cálculo relacional permitiendo efectuar consultas con el fin de recuperar -de una forma sencilla- información de interés de una base de datos, así como también hacer cambios sobre ella.

Características:

El SQL es un lenguaje de acceso a bases de datos que explota la flexibilidad y potencia de los sistemas relacionales permitiendo gran variedad de operaciones.

Es un lenguaje declarativo de "alto nivel" o "no procedural" (especifica qué es lo que se quiere y no cómo conseguirlo, por lo que una sentencia no establece explícitamente un orden de ejecución), que gracias a su fuerte base teórica y su orientación al manejo de conjuntos de registros, y no a registros individuales, permite una alta productividad en codificación y la orientación a objetos. De esta forma una sola sentencia puede equivaler a uno o más programas que se utilizarían en un lenguaje de bajo nivel orientado a registros.

El orden de ejecución interno de una sentencia puede afectar gravemente a la eficiencia del SGBD, por lo que se hace necesario que éste lleve a cabo una optimización antes de su ejecución. Muchas veces, el uso de índices acelera una instrucción de consulta, pero ralentiza la actualización de los datos. Dependiendo del uso de la aplicación, se priorizará el acceso indexado o una rápida actualización de la

información. La optimización difiere sensiblemente en cada motor de base de datos y depende de muchos factores.

Componentes del SQL

El lenguaje SQL está compuesto por comandos, cláusulas, operadores y funciones de agregado. Estos elementos se combinan en las instrucciones para crear, actualizar y manipular las bases de datos.

Las sentencias de SQL se clasifican según su finalidad e 3 grupos:

DCL (Data Control Language): Incluye órdenes para manejar la seguridad de los datos y de la base de datos. Permite crear roles y establecer permisos. Sentencias: GRANT, REVOKE

DDL (Data Definition Language): Incluye ordenes para definir, modificar o borrar objetos de la base de datos. Sentencias: CREATE, DROP, ALTER

DML (Data Manipulation Language): Permite actualizar y recuperar los datos almacenados en la base de datos. Este tipo de instrucciones son las que se desarrollaran en este capítulo. Sentencias: SELECT, INSERT, UPDATE, DELETE

Comandos DCL:

Comando	Descripción
GRANT	Utilizado para dar permisos a usuarios sobre los objetos de la base de datos
REVOKE	Utilizado para quitar permisos a usuarios sobre los objetos de la base de datos

Comandos DDL:

Comando	Descripción
CREATE	Utilizado para crear nuevas tablas, campos e índices
DROP	Empleado para eliminar tablas e índices
ALTER	Utilizado para modificar las tablas agregando campos o cambiando la definición de los campos.

Comandos DML:

Comando	Descripción
SELECT	Utilizado para consultar registros de la base de datos que satisfagan un criterio determinado
INSERT	Utilizado para cargar lotes de datos en la base de datos en una única operación.
UPDATE	Utilizado para modificar los valores de los campos y registros especificados
DELETE	Utilizado para eliminar registros de una tabla de una base de datos

Cláusulas

Las cláusulas son condiciones de modificación utilizadas para definir los datos que desea seleccionar o manipular.

Cláusula	Descripción
FROM	Utilizada para especificar la tabla de la cual se van a seleccionar los registros
WHERE	Utilizada para especificar las condiciones que deben reunir los registros que se van a seleccionar
GROUP BY	Utilizada para separar los registros seleccionados en grupos específicos
HAVING	Utilizada para expresar la condición que debe satisfacer cada grupo
ORDER BY	Utilizada para ordenar los registros seleccionados de acuerdo con un orden específico

Operadores lógicos

Operador	Uso
AND	Es el "y" lógico. Evalua dos condiciones y devuelve un valor de verdad sólo si ambas son ciertas.
OR	Es el "o" lógico. Evalúa dos condiciones y devuelve un valor de verdar si alguna de las dos es cierta.
NOT	Negación lógica. Devuelve el valor contrario de la expresión.

Operadores de Comparación

Operador	Uso
<	Menor que
>	Mayor que
<>	Distinto de
<=	Menor ó Igual que
>=	Mayor ó Igual que
=	Igual que
BETWEEN	Utilizado para especificar un intervalo de valores.
LIKE	Utilizado en la comparación de un modelo
IN	Utilizado para especificar registros de una base de datos

Funciones de Agregado:

Las funciones de agregado se usan dentro de una cláusula SELECT en grupos de registros para devolver un único valor que se aplica a un grupo de registros.

Función	Descripción
AVG	Utilizada para calcular el promedio de los valores de un campo determinado
COUNT	Utilizada para devolver el número de registros de la selección
SUM	Utilizada para devolver la suma de todos los valores de un campo determinado
MAX	Utilizada para devolver el valor más alto de un campo especificado
MIN	Utilizada para devolver el valor más bajo de un campo especificado

Ejemplos:

```
select * from sys.tables

Select  Name, ListPrice, ListPrice*10/100 AS Descuento FROM
Production.Product

select distinct GroupName from   HumanResources.Department
```

```
SELECT DISTINCT
       CASE
            WHEN Color  IS NULL  THEN 'SIN DATOS'
            ELSE Color
       END  Colores
FROM Production.Product


SELECT count(*) 'Cantidad de productos', sum(Weight) 'Peso total',
       avg(Weight) 'Promedio '
FROM Production.Product
SELECT p.ProductSubcategoryID, count(*) Cantidad
FROM Production.Product p
GROUP BY p.ProductSubcategoryID
SELECT c.ProductCategoryID, c.Name Categoria, p.ProductSubcategoryID,
            s.Name Subcategoria, count(*) Cantidad
FROM Production.Product p
INNER JOIN Production.ProductSubcategory s
ON p.ProductSubcategoryID = s.ProductSubcategoryID
INNER JOIN Production.ProductCategory c
ON s.ProductCategoryID = c.ProductCategoryID
GROUP BY c.ProductCategoryID, c.Name , p.ProductSubcategoryID, s.Name
```

Bases de datos orientadas a objetos (BDOO).

Las bases de datos orientadas a objetos están diseñadas para simplificar la programación orientada a objetos. Almacenan los objetos directamente en la base de datos, y emplean las mismas estructuras y relaciones que los lenguajes de programación orientados a objetos. Soporta el paradigma orientado a objetos almacenando datos y métodos. Está diseñada para ser eficaz, desde el punto de vista físico, para almacenar objetos complejos.

Evita el acceso a los datos; esto es mediante los métodos almacenados en ella. Es más segura ya que no permite tener acceso a los datos (objetos); esto debido a que para poder entrar se tiene que hacer por los métodos que haya utilizado el programador.

Para los usuarios tradicionales de bases de datos, esto quiere decir que pueden tratar directamente con objetos, no teniendo que hacer la traducción a tablas o registros. Para los programadores de aplicaciones, esto quiere decir que sus objetos se conservan, pueden ser gestionados aunque su tamaño sea muy grande, pueden ser compartidos entre múltiples usuarios, y se mantienen tanto su integridad como sus relaciones.

Las técnicas OO utilizan los mismos modelos conceptuales para el análisis, diseño y construcción. La tecnología de las BDOO da un paso más hacia la unificación, el modelo conceptual de la base de datos OO es igual al del resto del mundo OO, en lugar de utilizar tablas por relación independientes como SQL.

El uso del mismo modelo conceptual para todos los aspectos del desarrollo simplifica éste, particularmente con las herramientas CASE OO; mejora la comunicación entre usuarios, analistas y programadores, además de que reduce las posibilidades de error.

Muchos vendedores de bases de datos relacionales están incorporando capas de objetos sobre sus motores relacionales (basados en tablas), que son bastante útiles para la integración con aplicaciones y bases de datos de objetos, y que pueden ayudar en la generación de algunos códigos de ensamblaje/ desensamblaje.

Producto	Proveedor
Gemstone	Servio Corporation, Alameda,CA
Itasca	Itasca Systems,Inc.,Minneapolis,MN
Objectivity	Objectivity,Menlo Park,Ca
Object Store	Object Design,Inc.,Burlington,MA
Ontos	Ontos Inc.,Bellerica,MA
Versant	Versant Object Technology,Menlo Park,CA

Seis Productos de BDOO y sus Proveedores.

El paradigma de programación orientada a objetos incluye el concepto de tipos abstractos de datos en lenguajes de programación. Las declaraciones de tipos abstractos de datos explícitamente se definen públicos y privadas en algunas porciones de la estructura de datos, u objetos. Los tipos abstractos de datos en un lenguaje orientado a objeto, son implementados en clases, es decir encapsula porciones

privadas de datos del objeto con procedimientos públicos, llamados métodos. El argumento para encapsulación es uno de los más simples en la construcción y mantenimiento de programas a través de modularización. Un objeto es como una caja negra, que puede ser construida y modificada independientemente del resto del sistema, tan grande como una interfaz pública (método) en la cual las definiciones no cambian.

No hay un sólo paradigma orientado a objeto, y por lo tanto hay una variedad de modelos y como consecuencia diferentes estándares. Generalmente, los lenguajes de programación orientados a objeto parten de conceptos comunes además de encapsulación, en particular el uso de jerarquías de tipos de objetos con herencias en sus atributos y métodos. De cualquier modo, las características específicas varían, y pueden regular la definición estricta de encapsulación provista por tipos abstractos de datos - que los procedimientos son públicos, cuando los datos son privados. El tipo de modelado también influye en la manera como son manejados los DBMS´s Orientados a Objeto.

Definiciones:

Objeto: es cualquier cosa real ó abstracta acerca de la cual almacenamos datos y los métodos que controlan dichos datos. Por ej. En una empresa EMPLEADO se aplica a los objetos que son personas empleadas por alguna organización alguna INSTANCIA podría ser Juan Pérez, María Sánchez etc.

Tipo de Objeto: es una categoría de objeto. Ej.: EMPLEADO.

Un objeto es una Instancia de un tipo de objeto. PERSONA (Juan Pérez) Encapsulado: es el resultado (o acto) de ocultar los detalles de implantación de un objeto respecto de su usuario.

Una Solicitud: invoca una operación específica, con uno ó más objetos como parámetros.
Es decir, es para que se lleve acabo la operación indicada y que se produzca el resultado.
En consecuencia las implantaciones se refieren a los objetos como solicitudes.

Clase: es una implantación de un tipo de objetos.

Especifica una estructura de datos y los métodos operativos permisibles que se aplican a cada uno de sus objetos.

Herencia: Una clase implanta el tipo de objeto.

Una Subclase hereda propiedades de su clase padre, una subclase puede heredar la estructura y los métodos ó algunos de los métodos.
En las BDOO los datos están encapsulados y se dice que estos son activos más que pasivos; debido a que por ejemplo; La clase mayor detecta si tiene un hijo (objeto) más o uno menos, es por esto que se dice que están activos ya que cuentan los hijos u objetos que tiene.

Características de los SGBDOO

Un SGBDOO debe satisfacer dos criterios: Ser un sistema orientado a objetos, y ser un sistema de gestión de bases de datos. El primer criterio se traduce en ocho características generales [BOO94]: abstracción, encapsulación, modularidad, jerarquía, control de tipos, concurrencia, persistencia y genericidad. El segundo criterio se traduce en cinco características principales: persistencia, concurrencia, recuperación ante fallos del sistema, gestión del almacenamiento secundario y facilidad de consultas:

Persistencia:

Es la capacidad que tiene el programador para que sus datos se conserven al finalizar la ejecución de un proceso, de forma que se puedan reutilizar en otros procesos.

Concurrencia
Se relaciona con la existencia de muchos usuarios interactuando concurrentemente en el sistema. Este debe controlar la interacción entre las transacciones concurrentes para evitar que se destruya la consistencia de la base de datos.

Recuperación

Proporcionar como mínimo el mismo nivel de recuperación que los sistemas de bases de datos actuales. De forma que, tanto en caso de fallo de hardware como de fallo de software, el sistema pueda retroceder hasta un estado coherente de los datos

Gestión del almacenamiento secundario

Es soportada por un conjunto de mecanismos que no son visibles al usuario, tales como gestión de índices, agrupación de datos, selección del camino de acceso, optimización de consultas, etc. Estos mecanismos evitan que los programadores tengan que escribir programas para mantener índices, asignar el almacenamiento en disco, o trasladar los datos entre el disco y la memoria principal, creándose de esta forma una independencia entre los niveles lógicos y físicos del sistema.

Facilidad de Consultas
Permitir al usuario hacer cuestiones sencillas a la base de datos. Este tipo de consultas tienen como misión proporcionar la información solicitada por el usuario de una forma correcta y rápida.

Ventajas en BDOOs

Está su flexibilidad, y soporte para el manejo de tipos de datos complejos. Por ejemplo, en una base de datos convencional, si una empresa adquiere varios clientes por referencia de clientes servicio, pero la base de datos existente, que mantiene la información de clientes y sus compras, no tiene un campo

para registrar quién proporcionó la referencia, de qué manera fue dicho contacto, o si debe compensarse con una comisión, sería necesario reestructurar la base de datos para añadir este tipo de modificaciones. Por el contrario, en una BDOO, el usuario puede añadir una "subclase" de la clase de clientes para manejar las modificaciones que representan los clientes por referencia.

La subclase heredará todos los atributos, características de la definición original, además se especializará en especificar los nuevos campos que se requieren así como los métodos para manipular solamente estos campos. Naturalmente se generan los espacios para almacenar la información adicional de los nuevos campos. Esto presenta la ventaja adicional que una BDOO puede ajustarse a usar siempre el espacio de los campos que son necesarios, eliminando espacio desperdiciado en registros con campos que nunca usan.
La segunda ventaja de una BDOO, es que manipula datos complejos en forma rápida y ágilmente. La estructura de la base de datos está dada por referencias (o apuntadores lógicos) entre objetos.

Desventajas
Al considerar la adopción de la tecnología orientada a objetos, la inmadurez del mercado de BDOO constituye una posible fuente de problemas por lo que debe analizarse con detalle la presencia en el mercado del proveedor para adoptar su producto en una línea de producción sustantiva. Por eso, en este artículo se propone que se explore esta tecnología en un proyecto piloto.

El segundo problema es la falta de estándares en la industria orientada a objetos. Sin embargo, el "Grupo Manejador de Objetos" (OMG), es una organización Internacional de proveedores de sistemas de información y usuarios dedicada a promover estándares para el desarrollo de aplicaciones y sistemas orientados a objetos en ambientes de cómputo en red. La implantación de una nueva tecnología requiere que los usuarios iniciales acepten cierto riesgo. Aquellos que esperan resultados a corto plazo y con un costo reducido quedarán desilusionados. Sin embargo, para aquellos usuarios que planean a un futuro intermedio con una visión tecnológica avanzada, el uso de tecnología avanzada, el uso de tecnología orientada a objetos, paulatinamente compensará todos los riesgos.

Como dijimos anteriormente, la mayor limitación de las bases de datos orientadas a objetos es la carencia de un estándar. ODMG-93 (Object-Oriented Database Management Group) es un punto de partida muy importante para ello .Adopta una arquitectura que consta de un sistema de gestión que soporta un lenguaje de bases de datos orientado a objetos, con una sintaxis similar a un lenguaje de programación también orientado a objetos como puede ser C++ o Smalltalk. El lenguaje de bases de datos es especificado mediante un lenguaje de definición de datos (ODL), un lenguaje de manipulación de datos (OML), y un lenguaje de consulta (OQL), siendo todos ellos portables a otros sistemas con el fin de conseguir la portabilidad de la aplicación completa.

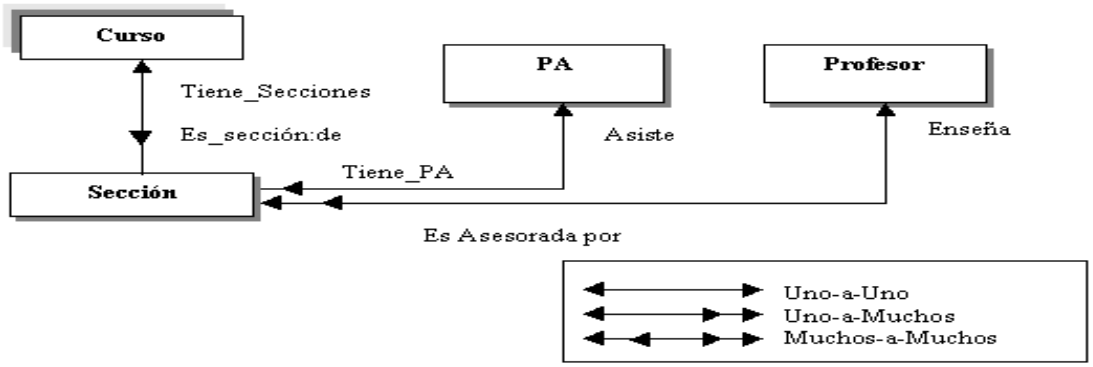
En definitiva, ODMG-93 intenta definir un SGBDOO que integre las capacidades de las bases de datos con las capacidades de los lenguajes de programación, de forma que los objetos de la base de datos aparezcan como objetos del lenguaje de programación, intentando de esta manera eliminar la falta de correspondencia existente entre los sistemas de tipos de ambos lenguajes .El SGBDOO extiende el lenguaje con persistencia, concurrencia, recuperación de datos, consultas asociativas, etc.

El lenguaje de definición de datos (ODL) en un SGBDOO es empleado facilitar la portabilidad de los esquemas de las bases de datos. Este ODL no es un lenguaje de programación completo, define las propiedades y los prototipos de las operaciones de los tipos, pero no los métodos que implementan esas operaciones.

El ODL intenta definir tipos que puedan implementarse en diversos lenguajes de programación; no está por tanto ligado a la sintaxis concreta de un lenguaje de programación particular. De esta forma un esquema especificado en ODL puede ser soportado por cualquier SGBDOO que sea compatible con ODMG-93.

La sintaxis de ODL es una extensión de la del IDL (Interface Definition Language) desarrollado por OMG como parte de CORBA (Common Object Request Broker Architecture).

Representación gráfica de un esquema de una base de datos.



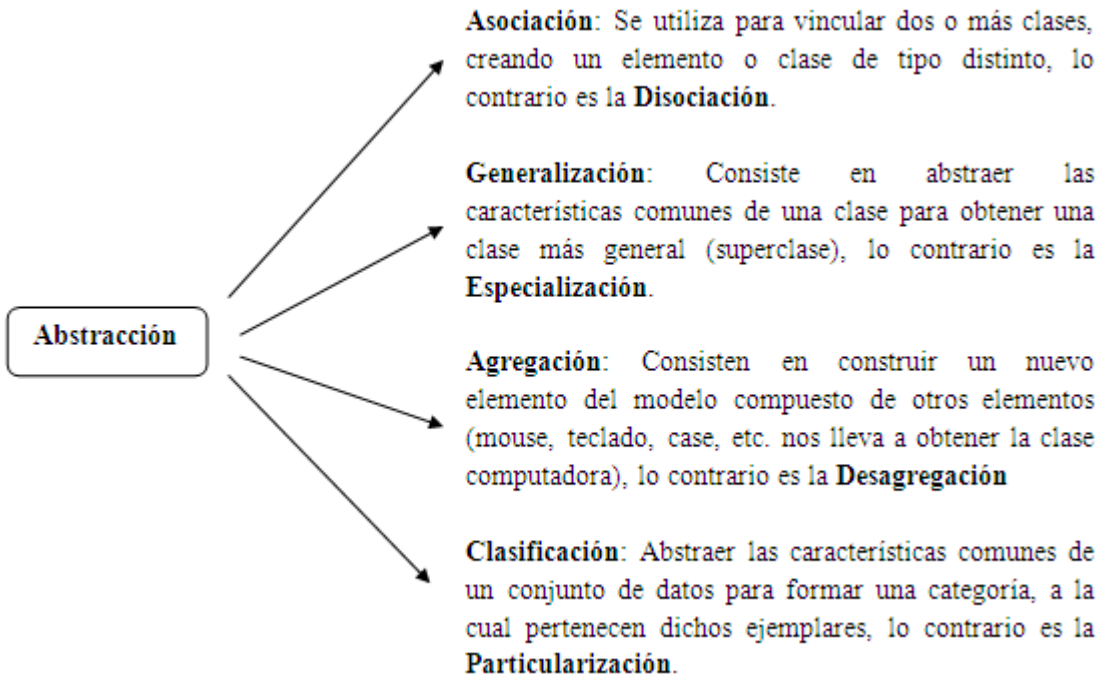
Modelado de los datos.

Los modelos de datos son una herramienta de abstracción que permiten representar la realidad captando su semántica.

El proceso de abstracción nos ayuda a modelar el mundo real, al hacer que nos centremos en lo verdaderamente importa, en el diseño de base de datos se utilizan cuatro tipos de abstracciones los cuales son: Asociación, generalización, agregación y clasificación. Se pueden aplicar solas o combinadas.

La abstracción nos ayuda a concentrarnos en lo que verdaderamente importa.

Los tipos de abstracción y su respectivo contravalor se definen a continuación en el siguiente diagrama:



Por ejemplo: Una base de datos diseñada para almacenar la geometría de ciertas partes mecánicas incluiría clases como CILINDRO, ESFERA Y CUBO. El comportamiento de CILINDRO podría incluir información relativa a sus dimensiones, volumen área superficial:

```
Clase de CILINDRO{
ALTURA FLOTANTE ();
RADIO FLOTANTE ();
VOLUMEN FLOTANTE ();
AREA DE SUPERFICIE FLOTANTE ();
};
```

Se puede llegar a definiciones similares para el cubo y la esfera. En la definición anterior, ALTURA, RADIO y ÁREA representan los mensajes que se pueden enviar a un objeto CILINDRO .

La Implantación se lleva a cabo en el mismo lenguaje, escribiendo funciones correspondientes a las solicitudes OO:

```
CILINDRO::ALTURA () {RETORNA CILINDRO_ALTURA;}
```

```
CILINDRO::VOLUMEN () {RETORNA PI*RADIO ()*ALTURA ();}
```

En este caso, la Altura se almacena como un elemento de los datos, mientras que volumen se calcula

mediante la fórmula apropiada. Observe que la implantación interna de volumen utiliza solicitudes para obtener altura y radio. Sin embargo, el aspecto más importante es la sencillez y uniformidad que experimentan los usuarios de CILINDRO. Sólo necesitan conocer la forma de enviar una solicitud y las solicitudes disponibles.

Persistencia de datos.

Se llama persistencia a la capacidad de trascender el tiempo o el espacio. Es la acción de preservar información de un objeto de forma permanente, permitiendo la recuperación de la misma para que pueda ser empleada y actualizada. En la programación previa al paradigma de objetos estaba representada con la entrada y salida de datos.

Es un concepto importante, pues permite que un objeto pueda ser usado en diferentes momentos a lo largo del tiempo, por el mismo programa o por otros, así como en diferentes instalaciones de hardware en el mismo momento. Esto es particularmente importante en la actualidad, dado que se ha vuelto cotidiano el compartir archivos entre distintas plataformas, invocar procedimientos remotos o usar objetos distribuidos.

Los datos generalmente se almacenan con algún tipo de organización, por ejemplo, un registro, que agrupa datos de diferentes tipos (enteros, string, etc).

En los lenguajes de programación OO, como Java, C++, etc, los datos se organizan en un tipo de datos definido por el usuario (UDT) que recibe el nombre de Clase. Esto implica que las BDOOs persisten tanto el estado de los objetos como su comportamiento. Es decir, los valores de sus atributos y la funcionalidad que tenga a cargo el objeto.

Los objetos pueden ser, según su persistencia:

- Transitorios o efímeros: el tiempo de vida de un objeto depende directamente del entorno del proceso que los instanció, tiempo en memoria.
- Persistentes: el tiempo de vida de un objeto es independiente del proceso que lo instanció, es decir, transciende el tiempo en memoria y es almacenado conservando su estado en un medio secundario. El mismo puede ser recuperado para su utilización o actualización posterior, pudiendo ser reconstruido por el mismo u otro proceso

Bases de datos objetos relacional (BDOR). Aplicaciones prácticas.

Las **Base de Datos Objeto Relacional** son extensión de las base de datos relacional tradicional, a la cual se le proporcionan características de la programación orientada a objetos.

Nacen como una extensión del modelo relacional, en el que los dominios de dicha base de datos ya no son sólo atómicos, por lo que no se cumple la 1FN, debido a que las tuplas también pueden ser una relación, que llevará a la creación de una relación de relaciones. De este modo, se genera la posibilidad de guardar objetos más complejos en una sola tabla con referencias a otras relaciones, con lo que se acerca más al paradigma de programación orientada a objetos

La relación entre bases de datos y objetos es interesante por la importancia de las bases de datos en las empresas y el avance de la tecnología de orientación a objetos. Por eso, no puede haber auténtica POO si no se puede garantizar la persistencia en bases de datos.

A lo largo del tiempo, ha habido diferentes niveles de orientación a objetos en las bases de datos.

En un primer momento, lo único que se hizo fue incorporar BLOBs (Binary Large Object, largas cadenas de bytes sin formato alguno) en las bases de datos relacionales, de modo de poder representar cualquier tipo de información como contenido, así como partes variantes de los objetos. El inconveniente principal es que los BLOBs no sirven para realizar búsquedas o indexaciones.

El siguiente paso fue la incorporación de "front-ends" orientados a objetos a las bases de datos relacionales. Esta tecnología, que es aún la que más se utiliza, se la llama también de bases de datos relacionales basadas en objetos (BDOR). En este caso, hay una capa filtrante que traduce entre objetos y el modelo relacional tradicional. Por lo tanto, los objetos deben ser interceptados antes de ser almacenados y traducidos de formato, y al recuperarlos se hace el procedimiento inverso. Entre sus ventajas está la simplicidad de implementación, que aprovecha el éxito del modelo relacional y de SQL y la facilidad de adaptación de recursos humanos y de datos. Sin embargo, presenta tiempos de respuesta elevados y no hay una genuina orientación a objetos, no habiendo herencia, polimorfismo, ni nada que se le parezca. Además, el software de traducción está disociado del universo del sistema: es una solución de problemas tecnológicos. En definitiva, esta solución deja al descubierto la complejidad de representar un objeto complejo en el modelo relacional.

En el software orientado a objetos, la información se representa como clases y objetos. En las bases de datos relacionales, como tablas y sus restricciones. Por tanto, para almacenar la información tratada en un programa orientado a objetos en una base de datos relacional es necesaria una traducción entre ambas formas.

Una aproximación ampliamente usada es el mapeo objeto-relacional (ORM). Tanto en la forma de Table gateway como de Table-row gateway, se hace una correspondencia entre los siguientes elementos:

Programa OO	BD relacional
Clase	Tabla
Propiedad	Campo
Objeto	Fila
Identificador	Clave primaria
Puntero a otro objeto	Clave foránea

Esta correspondencia es muy "natural" ya que, en realidad el modelo orientado a objetos y el modelo relacional no son tan diferentes. Veámoslo: en los programas las instancias de un objeto son accedidas a través de un puntero a su posición de memoria. Por ejemplo:

```
$a = new A;

$b = $a;
```

En la ejecución del código anterior se realizan las siguientes acciones:

Se instancia la clase A, es decir, se almacena en memoria el registro correspondiente al TDA representado en la clase A (Línea 1)

Se crea una variable tipo puntero llamada $a que apunta al registro anterior (Línea 1)

Se crea otra variable ($b) de tipo puntero que toma el mismo valor que $a, es decir, al registro creado en #1 (Línea 2)

Conceptualmente, podemos decir que el registro en memoria es propiamente el estado del objeto y su posición, indicada en $a, un identificador único del mismo. Y recordemos que los objetos anidados (es decir, las propiedades de $a que sean, a su vez, otros objetos) se almacenarán en el registro como punteros a otros registros.

Esta forma de almacenar información a bajo nivel —manejada comúnmente en el lenguaje C— se asemeja al modelo de las tablas y las relaciones de las bases de datos relacionales: cada registro es una fila, cada puntero una referencia foreign key. Las posiciones en memoria, es decir, los "identificadores únicos" de los objetos podríamos tratarlos como primary keys.

Tema IV: Desarrollo de la Aplicación.

Herramientas de Desarrollo Integrado (IDE). Funciones y Componentes principales. Especificación del proceso de desarrollo. Interpretación de los casos de uso. Selección de las herramientas de desarrollo. Codificación. Compilación y Pruebas. Aseguramiento de la calidad del software: eficiencia, flexibilidad, corrección, confiabilidad, mantenibilidad, portabilidad, usabilidad, seguridad e integridad. Desarrollo de una aplicación orientada a objetos y desarrollo de una aplicación estructurada.

Herramientas de Desarrollo Integrado (IDE):

Un entorno de desarrollo integrado, llamado también IDE (sigla en inglés de integrated development environment), es un programa informático compuesto por un conjunto de herramientas de programación. Puede dedicarse en exclusiva a un solo lenguaje de programación o bien poder utilizarse para varios.

Un IDE es un entorno de programación que ha sido empaquetado como un programa de aplicación, es decir, consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI). Los IDEs pueden ser aplicaciones por sí solas o pueden ser parte de aplicaciones existentes. El lenguaje Visual Basic, por ejemplo, puede ser usado dentro de las aplicaciones de Microsoft Office, lo que hace posible escribir sentencias Visual Basic en forma de macros para Microsoft Word.

Funciones y Componentes principales.

Los IDE proveen un marco de trabajo amigable para la mayoría de los lenguajes de programación tales como C++, PHP, Python, Java, C#, Delphi, Visual Basic, etc. En algunos lenguajes, un IDE puede funcionar como un sistema en tiempo de ejecución, en donde se permite utilizar el lenguaje de programación en forma interactiva, sin necesidad de trabajo orientado a archivos de texto, como es el caso de Smalltalk u Objective-C.

Algunos entornos son compatibles con múltiples lenguajes de programación, como Eclipse o NetBeans, ambos basados en Java; o MonoDevelop, basado en C#. También puede incorporarse la funcionalidad para lenguajes alternativos mediante el uso de plugins. Por ejemplo, Eclipse y NetBeans tienen plugins para C, C++, Ada, Perl, Python, Ruby y PHP, entre otros.

Editor de texto:

Es la parte que nos permite escribir el código del programa, lo que se llama comúnmente el código fuente, compuesto por caracteres alfanuméricos y caracteres especiales como *, +, -, /, {,}, (,)! ... Ofrece funciones para el usuario tales como cortar, pegar, buscar...

Además es capaz de reconocer resaltar y cambiar los colores de las variables, las cadenas de caracteres el inicio y fin de los corchetes.... Todo esto Para que el código fuente sea mucho más visual, cómodo y podamos reconocer errores a simple vista.

Compilador:

Es el traductor; es el encargado de recoger el código fuente que el usuario es capaz de interpretar y lo traduce generando un programa equivalente que la máquina será capaz de interpretar. Usualmente el segundo lenguaje es lenguaje de máquina. El proceso de traducción se conoce como compilación.

Intérprete o interpretador:

Es el programa que se encarga de analizar y ejecutar otros programas, escritos en un lenguaje de alto nivel. Los intérpretes se diferencian de los compiladores en que sólo realizan la traducción a medida que sea necesaria, típicamente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción estos suelen ser más lentos que los compilados debido a la necesidad de traducir el programa mientras se ejecuta, pero a cambio son más flexibles como entornos de programación y depuración y permiten ofrecer al programa interpretado un entorno no dependiente de la máquina donde se ejecuta el intérprete, sino del propio intérprete (lo que se conoce comúnmente como máquina virtual), algunas implementaciones de programación de lenguajes de programación pueden interpretar o compilar el código fuente original en una más compacta forma intermedia y después traducir eso al código de máquina (ej. Perl, Python, MATLAB, y Ruby). Algunos aceptan los archivos fuente guardados en esta representación intermedia (ej. Python, UCSD Pascal y Java).

Depurador (Debugger):

Es un programa que permite depurar o limpiar los errores en el código fuente de otro programa informático. La depuración, el depurador lanza el programa a depurar. Éste se ejecuta normalmente hasta que el depurador detiene su ejecución, permitiendo al usuario examinar la situación. Se depuraran los errores de los programas. El depurador permite detener el programa en:

1. Un punto determinado mediante un punto de ruptura.

2. Un punto determinado bajo ciertas condiciones mediante un punto de ruptura condicional.

3. Un momento determinado cuando se cumplan ciertas condiciones.

4. Un momento determinado a petición del usuario.

5. Durante esa interrupción, el usuario puede:

6. Examinar y modificar la memoria y las variables del programa.

7. Examinar el contenido de los registros del procesador.

8. Examinar la pila de llamadas que han desembocado en la situación actual.

9. Cambiar el punto de ejecución, de manera que el programa continúe su ejecución en un punto diferente al punto en el que fue detenido.

10. Ejecutar instrucción a instrucción.

11. Ejecutar partes determinadas del código, como el interior de una función, o el resto de código antes de salir de una función.

12. El depurador depende de la arquitectura y sistema en el que se ejecute, por lo que sus funcionalidades cambian de un sistema a otro. Aquí se han mostrado las más comunes.

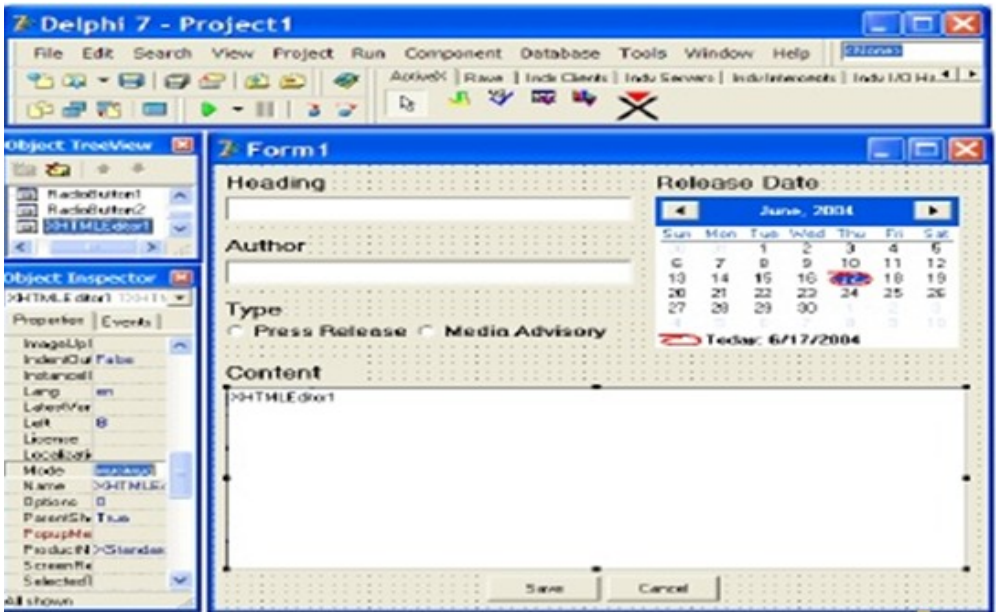
Constructor de interfaz gráfica:

Es una herramienta de programación que simplifica la creación de interfaces gráficas de usuario permitiendo que el diseñador para organizar los widgets utilizando un editor WYSIWYG de arrastrar y soltar. Sin un constructor de interfaz gráfica de usuario, una interfaz gráfica de usuario debe ser construido por especificar manualmente los parámetros de cada widget en el código, sin retroalimentación visual hasta que el programa se ejecuta. Las interfaces de usuario se suelen programar utilizando una arquitectura orientada a eventos, por lo que los constructores GUI también simplifican la creación de código orientado a eventos. Este código se conecta a los reproductores de apoyo a los acontecimientos salientes y entrantes que activan las funciones de proporcionar la lógica de la aplicación.

Ejemplos de entornos

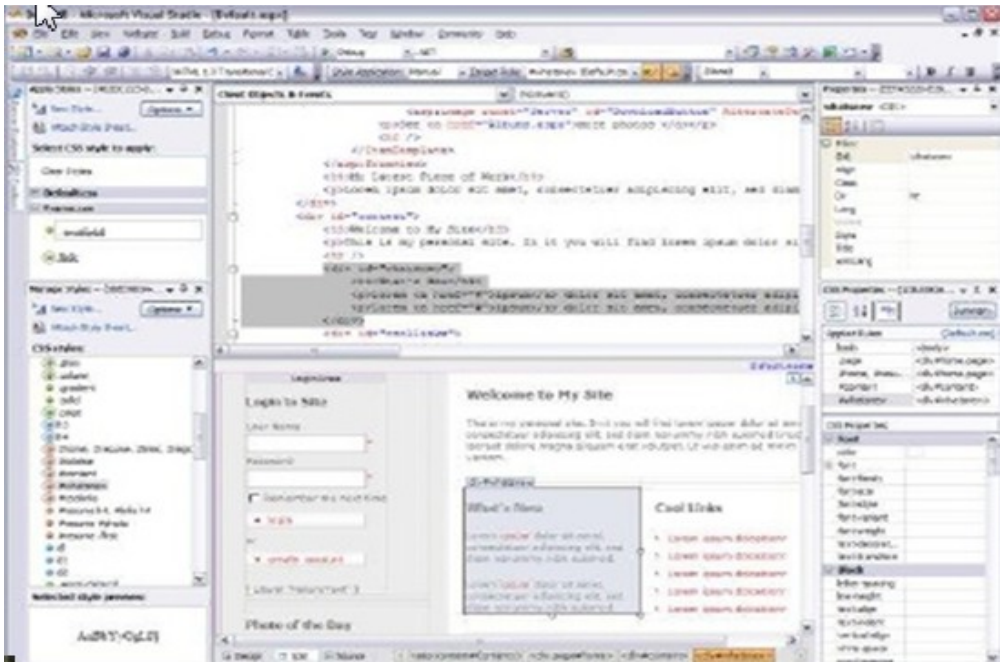
DELPHI

Delphi es un entorno de desarrollo de software diseñado para la programación de propósito general con énfasis en la programación visual. En Delphi se utiliza como lenguaje de programación una versión moderna de Pascal llamada Object Pascal. Es producido comercialmente por la empresa estadounidense CodeGear, adquirida en mayo de 2008 por Embarcadero Technologies, una empresa del grupo Thoma Cressey Bravo, en una suma que ronda los 30 millones de dólares. En sus diferentes variantes, permite producir archivos ejecutables para Windows, GNU/Linux y la plataforma .NET. El Entorno Integrado de Desarrollo (EID) o IDE es el ambiente de desarrollo de programas de Delphi. Se trata de un editor de formularios (que permite el desarrollo visual), un potente editor de textos que resalta la sintaxis del código fuente, la paleta de componentes y el depurador integrado, además de una barra de botones y un menú que nos permite la configuración de la herramienta y la gestión de proyectos. En las ediciones Client/Server y Enterprise el EID también ofrece integración con una herramienta de control de versiones.



MICROSOFT VISUAL STUDIO

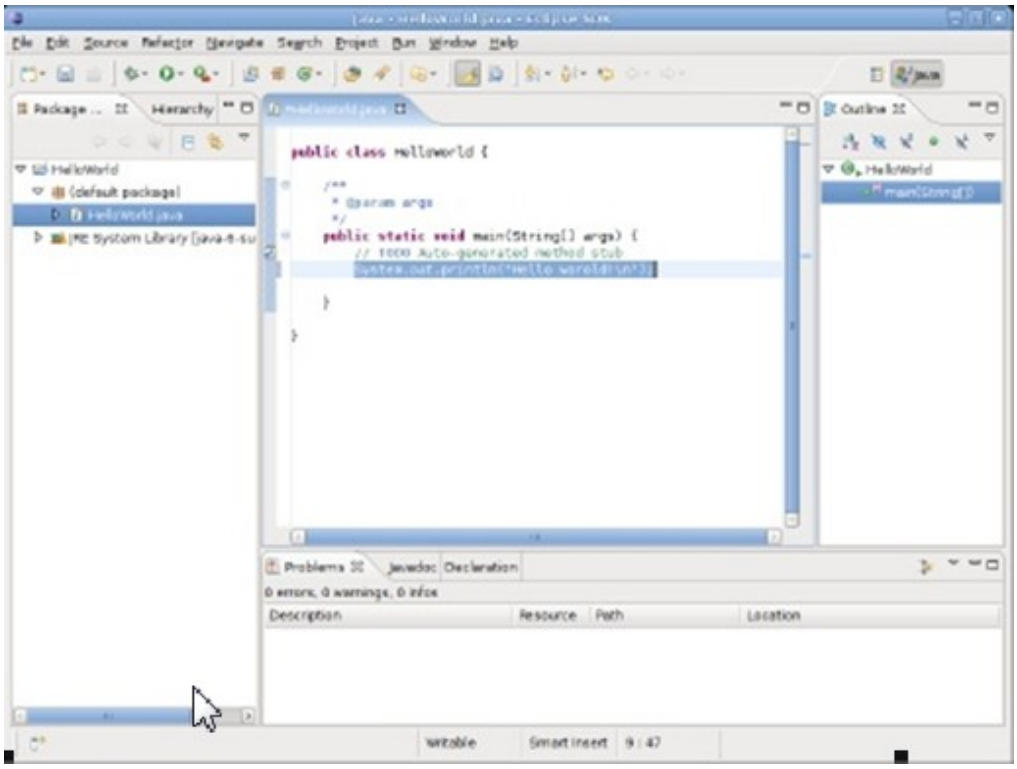
Microsoft Visual Studio es un entorno de desarrollo integrado para sistemas operativos Windows. Soporta varios lenguajes de programación tales como Visual C++, Visual C#, Visual J#, ASP.NET y Visual Basic .NET, aunque actualmente se han desarrollado las extensiones necesarias para muchos otros. Visual Studio permite a los desarrolladores crear aplicaciones, sitios y aplicaciones web, así como servicios web en cualquier entorno que soporte la plataforma .NET (a partir de la versión net 2002). Así se pueden crear aplicaciones que se intercomuniquen entre estaciones de trabajo, páginas web y dispositivos móviles.



ECLIPSE

Eclipse es un entorno de desarrollo integrado multiplataforma, desarrollado inicialmente por IBM y liberado por Eclipse Foundation en 2006, que se caracteriza por la posibilidad de extender la funcionalidad del entorno mediante la adición de plug-ins y el empleo de la interfaz gráfica de usuario JFace, que simplifica la construcción de aplicaciones basadas en SWT. Otras características de Eclipse son:

- Editor de texto con resaltado de sintaxis.
- Compilación en tiempo real.
- Posibilidad de emplear asistentes para la creación de proyectos, de clases
- Control de versiones con CVS. Integración con Ant.



Especificación del proceso de desarrollo.

Un proceso para el desarrollo de software, también denominado ciclo de vida del desarrollo de software es una estructura aplicada al desarrollo de un producto de software. Hay varios modelos a seguir para el establecimiento de un proceso para el desarrollo de software, cada uno de los cuales describe un enfoque diferente para diferentes actividades que tienen lugar durante el proceso.

Algunos autores consideran un modelo de ciclo de vida un término más general que un determinado proceso para el desarrollo de software. Por ejemplo, hay varios procesos de desarrollo de software específicos que se ajustan a un modelo de ciclo de vida de espiral.

La gran cantidad de organizaciones de desarrollo de software implementan metodologías para el proceso de desarrollo. Muchas de estas organizaciones pertenecen a la industria armamentística, que en los Estados Unidos necesita un certificado basado en su modelo de procesos para poder obtener un contrato.

Los más conocidos son:

• ***Modelo secuencial***. Representado por metodologías tan famosas como Waterfall. Se inicia con un completo análisis de los requisitos de los usuarios. En el siguiente paso, los programadores implementan el diseño y finalmente, el completado y perfecto sistema es probado y enviado.

• ***Desarrollo incremental.*** Su principal objetivo es reducir el tiempo de desarrollo, dividiendo el proyecto en intervalos incrementales superpuestos. Del mismo modo que con el modelo waterfall, todos los requisitos se analizan antes de empezar a desarrollar, sin embargo, los requisitos se dividen en "incrementos" independientemente funcionales.

• ***Desarrollo iterativo*.** A diferencia del modelo incremental se centra más en capturar mejor los requisitos cambiantes y la gestión de los riesgos. En el desarrollo iterativo se rompe el proyecto en iteraciones de diferente longitud, cada una de ellas produciendo un producto completo y entregable.

• ***Modelo en espiral.*** Comprende las mejores características de ciclo de vida clásico y el prototipado (desarrollo iterativo). Además, incluye el análisis de alternativas, identificación y reducción de riesgos.

Durante décadas se ha perseguido la meta de encontrar procesos reproducibles y predecibles que mejoren la productividad y la calidad. Algunas de estas soluciones intentan sistematizar o formalizar la aparentemente desorganizada tarea de desarrollar software. Otros aplican técnicas de gestión de proyectos para la creación del software. Sin una gestión del proyecto, los proyectos de software corren el riesgo de demorarse o consumir un presupuesto mayor que el planeado. Dada la cantidad de proyectos de software que no cumplen sus metas en términos de funcionalidad, costos o tiempo de entrega, una gestión de proyectos efectiva es algo que a menudo falta. Por ello en todo desarrollo de software se deben tener en cuenta como mínimo las siguientes actividades:

Actividades principales en el desarrollo de software:

Planificación:

La importante tarea a la hora de crear un producto de software es obtener los requisitos o el análisis de los requisitos. Los clientes suelen tener una idea más bien abstracta del resultado final, pero no sobre las funciones que debería cumplir el software.

Una vez que se hayan recopilado los requisitos del cliente, se debe realizar un análisis del ámbito del desarrollo. Este documento se conoce como especificación funcional.

Desarrollo, pruebas y documentación:

El desarrollo es parte del proceso en el que los ingenieros de software programan el código para el proyecto.

Las pruebas de software son parte esencial del proceso de desarrollo del software. Esta parte del proceso tiene la función de detectar los errores de software lo antes posible.

La documentación del diseño interno del software con el objetivo de facilitar su mejora y su mantenimiento se realiza a lo largo del proyecto. Esto puede incluir la documentación de un API, tanto interior como exterior.

Implementación y mantenimiento:

La implementación comienza cuando el código ha sido suficientemente probado, ha sido aprobado para su liberación y ha sido distribuido en el entorno de producción.

Entrenamiento y soporte para el software es de suma importancia y algo que muchos desarrolladores de software descuidan. Los usuarios, por naturaleza, se oponen al cambio porque conlleva una cierta inseguridad, es por ello que es fundamental instruir de forma adecuada a los futuros usuarios del software.

El mantenimiento y mejora del software de un software con problemas recientemente implementado puede requerir más tiempo que el desarrollo inicial del software. Es posible que haya que incorporar código que no se ajusta al diseño original con el objetivo de solucionar un problema o ampliar la funcionalidad para un cliente. Si los costos de mantenimiento son muy elevados puede que sea oportuno rediseñar el sistema para poder contener los costos de mantenimiento.

Interpretación de los casos de uso.

Un caso de uso es una descripción de los pasos o las actividades que deberán realizarse para llevar a cabo algún proceso.

Los personajes o entidades que participarán en un caso de uso se denominan actores.

En el contexto de ingeniería del software, un caso de uso es una secuencia de interacciones que se desarrollarán entre un sistema y sus actores en respuesta a un evento que inicia un actor principal sobre el propio sistema.

Los diagramas de casos de uso sirven para especificar la comunicación y el comportamiento de un sistema mediante su interacción con los usuarios y/u otros sistemas. O lo que es igual, un diagrama que muestra la relación entre los actores y los casos de uso en un sistema.

Una relación es una conexión entre los elementos del modelo, por ejemplo la especialización y la generalización son relaciones. Los diagramas de casos de uso se utilizan para ilustrar los requerimientos del sistema al mostrar cómo reacciona a eventos que se producen en su ámbito o en él mismo.

Los casos de uso evitan típicamente la jerga técnica, prefiriendo la lengua del usuario final o del experto del campo del saber al que se va a aplicar. Los casos del uso son a menudo elaborados en colaboración por los analistas de requerimientos y los clientes.

Cada caso de uso se centra en describir cómo alcanzar una única meta o tarea de negocio.

El grado de la formalidad de un proyecto particular del software y de la etapa del proyecto influenciará el nivel del detalle requerido en cada caso de uso.

Los casos de uso pretenden ser herramientas simples para describir el comportamiento del software o de los sistemas. Un caso de uso contiene una descripción textual de todas las maneras que los actores previstos podrían trabajar con el software o el sistema.

Los casos de uso no describen ninguna funcionalidad interna (oculta al exterior) del sistema, ni explican cómo se implementará. Simplemente muestran los pasos que el actor sigue para realizar una operación.

Un caso de uso debe:

- describir una tarea del negocio que sirva a una meta de negocio
- tener un nivel apropiado del detalle
- ser bastante sencillo como que un desarrollador lo elabore en un único lanzamiento

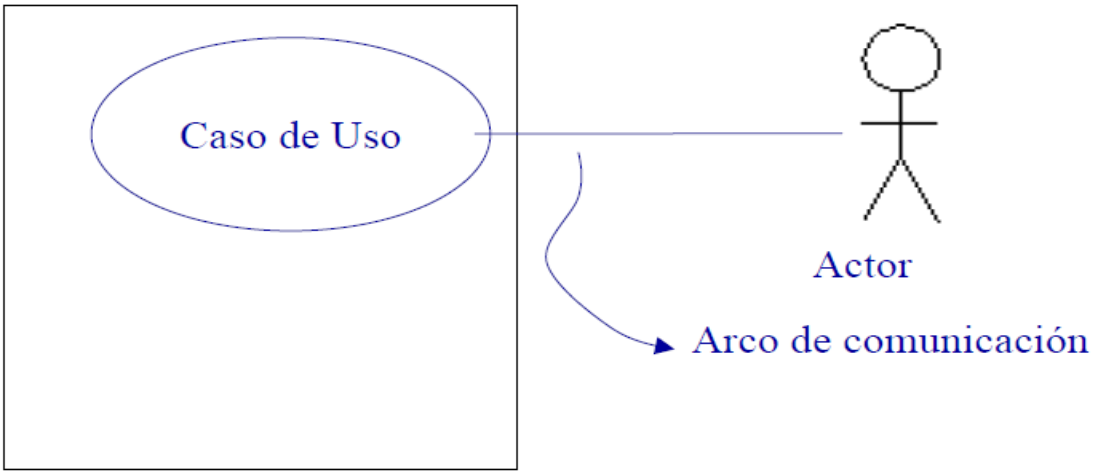
Situaciones que pueden darse:

- Un actor se comunica con un caso de uso (si se trata de un actor primario la comunicación la iniciará el actor, en cambio si es secundario, el sistema será el que inicie la comunicación).
- Un caso de uso extiende otro caso de uso.
- Un caso de uso utiliza otro caso de uso.

Notación de los casos de uso en UML

• Los casos de uso se representan por una elipse conteniendo el nombre, que opcionalmente podría ir debajo de la elipse.

• Los actores se representan con un monigote y el nombre del actor al pie de la figura. Los nombres de los actores suelen empezar por mayúscula.



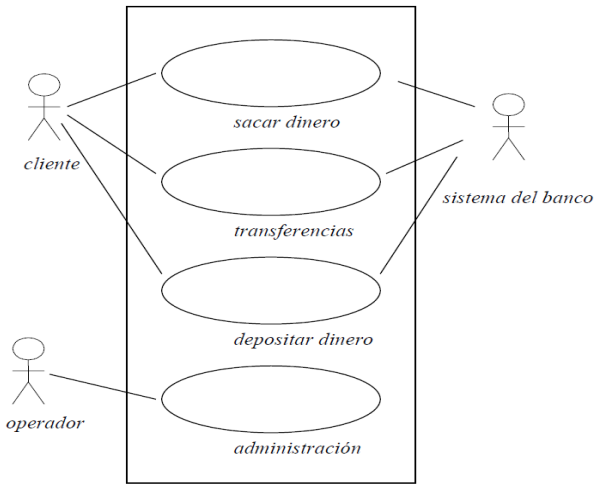
Desde una perspectiva tradicional de la ingeniería de software, un caso de uso describe una característica del sistema. Para la mayoría de proyectos de software, esto significa que quizás a veces es necesario especificar diez o centenares de casos de uso para definir completamente el nuevo sistema.

Descripción de los Casos de uso

<Identificador>	<nombre descriptivo>	
Descripción	El sistema deberá permitir a [lista actores] en [instante en el que se puede realizar el caso de uso] [funcionalidad que define el caso de uso] según se describe en el siguiente caso de uso:	
Secuencia Normal	**Paso**	**Acción**
	1	{<acción a realizar>, realizar el caso de uso [caso de uso]}
	2	<Situación que produce una alternativa>
	2a	Si [Situación que produce una alternativa] el sistema deberá {<acción a realizar>, realizar el caso de uso [caso de uso]}
	2b	Si [Situación que produce una alternativa] el sistema deberá {<acción a realizar>, realizar el caso de uso [caso de uso]}
	…	….
	…	…
	n	….
Excepciones	**Paso**	**Acción**
	p	En el caso de que [situación que provoca la excepción] el sistema deberá {<acción a realizar>, realizar el caso de uso [caso de uso]}
	…	…
	q	…
Rendimiento	El sistema deberá realizar la/s acción /es descrita/s en {los pasos [primer paso] al [último paso], el paso [número de paso]} en un máximo de [cota de tiempo]	
Frecuencia	Este caso de uso se espera que se lleve a cabo una media de [número de veces] al [unidad temporal]	
Importancia	{vital, importante,quedaría bien}	
Urgencia	{inmediatamente, hay presión, puede esperar}	
Comentarios	<otras consideraciones en formato libre>	

Ejemplo de casos de uso:

Figura 1:



Descripción del caso "Sacar Dinero":

CU-003	Sacar dinero
Descripción	El sistema deberá permitir al cliente del banco, en cualquier momento, sacar dinero según se describe en el siguiente caso de uso:
Secuencia Normal	1+ El usuario inserta la tarjeta en el cajero 2 + El cajero lee el código de la banda magnética de la tarjeta y verifica si es aceptable y pide el código del usuario
	3+ El usuario introduce el código 4 + Si el código es correcto, el cajero pide al usuario que seleccione el tipo de transacción deseada
	5+ El usuario selecciona la función sacar dinero, 6 + El cajero le pide al usuario que teclee la cantidad deseada
	7 + El usuario teclea la cantidad que quiere sacar, 8 + El cajero envía la petición al sistema del banco
	9 a Si conecta el sistema deberá comprobar si hay dinero en la cuenta
	9 b Si no conecta el sistema deberá comprobar si el dinero es menos que el límite
	10 En cualquiera de los dos casos el sistema: + expulsa la tarjeta + imprime el recibo + entrega el dinero
Excepciones	2' La tarjeta no es aceptada + Se expulsa emitiendo un sonido 4' Código incorrecto (1,2) + Se emite un mensaje dando al usuario la oportunidad de volver a introducir el código (paso 3) 4" Código incorrecto (3) + Se emite un mensaje y se retiene la tarjeta 9' No autorizado para sacar dinero + El sistema de banco no autoriza a sacar dinero. Se emite un mensaje de información y se expulsa la tarjeta 9 a ', 9 b' No hay dinero suficiente + El cajero no dispone de la cantidad pedida. Emite un mensaje y vuelve al paso 7 1..10' Cancelar + En cualquier momento el usuario puede cancelar la transacción, con lo que se expulsa la tarjeta

El caso de uso "sacar dinero", puede describirse como sigue:
o Se visualiza un mensaje de bienvenida en la pantalla:
+ El usuario inserta la tarjeta en el cajero
+ El cajero lee el código de la banda magnética de la tarjeta y verifica si es aceptable, en caso positivo, pide el código del usuario
o Esperando el código:
+ El usuario introduce el código
+ Si el código es correcto, el cajero pide al usuario que seleccione el tipo de transacción deseada
o Esperando el tipo de transacción:
+ El usuario selecciona la función sacar dinero, el cajero le pide al usuario que teclee la cantidad deseada
o Esperando la cantidad deseada:
+ El usuario teclea la cantidad que quiere sacar, el cajero envía la petición al sistema del banco
+ Se prepara un recibo
+ Se expulsa la tarjeta
+ Se imprime el recibo
+ Se entrega el dinero

EXPCEPCIONES
o La tarjeta no es aceptada
+ Se expulsa emitiendo un sonido
o Código incorrecto
+ Se emite un mensaje dando al usuario la oportunidad de volver a introducir el código
o No autorizado para sacar dinero
+ El sistema de banco no autoriza a sacar dinero. Se emite un mensaje de información y se expulsa la tarjeta
o No hay dinero
+ El cajero no dispone de la cantidad pedida. Emite un mensaje y expulsa la tarjeta

Selección de las herramientas de desarrollo. Codificación. Compilación y Pruebas.

Qué son las herramientas

Una herramienta es según la definición de la Real Academia de la lengua Española (RAE) , un instrumento del cual nos servimos para hacer algo. Por tanto, si aplicamos esta definición a las herramientas que utilizamos en el desarrollo de un proyecto software, lo que obtenemos es que una herramienta es un instrumento del cual nos servimos para construir software y que es utilizado en cualquiera de las fases de su desarrollo. Estos instrumentos generalmente son aplicaciones software, pero también pueden ser cualquier otro tipo de instrumento que nos ayude en el desarrollo de nuestro producto, como una calculadora o un simple bolígrafo, estas últimas no las tendremos en cuenta.

El concepto que realmente nos interesa en este capítulo es el de herramientas de ingeniería del software, ya que el desarrollo del software debe ser una actividad desarrollada por ingenieros que utilizan técnicas, métodos y herramientas adecuadas para cada solución. Una herramienta de ingeniería del software es un recurso que ayuda a las personas a realizar su trabajo de ingeniería del software. Para el propósito de este capítulo definiremos herramienta como: "Una aplicación software que se utiliza para desarrollar, probar, analizar o mantener otra aplicación software o su

documentación". Esta definición la deberíamos extender de tal manera que la aplicación que denominamos herramienta pueda ser utilizada a lo largo de todo el proceso de desarrollo del producto o aplicación software final y por tanto, incluir tareas como la gestión de los procesos o detección de los requisitos. También consideraremos herramientas los plugins, frameworks, librerías y componentes, a pesar de no ser aplicaciones software en si mismas, pero si que son utilizadas por las mismas.

Necesidad de las herramientas

Las herramientas, como aplicaciones de software que son, han ido evolucionando a lo largo de estos últimos años y adaptándose a las necesidades y tendencias de cada momento. Actualmente nos encontramos en una época dominada por las aplicaciones web y por metodologías de desarrollo que tienden a realizar iteraciones cortas. Dentro de este nuevo entorno, necesitamos herramientas que se adapten a los procesos ágiles que se dan en el ciclo de vida del desarrollo del software, estas, como hemos visto requieren muchas veces de actividades muy concretas y mecanismos sofisticados de integración.

Podemos observar como la popularidad del desarrollo del software de una forma distribuida o colaborativa es cada vez mayor, ya sea a través del outsourcing, comunicación a distancia o equipos de desarrollo virtuales, incrementando la demanda de intercambio de información y trabajo apoyado en herramientas colaborativas.

No solo se han modificado los procesos de desarrollo del software, sino que los productos que desarrollamos ahora no son los mismos que desarrollábamos hace 15 años, tal y como hemos comentado al principio de esta sección, actualmente las aplicaciones web ocupan la mayoría de los recursos de las consultorías informáticas. Pero no es tan solo eso, sino que las aplicaciones actuales cada vez son más grandes, complejas y diversas, utilizan nuevas tecnologías, componentes y métodos, que obviamente necesitan herramientas nuevas y específicas. Los desarrolladores tienen nuevas necesidades, ahora requieren de mejores herramientas para especificar, diseñar, generar, probar y desplegar complejos sistemas distribuidos e integrar diferentes tecnologías de sistemas heredados. Necesitan soporte para el diseño de interfaces de usuario cada vez más exigentes, control de versionado, gestión de la configuración, documentación y reutilización de un gran número de artefactos, …

En definitiva, los procesos y productos generados son más complejos que hace años y es necesaria la utilización de herramientas que nos faciliten la realización de los proyectos mejor, más rápidamente y de un modo más eficiente.

Clasificación de las herramientas

Existen diferentes tipos de clasificaciones o categorías por las cuales podemos identificar las herramientas de desarrollo de nuestros proyectos de software.

Una clasificación muy intuitiva y que esta orientada a clasificar las herramientas de diferentes formas es:

- Licencia. Según el tipo de licencia con la que se distribuye, puede ser comercial, open-source, freeware, shareware.
- Lenguaje soportado. Según el lenguaje de programación que soporte. Por ejemplo Java, .Net, C++, etc.

- Plataforma. Plataforma en la cual se puede ejecutar la aplicación, windows, unix, linux, web, mainframe, etc.

- Gestión de los datos. Se refiere al acceso a las bases de datos. Según el acceso a datos que soporte, Odbc, Ado.Net, Oledb, etc.,

Según esta clasificación, uno de los temas críticos al momento de la selección de una herramienta de software para el desarrollo de una aplicación, es el lenguaje de programación que soporta la misma.

En la actualidad existen muchas herramientas que generan gran parte de este código en forma automática, pero siempre se necesita del conocimiento de especialistas informáticos para la adaptación de ese código a las necesidades propias del proyecto.

Una vez determinado el lenguaje soportado, existen otros criterios que se deberá considerar para la selección de la herramienta:

- ***IDE Amigable e intuitivo:*** se debería buscar una herramienta que tenga un IDE (Entorno de desarrollo integrado) fácil de entender y manejar. Que posea un adecuado editor de código con sintaxis resaltada con integración al compilador y al depurador. Que posea también un sistema de ayuda en línea y ambiente RAD, soporte para control de versión y programación múltiple.

- ***Posibilidad de depuración:*** integrado o en línea de comando, que permita un seguimiento vinculado al código fuente escrito.

- ***Librerías para programación de interfaces gráficas:*** que se puedan utilizar con cualquier compilador, que sea multiplataforma (Windows® y Linux) y sea adecuada para la implementación de interfaces gráficas de usuario (GUI) de programas visuales (ventanas múltiples, menúes, cuadros de diálogo, botones, cajas de selección, cajas de texto y cajas desplegables como mínimo).

- ***Compatibilidad con Normas Internacionales:*** ANSI.

- ***Facilidades Multiplataforma:*** se requiere que el código escrito en una plataforma no esté vinculado a la herramienta ni a la plataforma y cuya portablidad (intercambio a otra herramienta o entre Windows® y Linux) sea directa.

- ***Costo:*** El costo debe ser razonable o gratuito.

Una vez seleccionada la herramienta se procede a la codificación, compilación y pruebas de la aplicación, etapas que ya fueron explicadas en otra sección de este capítulo.

Aseguramiento de la calidad del software: eficiencia, flexibilidad, corrección, confiabilidad, mantenibilidad, portabilidad, usabilidad, seguridad e integridad.

Uno de los problemas que se afrontan actualmente en la esfera de la computación es la calidad del *software*. Desde la década del 70, este tema ha sido motivo de preocupación para especialistas, ingenieros, investigadores y comercializadores de *softwares*, los cuales han realizado gran cantidad de investigaciones al respecto con dos objetivos fundamentales:

1. ¿Cómo obtener un *software* con calidad?
2. ¿Cómo evaluar la calidad del *software*?

Ambas interrogantes conllevan amplias respuestas, pero están estrechamente ligadas con el concepto de la calidad del *software*, que es el resultado de la primera y la fuente de la segunda.

¿QUE ES LA CALIDAD DEL SOFTWARE?

La calidad del *software* es el conjunto de cualidades que lo caracterizan y que determinan su utilidad y existencia. La calidad es sinónimo de eficiencia, flexibilidad, corrección, confiabilidad, mantenibilidad, portabilidad, usabilidad, seguridad e integridad.

La calidad del *software* es medible y varía de un sistema a otro o de un programa a otro. Un *software* elaborado para el control de naves espaciales debe ser confiable al nivel de "cero fallas"; un *software* hecho para ejecutarse una sola vez no requiere el mismo nivel de calidad; mientras que un producto de *software* para ser explotado durante un largo período (10 años o más), necesita ser confiable, mantenible y flexible para disminuir los costos de mantenimiento y perfeccionamiento durante el tiempo de explotación.

La calidad del software puede medirse después de elaborado el producto. Pero esto puede resultar muy costoso si se detectan problemas derivados de imperfecciones en el diseño, por lo que es imprescindible tener en cuenta tanto la obtención de la calidad como su control durante todas las etapas del ciclo de vida del *software*.

¿COMO OBTENER UN SOFTWARE DE CALIDAD?

La obtención de un *software* con calidad implica la utilización de metodologías o procedimientos estándares para el análisis, diseño, programación y prueba del *software* que permitan uniformar la filosofía de trabajo, en aras de lograr una mayor confiabilidad, mantenibilidad y facilidad de prueba, a la vez que eleven la productividad, tanto para la labor de desarrollo como para el control de la calidad del *software*.

La política establecida debe estar sustentada sobre tres principios básicos: tecnológico, administrativo y ergonómico.

El principio tecnológico define las técnicas a utilizar en el proceso de desarrollo del *software.*

El principio administrativo contempla las funciones de planificación y control del desarrollo del *software*, así como la organización del ambiente o centro de ingeniería de *software*.

El principio ergonómico define la interfaz entre el usuario y el ambiente automatizado.

La adopción de una buena política contribuye en gran medida a lograr la calidad del *software*, pero no la asegura. Para el aseguramiento de la calidad es necesario su control o evaluación.

¿COMO CONTROLAR LA CALIDAD DEL SOFTWARE?

Para controlar la calidad del *software* es necesario, ante todo, definir los parámetros, indicadores o criterios de medición, ya que, como bien plantea Tom De Marco, "usted no puede controlar lo que no se puede medir".

Las cualidades para medir la calidad del *software* son definidas por innumerables autores, los cuales las denominan y agrupan de formas diferentes. Por ejemplo, John Wiley define métricas de calidad y criterios, donde cada métrica se obtiene a partir de combinaciones de los diferentes criterios. La ***Metodología para la evaluación de la calidad de los medios de programas*** **de la CIC**, de Rusia, define indicadores de calidad estructurados en cuatro niveles jerárquicos: factor, criterio, métrica, elemento de evaluación, donde cada nivel inferior contiene los indicadores que conforman el nivel precedente. Otros autores identifican la calidad con el nivel de complejidad del ***software*** y definen dos categorías de métricas: de complejidad de programa o código, y de complejidad de sistema o estructura.

Todos los autores coinciden en que el *software* posee determinados índices medibles que son las bases para la calidad, el control y el perfeccionamiento de la productividad.

Una vez seleccionados los índices de calidad, se debe establecer el proceso de control, que requiere los siguientes pasos:

- Definir el *software* que va a ser controlado: clasificación por tipo, esfera de aplicación, complejidad, etc., de acuerdo con los estándares establecidos para el desarrollo del *software*.
- Seleccionar una medida que pueda ser aplicada al objeto de control. Para cada clase de *software* es necesario definir los indicadores y sus magnitudes.
- Crear o determinar los métodos de valoración de los indicadores: métodos manuales como cuestionarios o encuestas estándares para la medición de criterios periciales y herramientas automatizadas para medir los criterios de cálculo.
- Definir las regulaciones organizativas para realizar el control: quiénes participan en el control de la calidad, cuándo se realiza, qué documentos deben ser revisados y elaborados, etc.

A partir del análisis de todo lo anterior, siempre es recomendable la confección de una *Metodología para el Aseguramiento de la Calidad del Software* y el desarrollo de herramientas manuales y automatizadas de apoyo para la aplicación de las técnicas y procedimientos, de forma tal que se conforme un Sistema de Aseguramiento de la Calidad del *Software*.

CONCLUSIONES

Lograr el éxito en la producción de *software* es hacerlo con calidad y demostrarlo objetivamente. Esto sólo es posible con la implantación de un Sistema para el Aseguramiento de la Calidad del *Software* directamente relacionado con la política establecida para su elaboración y que esté en correspondencia con la definición internacional ISO de calidad, ampliamente aceptada, y por los estándares del grupo ISO 9000.

Desarrollo de una aplicación orientada a objetos y desarrollo de una aplicación estructurada.

Programación estructurada

La programación estructurada consiste en escribir un programa de acuerdo con unas reglas y un conjunto de técnicas. Las reglas son: el programa tiene un diseño modular, los módulos son diseñados descendentemente, cada módulo de programa se codifica usando tres estructuras de control

(secuencia, selección e iteración); es el conjunto de técnicas que han de incorporar: recursos abstractos; diseño descendente y estructuras básicas de control.

Descomponer un programa en términos de recursos abstractos consiste en descomponer acciones complejas en términos de acciones más simples capaces de ser ejecutadas en una computadora.

El diseño descendente se encarga de resolver un problema realizando una descomposición en otros más sencillos mediante módulos jerárquicos. El resultado de esta jerarquía de módulos es que cada módulo se refina por los de nivel más bajo que resuelven problemas más pequeños y contienen más detalles sobre los mismos.

Las estructuras básicas de control sirven para especificar el orden en que se ejecutarán las distintas instrucciones de un algoritmo. Este orden de ejecución determina el flujo de control del programa.

La programación estructurada significa:

- El programa completo tiene un diseño modular.
- Los módulos se diseñan con metodología descendente (puede hacerse también ascendente).
- Cada módulo se codifica utilizando las tres estructuras de control básicas: secuenciales, selectivas y repetitivas (ausencia total de sentencias ir → a (goto)).
- Estructuración y modularidad son conceptos complementarios (se solapan).

EJEMPLO 1.2. Calcular la media de una serie de números positivos, suponiendo que los datos se leen desde un terminal. Un valor de cero —como entrada— indicará que se ha alcanzado el final de la serie de números positivos.

El primer paso a dar en el desarrollo del algoritmo es descomponer el problema en una serie de pasos secuenciales. Para calcular una media se necesita sumar y contar los valores. Por consiguiente, el algoritmo en forma descriptiva sería:

inicio

1. Inicializar contador de números C y variable suma S a cero (S←0, C←1).

2. Leer un número en la variable N (leer(N))

3. Si el número leído es cero: (si (N =0) entonces)

3.1. Si se ha leído algún número (Si C>0)

- calcular la media; (media← S/C)
- imprimir la media; (Escribe(media)

3.2. si no se ha leído ningún número (Si C=0))

- escribir no hay datos.

3.3. Fin del proceso.

4. Si el numero leído no es cero : (Si (N <> 0) entonces

- calcular la suma; (S← S+N)

• incrementar en uno el contador de números; (C←C+1)

• ir al paso 2.

Fin

Un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias. Lenguajes de programación como C, Pascal, FORTRAN, y otros similares, se conocen como lenguajes procedimentales (por procedimientos). Es decir, cada sentencia o instrucción indica al compilador que realice alguna tarea: obtener una entrada, producir una salida, sumar tres números, dividir por cinco, etc. En el caso de pequeños programas, estos principios de organización (denominados paradigma) se demuestran eficientes. El programador sólo ha de crear esta lista de instrucciones en un lenguaje de programación, compilar en la computadora y ésta, a su vez, ejecuta las instrucciones.

Cuando los programas se vuelven más grandes, la lista de instrucciones aumenta considerablemente, de modo tal que el programador tiene muchas dificultades para controlar ese gran número de instrucciones. Para resolver este problema los programas

se descomponen en unidades más pequeñas que adoptan el nombre de funciones (procedimientos, subprogramas o subrutinas en otros lenguajes de programación). De este modo un programa orientado a procedimientos se divide en funciones, cada una de las cuales tiene un propósito bien definido y resuelve una tarea concreta, y se diseña una interfaz claramente definida (el prototipo o cabecera de la función) para su comunicación con otras funciones.

Con el paso de los años, la idea de romper un programa en funciones fue evolucionando y se llegó al agrupamiento de las funciones en otras unidades más grandes llamadas módulos (normalmente, en el caso de C++, denominadas archivos o ficheros); sin embargo, el principio seguía siendo el mismo: agrupar componentes que ejecutan listas de instrucciones (sentencias).

Existen varias razones de la debilidad de los programas estructurados para resolver problemas complejos.

Tal vez las dos razones más evidentes son éstas. Primera, las funciones tienen acceso ilimitado a los datos globales; segundo, las funciones inconexas y datos, fundamentos del paradigma procedimental proporcionan un modelo pobre del mundo real.

La programación orientada a objetos se desarrolló para tratar de paliar diversas limitaciones que se encontraban en anteriores enfoques de programación.

El paradigma de orientación a objetos

La programación orientada a objetos aporta un nuevo enfoque a los retos que se plantean en la programación estructurada cuando los problemas a resolver son complejos. Al contrario que la programación procedimental que enfatiza en los algoritmos, la POO enfatiza en los datos. En lugar de intentar ajustar un problema al enfoque procedimental de un lenguaje, POO intenta ajustar el lenguaje al problema. La idea es diseñar formatos de datos que se correspondan con las características esenciales de un problema. Los lenguajes orientados combinan en una única unidad o módulo, tanto los datos como las funciones que operan sobre esos datos. Tal unidad se llama objeto. Si se desea modificar los datos de un objeto, hay que realizarlo mediante las funciones miembros del objeto. Ninguna otra función puede acceder a los datos. Esto simplifica la escritura, depuración y mantenimiento del programa.

En el paradigma orientado a objetos, el programa se organiza como un conjunto finito de objetos que contienen datos y operaciones (funciones miembro en C++) que llaman a esos datos y que se comunican entre sí mediante mensajes.

Cuando se trata de resolver un problema con orientación a objetos, dicho problema no se descompone en funciones como en programación estructurada tradicional, sino en objetos. El pensar en términos de objetos tiene una gran ventaja: se asocian los objetos del problema a los objetos del mundo real.

Una clase puede describir las propiedades genéricas de un ejecutivo de una empresa (nombre, título, salario, cargo...) mientras que un objeto representará a un ejecutivo específico (Luis Mackoy, Lucas Soblechero). Un objeto es un elemento individual con su propia identidad; por ejemplo, un libro, un automóvil...

En general, una clase define qué datos se utilizan para representar un objeto y las operaciones que se pueden ejecutar sobre esos datos.

Los conceptos Clases y objetos se podrían asociar con los planos de una casa (Clase) y la casa (Objeto).

En el sentido estricto de programación, una clase es un tipo de datos. Diferentes variables se pueden crear de este tipo. En programación orientada a objetos, estas variables se llaman instancias. Las instancias son, por consiguiente, la realización de los objetos descritos en una clase. Estas instancias constan de datos o atributos descritos en la clase y se pueden manipular con las operaciones definidas dentro de ellas.

Una clase es una descripción general de un conjunto de objetos similares. Por definición todos los objetos de una clase comparten los mismos atributos (datos) y las mismas operaciones (métodos). Una clase encapsula las abstracciones de datos y operaciones necesarias para describir una entidad u objeto del mundo real.

El diseño de clases fiables y útiles puede ser una tarea difícil. Afortunadamente los lenguajes de programación orientada a objetos facilitan la tarea ya que incorporan clases existentes en su propia programación. Uno de los beneficios reales es que permite la reutilización y adaptación de códigos existentes y ya bien probados y depurados.

En el diseño de programas orientados a objetos se realiza en primer lugar el diseño de las clases que representan con precisión aquellas cosas que trata el programa; por ejemplo, un programa de dibujo, puede definir clases que representan rectángulos, líneas, pinceles, colores, etc. Las definiciones de clases, incluyen una descripción de operaciones permisibles para cada clase, tales como desplazamiento de un círculo o rotación de una línea. A continuación se prosigue el diseño de un programa utilizando objetos de las clases.

Los términos objeto e instancia se utilizan frecuentemente como sinónimos.

Las operaciones definidas en los objetos se llaman métodos. Cada operación llamada por un objeto se interpreta como un mensaje al objeto, que utiliza un método específico para procesar la operación, desde el punto de vista de implementación un objeto es una entidad que posee un conjunto de datos y un conjunto de operaciones (funciones o métodos). El estado de un objeto viene determinado por los valores que toman sus datos, cuyos valores pueden tener las restricciones impuestas en la definición del problema. Los datos se denominan también atributos y componen la estructura del objeto y las operaciones —también llamadas métodos— representan los servicios que proporciona.

TEMA V: Manejo de Errores

Introducción. Tipos de errores: errores de sintaxis, errores en tiempo de ejecución, errores lógicos. Tratamiento de excepciones. Herramientas de depuración. Caso práctico.

- Introducción

La mayor parte de los lenguajes de programación presentan al menos dos tipos de errores que permiten a los programadores manejar las fallas de los programas de una manera eficiente y que no resulte agresiva con el usuario final. Dichos errores son suelen ser los de compilación y errores en tiempo de ejecución.

Los programadores más experimentados cometen errores; y conocer cómo depurar una aplicación y encontrar esos errores es una parte importante de la programación. No obstante, antes de obtener información sobre el proceso de depuración, conviene conocer los tipos de errores que deberá buscar y corregir.

- Error de Sintaxis:

En programación, un **error de sintaxis** se produce al escribir, incorrectamente, alguna parte del código fuente de un programa. De forma que, dicho error impedirá, tanto al compilador como al intérprete, traducir dicha instrucción, ya que, ninguno de los dos entenderá qué le está diciendo el programador.

Los errores de sintaxis siempre ocurren antes de que el programa sea ejecutado. Es decir, un programa mal escrito no logra ejecutar ninguna instrucción. Por lo mismo, el error de sintaxis no es una excepción.

Los **errores de programación** pertenecen a tres categorías:

1. **Errores de compilación,**
2. **Errores en tiempo de ejecución y**
3. **Errores lógicos.**

1. Errores de compilación

Los **errores de compilación**, también conocidos como ***errores del compilador***, son errores que impiden que su programa se ejecute. Cuando se presiona F5 para ejecutar un programa por ejemplo en Visual Basic, se compila el código en un lenguaje binario que entiende el equipo. Si el compilador de Visual Basic se encuentra con código que no entiende, emite un error de compilador.

La mayoría de los errores del compilador se deben a errores cometidos al escribir el código. Por ejemplo, puede escribir mal una palabra clave, omitir alguna puntuación necesaria o intentar utilizar una instrucción **End If** sin antes utilizar una instrucción **If**.

Afortunadamente el Editor de código de Visual Basic fue diseñado para identificar estos errores antes de que se intente ejecutar el programa.

Los errores del compilador aparecen cuando el compilador de Visual Basic se encuentra con código irreconocible, generalmente porque se cometió algún error al escribir. Dado que los errores del compilador impiden que se ejecute un programa, deberá encontrarlos y corregirlos, o depurarlos, antes de ejecutar el programa.

- Encontrar y corregir errores del compilador

Encontrar los errores del compilador es bastante fácil, ya que el programa no se ejecuta hasta que se han corregido. Cuando se presiona F5, si hay algún error del compilador, aparecerá un cuadro de diálogo que indica "Errores al generar. ¿Desea continuar?". Si selecciona Sí, se ejecutará la última versión sin errores del programa; si selecciona No, el programa se detendrá y aparecerá la ventana Lista de errores.

La ventana Lista de errores muestra toda la información sobre el error, incluida su descripción y ubicación en el código. Si hace doble clic en el error en la Lista de errores, se resaltará la línea incorrecta del código en el Editor de código. También puede presionar F1 para mostrar Ayuda y obtener más información sobre el error y cómo corregirlo.

El Editor de código de Visual Basic también puede ayudar a encontrar y corregir los errores del compilador antes incluso de que se intente ejecutar el programa. Mediante una característica llamada **IntelliSense**, Visual Basic observa el código a medida que se escribe y si encuentra código que producirá un error del compilador, lo subraya con una línea ondulada de color azul. Si mantiene presionado el mouse sobre esa línea, se muestra un mensaje que describe el error. Si la ventana Lista de errores está visible, también mostrará los mensajes de error.

- Para encontrar y corregir errores del compilador

1. En el menú Archivo, seleccione Nuevo proyecto.
2. En el panel Plantillas, en el cuadro de diálogo Nuevo proyecto, haga clic en Aplicación para Windows.
3. En el cuadro Nombre, escriba CompilerErrors y haga clic en Aceptar.
4. Se abre un nuevo proyecto de formularios Windows Forms.
5. Haga doble clic en el formulario para abrir el Editor de código.
6. En el controlador de eventos **Form_Load,** agregue el siguiente código.

VB

```
End If
```

7. Presione ENTRAR. Verá una línea ondulada de color azul debajo de End If.

   Si mantiene presionado el mouse sobre la línea, verá el mensaje "'End If' debe ir precedida por la instrucción 'If ' " correspondiente.

8. Cambie el código para que tenga la siguiente apariencia.

   VB

   ```
   If 1 < 2 Then

   End If
   ```

   Observe que ha desaparecido la línea ondulada de color azul.

9. Agregue la nueva línea de código siguiente después de la instrucción If... Then.

   VB

   ```
   MgBox("Hello")
   ```

10. Presione F5 para ejecutar el programa. Aparecerá un cuadro de diálogo con el mensaje "**Errores al generar. ¿Desea continuar y ejecutar la última versión generada correctamente**?"
11. Haga clic en **No**. Se mostrará la ventana **Lista de errores** con el mensaje de error "No se ha declarado el 'nombre MgBox'".
12. Haga doble clic en el mensaje de error de la **Lista de errores** y cambie el código por MsgBox("Hello").
13. Presione F5 de nuevo. Ahora el programa debería ejecutarse y causar la aparición de un cuadro de mensaje.

2. Errores en tiempo de ejecución.

Los **errores en tiempo de ejecución** son errores que aparecen mientras se ejecuta su programa. Estos errores aparecen normalmente cuando su programa intenta una operación que es imposible que se lleve a cabo.

Los errores en tiempo de ejecución se producen cuando el programa intenta realizar una operación que es imposible finalizar. Cuando se produce un error en tiempo de ejecución, el programa se detiene y aparece un mensaje de error; debe depurar el error y corregirlo para que el programa pueda continuar.

Un ejemplo de esto es la división por cero. Suponga que tiene la instrucción siguiente:

```
Speed = Miles / Hours
```

Si la variable `Hours` tiene un valor de 0, se produce un error en tiempo de ejecución en la operación de división. El programa se debe ejecutar para que se pueda detectar este error y si `Hours` contiene un valor válido, no se producirá el error.

Cuando aparece un error en tiempo de ejecución, puede utilizar las herramientas de depuración de Visual Basic para determinar la causa.

- Encontrar y corregir errores en tiempo de ejecución.

La mayoría de los errores en tiempo de ejecución se producen porque se cometió un error en el código; por ejemplo, olvidó asignar un valor a una variable antes de utilizarla. Cuando se ejecute el programa y se descubra el error, el programa se detendrá y el cuadro de diálogo **Ayudante de excepciones** se mostrará en la ventana Editor de código. Cuando esto sucede, el programa está en modo de interrupción, que es el modo en que se realiza la depuración.

El cuadro de diálogo **Ayudante de excepciones** contiene una descripción del error, así como sugerencias para la solución de problemas que indican la causa. Puede hacer clic en las sugerencias sobre solución de problemas para mostrar los temas de Ayuda y obtener más detalles.

Es necesario corregir el error para que pueda continuar con el programa; para ello, debe inspeccionar el código para encontrar la causa del error. Por ejemplo, si sospecha que se produjo un error porque una variable contiene el valor equivocado, estando todavía en el modo de interrupción, puede utilizar IntelliSense para ver el valor de la variable. Cuando se coloca el mouse sobre la variable en el Editor de código, la información sobre herramientas muestra el valor de la variable. Si el valor no es lo que esperaba, compruebe en el código anterior dónde se estableció el valor y después arregle el código y continúe.

Para revisar el valor de una variable

1. En el menú **Archivo**, seleccione **Nuevo proyecto**.
2. En el panel **Plantillas**, en el cuadro de diálogo **Nuevo proyecto**, haga clic en **Aplicación para Windows**.
3. En el cuadro **Nombre**, escriba RunTimeErrors y haga clic en **Aceptar**.

   Se abrirá un nuevo proyecto de formularios Windows Forms.

4. Haga doble clic en el formulario para abrir el Editor de código.
5. En el controlador de eventos **Form_Load**, agregue el siguiente código.

   VB

```
Dim miles As Integer = 0
Dim hours As Integer = 0
Dim speed As Integer = 0
```

VB

```
miles = 55
speed = miles / hours
MsgBox(CStr(speed) & " miles per hour")
```

6. Presione **F5** para ejecutar el programa. Aparece un cuadro de diálogo **Ayudante de excepciones** con el mensaje **"No se controló OverflowException".**

   Una línea de puntos que va del cuadro de diálogo a su archivo de código señala la línea de código que produjo el error.

   Observe que la primera sugerencia sobre solución de problemas del **Ayudante de excepciones** indica que debe asegurarse de no estar dividiendo por cero.

   Mueva el mouse sobre la variable miles y manténgalo ahí durante unos segundos. La información sobre herramientas que verá dice "miles 55".

   Ahora mueva el mouse sobre la variable hours; la información sobre herramientas debe decir "hours 0".

   Debido a que no se puede dividir por cero y el valor de hours es cero, ya ha encontrado la causa del error: no haber actualizado el valor de hours.

   Agregue la siguiente línea de código sobre la línea miles = 55.

   VB

   ```
   hours = 2
   ```

   Haga clic en la flecha amarilla situada en el margen izquierdo del código y arrástrela hasta la línea **hours = 2.**

   Esto permite que el programa continúe desde esa línea en lugar de continuar desde la línea que contiene el error. Para que se reconozca la solución del error es necesario ejecutar la nueva línea de código recién agregada.

   Presione **F5** para que el programa continúe. Aparece un cuadro de diálogo que muestra "28 miles per **hour".**

3. Errores lógicos

Los **errores lógicos** son errores que impiden que su programa haga lo que estaba previsto. Su código puede compilarse y ejecutarse sin errores, pero el resultado de una operación puede generar un resultado no esperado.

Por ejemplo, puede tener una variable llamada `FirstName` y establecida inicialmente en una cadena vacía. Después en el programa, puede concatenar `FirstName` con otra variable

denominada `LastName` para mostrar un nombre completo. Si olvida asignar un valor a `FirstName`, sólo se mostrará el apellido, no el nombre completo como pretendía.

Los **errores lógicos** son los más difíciles de detectar y corregir, pero Visual Basic también dispone de herramientas de depuración que facilitan el trabajo.

- Detectar errores lógicos.

Los errores lógicos, pueden ser los más difíciles de descubrir. Con los errores lógicos no se obtiene ninguna advertencia, se ejecutará el programa pero proporcionará resultados incorrectos. Es necesario investigar el código y determinar la razón del problema.

Afortunadamente, las herramientas de depuración de Visual Basic pueden ayudar. Dos técnicas de depuración, que establecen puntos de interrupción e instrucciones paso a paso a través del código, permiten inspeccionar el código línea por línea mientras se ejecuta para encontrar el error.

Se puede establecer un punto de interrupción en el **Editor de código** para cualquier línea ejecutable de código. Cuando se ejecuta el programa, los puntos de interrupción fuerzan que se detenga y el programa entra en el modo de interrupción cuando llega a esa línea de código. Puede obtener la información que desee sobre el estado del programa en ese momento. Puede comprobar el valor de cualquier variable, comprobar expresiones en la ventana Inmediato o realizar cambios en el código con Editar y continuar.

Cuando está en modo de interrupción, puede recorrer el código, ejecutando línea por línea para ver cómo funciona. Al presionar la **tecla F8,** se ejecutará la línea de código actual y se detendrá en la línea siguiente. Puede inspeccionar los valores de variables para ver cómo cambian de una línea a la siguiente.

Si la línea de código actual llama a una función o procedimiento **Sub** en otra parte del código, cuando presiona **F8**, la ejecución se desplazará a ese procedimiento. Una vez que se haya ejecutado ese procedimiento, el programa volverá a la línea siguiente a la que llamó al procedimiento. Si no desea recorrer un procedimiento, puede presionar **MAYÚS+F8** para saltarlo.

Para observar un error lógico

1. En el menú **Archivo**, seleccione **Nuevo proyecto**.
2. En el panel **Plantillas**, en el cuadro de diálogo **Nuevo proyecto**, haga clic en **Aplicación para Windows**.
3. En el cuadro **Nombre**, escriba **`LogicErrors`** y haga clic en **Aceptar**.

   Se abre un nuevo proyecto de formularios Windows Forms.
4. Desde el **Cuadro de herramientas**, arrastre dos controles TextBox y un control Button hacia el formulario.
5. Haga doble clic en **Button1** para abrir el Editor de código.
6. En el controlador de eventos **Button1_Click**, agregue el siguiente código.

VB

```
Dim minutes As Integer = CInt(Textbox1.Text)
Dim miles As Double = CDbl(Textbox2.Text)
Dim hours As Double = 0
hours = minutes / 60
MsgBox("Average speed " & GetMPH(hours, miles))
```

7. Debajo de la línea **`End Sub`**, agregue la siguiente función.

VB

```
Function GetMPH(ByVal miles As Double, ByVal hours As Double) _
As String
    GetMPH = CStr(miles / hours)
End Function
```

8. Presione **F5** para ejecutar el programa. En el primer cuadro de texto, escriba 10 (para representar 10 minutos) y en el segundo cuadro de texto, escriba 5 (para representar las millas) y, a continuación, haga clic en **Button1**.

   Aparecerá un cuadro con el mensaje "Average speed 0.03333334" (velocidad media 0,03333334) ; no obstante, si recorre 5 millas en diez minutos, la respuesta correcta serían 30 mph.

   Mantenga abierto el proyecto: en el siguiente procedimiento aprenderá cómo encontrar el error lógico.

- Encontrar errores lógicos

En el último ejemplo, algo está obviamente mal con la lógica del programa. Según el resultado, viaja menos de una milla por hora, no treinta millas por hora como espera, pero ¿dónde está el error?

En el siguiente procedimiento se establecerá un punto de interrupción y se examinará el código para encontrar el error.

Para establecer un punto de interrupción y recorrer el código

1. En el Editor de código, busque la línea hours = minutes / 60 y haga clic en el margen izquierdo.

   Aparecerá un punto rojo en el margen y el código resaltado en rojo, lo que representa un punto de interrupción.

2. Presione F5 para ejecutar el programa nuevamente. En el primer cuadro de texto, escriba 10 y en el segundo cuadro de texto, escriba 5. Haga clic en **Button1.**

   El programa se detendrá cuando llegue al punto de interrupción. La línea hours = minutes / 60 aparecerá resaltada en amarillo.

Inspeccione los valores de las variables manteniendo el mouse sobre ellos; el valor de hours debe ser 0 y el valor de minutes debe ser 10.

3. Presione **F8** para ejecutar la línea hours = minutes / 60 y pasar a la siguiente línea.

   Inspeccione los valores de las variables de la línea MsgBox("Average speed " & GetMPH(hours, miles)), el valor de hours debe ser ahora 0.166666672 y el valor de miles debe ser 5.0.

4. Presione **F8** de nuevo para ejecutar la línea actual.

   Observe que la ejecución baja a la línea Function GetMPH.

   Inspeccione los valores de las variables en esta línea; observará que el valor de miles es ahora 0.166666672y el de hours es 5.0, lo contrario de lo que eran en la línea anterior. Ha encontrado el error.

   Mantenga abierto el proyecto: en el siguiente procedimiento aprenderá a corregir el error lógico.

- Corregir errores lógicos.

En el último procedimiento, los valores para las variables miles y hours cambiaron de lugar. ¿Puede identificar la causa?

Si examina la línea **MsgBox**("Average speed " & **GetMPH**(hours, miles)), verá que a la función **GetMPH** se pasan dos argumentos, hours y miles, en ese orden. Si examina la declaración de función Function **GetMPH(ByVal miles As Double, ByVal hours As Double)...,** observará que los argumentos se muestran como miles primero y como hours después.

Se produjo un error en la lógica porque los argumentos se pasaron en el orden equivocado, produciendo un cálculo incorrecto. Si los argumentos hubieran sido de tipos diferentes, habría visto un error en tiempo de ejecución, pero como los argumentos eran del mismo tipo, no se produjo el error. Fue un error simple, pero el error resultante fue difícil de encontrar.

En el siguiente procedimiento se establecerá un punto de interrupción y se recorrerá el código para encontrar el error.

- Para corregir el error lógico

1. En el Editor de código, cambie la línea **MsgBox("Average speed " & GetMPH(hours, miles))** para que se lea de la siguiente manera:

   VB

   ```
   MsgBox("Average speed " & GetMPH(miles, hours))
   ```

2. Haga clic en el punto rojo en el margen izquierdo para borrar el punto de interrupción.
3. Presione **F5** para ejecutar el programa. En el primer cuadro de texto, escriba 10 y en el segundo cuadro de texto, escriba 5. A continuación, haga clic en **Button1.**

   Esta vez el cuadro de mensaje debe mostrar el resultado correcto, "Average speed 30" (velocidad media 30).

Puede parecer que se corrigió el programa, pero hay otro error lógico aun más difícil de encontrar.

Para corregir el error

1. En el Editor de código, cambie el código en el controlador de eventos **Button1_Click** de la siguiente manera:

   VB

```
Dim minutes As Integer = CInt(Textbox1.Text)
Dim miles As Double = CDbl(Textbox2.Text)
Dim hours As Double = 0
If minutes <= 0 Or miles <= 0 Then
  MsgBox("Please enter a number greater than zero")
Else
  hours = minutes / 60
  MsgBox("Average speed " & GetMPH(miles, hours))
End If
```

2. Presione **F5** para ejecutar el programa nuevamente. En el primer cuadro de texto, escriba `0`, y en el segundo, escriba `5`. A continuación, haga clic en **Button1**.

   Aparecerá el cuadro de mensaje indicándole que especifique un número mayor que 0. Inténtelo probando el programa con otras combinaciones de números hasta que esté seguro de que se ha corregido el error.

Qué hacer cuando algo sale mal: control de errores

Incluso los programas mejor diseñados a veces encuentran errores. Algunos errores son defectos en el código que se pueden encontrar y corregir. Otros errores son una consecuencia natural del programa; por ejemplo, el programa puede intentar abrir un archivo que ya está en uso. En casos así, los errores se pueden predecir, pero no evitar. Como desarrollador, es su trabajo predecir estos errores y ayudar a que el programa los solucione.

Errores en tiempo de ejecución

Un error que se produce mientras un programa se está ejecutando se *llama error en tiempo de ejecución*. Los errores en tiempo de ejecución se producen cuando un programa trata de hacer algo para lo cual no fue diseñado. Por ejemplo, si el programa intenta realizar una operación no válida, como convertir una cadena no numérica en un valor numérico, se producirá un error en tiempo de ejecución.

Cuando se produce un *error en tiempo de ejecución*, el programa produce una excepción, que soluciona los errores buscando código dentro del programa para tratar el error. Si no se encuentra tal código, se detiene el programa y se tiene que reiniciar. Dado que esto puede

conducir a la pérdida de datos, es prudente crear el código de control de errores dondequiera que se tenga previsto que se produzcan errores.

El bloque Try...Catch...Finally.

Esta instrucción proporciona una manera de controlar algunos o todos los errores posibles que pueden ocurrir en un bloque de código determinado mientras se ejecuta el código.

Se puede utilizar el bloque **Try...Catch...Finally** para controlar ***errores en tiempo de ejecución*** en el código. Puede utilizar **Try** para un segmento de código; si ese código produce una excepción, salta al bloque **Catch** y se ejecuta el código del bloque **Catch**. Después de que ese código ha finalizado, se ejecuta cualquier código en el bloque **Finally.** La instrucción **End Try** cierra el bloque **Try...Catch...Finally** completo. En el ejemplo siguiente se ilustra cómo se utiliza cada bloque.

Sintaxis:

```
Try
    [ tryStatements ]
    [ Exit Try ]
[ Catch [ exception [ As type ] ] [ When expression ]
    [ catchStatements ]
    [ Exit Try ] ]
[ Catch ... ]
[ Finally
    [ finallyStatements ] ]
End
```

Partes

tryStatements

Opcional. Instrucciones en las que puede ocurrir un error. Puede ser una instrucción compuesta.

Catch

Opcional. Se permite utilizar varios bloques **Catch**. Si tiene lugar una excepción mientras se procesa el bloque **Try**, cada instrucción **Catch** se examina en orden textual para determinar si controla la excepción; el parámetro exception representa la excepción que se ha producido.

exception

Opcional. Cualquier nombre de variable. El valor inicial de exception es el valor del error producido. Se utiliza con **Catch** para especificar la captura del error. Si se omite, la instrucción **Catch** detecta cualquier excepción.

type

Opcional. Especifica el tipo de filtro de clase. Si el valor de exception es del tipo especificado en type o de un tipo derivado, el identificador queda enlazado al objeto de excepción.

When

Opcional. Una instrucción **Catch** con una cláusula **When** sólo detecta las excepciones cuando expression se evalúa como **True**. Una cláusula **When** sólo se aplica después de comprobar el tipo de la excepción y expression puede hacer referencia al identificador que representa la excepción.

expression

Opcional. Debe ser convertible implícitamente a **Boolean**. Cualquier expresión que describe un filtro genérico. Se utiliza normalmente para filtrar por número de error. Se utiliza con la palabra clave **When** para especificar las circunstancias bajo las que se captura el error.

catchStatements

Opcional. Instrucciones para controlar los errores que se producen en el bloque **Try**. Puede ser una instrucción compuesta.

Exit Try

Opcional. Palabra clave que interrumpe la ejecución de la estructura **Try...Catch...Finally**. La ejecución se reanuda con el código que sigue inmediatamente a la instrucción **End Try**. Se ejecutará la instrucción **Finally** todavía. No se permite en bloques **Finally**.

Finally

Opcional. Siempre se ejecuta un bloque **Finally** cuando la ejecución sale de cualquier parte de la instrucción **Try**.

finallyStatements

Opcional. Instrucciones que se ejecutan después de las demás operaciones de procesamiento de error.

End Try

Finaliza la estructura **Try...Catch...Finally**.

Las variables locales de un bloque **Try** no se encuentran disponibles en un bloque **Catch** porque se trata de bloques independientes. Si se desea utilizar una variable en más de un bloque, se debe declarar la variable fuera de la estructura **Try...Catch...Finally**.

El bloque **Try** contiene código donde puede producirse un error, mientras que el bloque **Catch** contiene el código para controlar cualquier error que tenga lugar. Si se produce un error en el bloque **Try**, el control del programa pasa a la instrucción **Catch** apropiada para su

procesamiento. El argumento exception es una instancia de la clase **Exception** o una clase que se deriva de la clase **Exception**. La instancia de la clase **Exception** corresponde al error que se produjo en el bloque **Try**. La instancia contiene información acerca del error, como el número de error y el mensaje.

Si una instrucción **Catch** no especifica un argumento exception, detectará cualquier tipo de excepción del sistema o de la aplicación. Esta variación debe utilizarse siempre como el último bloque **Catch** en la estructura **Try...Catch...Finally**, después de detectar todas las excepciones específicas anticipadas. El flujo de control nunca puede alcanzar un bloque **Catch** situado detrás de **Catch** sin un argumento exception.

En situaciones de confianza parcial, como una aplicación alojada en un recurso compartido de red, **Try...Catch...Finally** no detectará las excepciones de seguridad que se produzcan antes de invocar al método que contiene la llamada. El ejemplo siguiente, si se coloca en un recurso compartido de servidor y se ejecuta desde el mismo, producirá el error: "Sub System.Security.SecurityException: Error de solicitud".

Ejemplo de uso de Try, Catch , Finally

VB

```
Try
  ' Code here attempts to do something.
Catch
  ' If an error occurs, code here will be run.
Finally
  ' Code in this block will always be run.
End Try
```

Primero, se ejecuta el código del bloque **Try.** Si se ejecuta sin error, el programa omite el bloque **Catch** y ejecuta el código del bloque **Finally**. Si se produce un error en el bloque **Try**, la ejecución salta inmediatamente al bloque **Catch** y se ejecuta el código que se encuentra allí; luego se ejecuta el código del bloque **Finally**.

Para utilizar el bloque Try...Catch

1. En el menú **Archivo**, seleccione **Nuevo proyecto**.
2. En el cuadro de diálogo **Nuevo proyecto**, en el panel **Plantillas**, haga clic en **Aplicación para Windows**.
3. En el cuadro **Nombre**, escriba **TryCatch** y haga clic en **Aceptar**.
   Se abre un nuevo proyecto de formularios Windows Forms.
4. En el **Cuadro de herramientas**, arrastre un control **TextBox** y un control **Button** al formulario.
5. Haga doble clic en **Button** para abrir el Editor de código.
6. En el controlador de eventos **Button1_Click**, escriba el siguiente código:

VB

```
Try
  Dim aNumber As Double = CDbl(Textbox1.Text)
  MsgBox("You entered the number " & aNumber)
Catch
  MsgBox("Please enter a number.")
Finally
  MsgBox("Why not try it again?")
End Try
```

7. Presione F5 para ejecutar el programa.
8. En el cuadro de texto, escriba un valor numérico y haga clic en el botón. Aparece un cuadro de mensaje que muestra el número que ha escrito, seguido por una invitación para volver a intentarlo.
9. A continuación, escriba un valor no numérico en el cuadro de texto, como una palabra y haga clic en el botón. Esta vez, cuando el programa intente convertir el texto del cuadro de texto en un número, no podrá hacerlo y se producirá un error. En lugar de finalizar el código en el bloque **Try,** se ejecuta el bloque **Catch** y aparece un cuadro de mensaje solicitando que se escriba un número. Se ejecuta el bloque **Finally** y se le invita a intentarlo de nuevo.

Detectar una excepción en Visual Basic

Este ejemplo muestra cómo se utiliza un bloque **Try** y **Catch** para detectar excepciones.

Ejemplo

Este ejemplo muestra cómo se utiliza un bloque **Try...Catch** para detectar una excepción **OverflowException.**

Este ejemplo de código también está disponible en el fragmento de código de IntelliSense. En el selector de fragmentos de código, se encuentra en **Lenguaje Visual Basic**.

VB

```
Dim Top As Double = 5
Dim Bottom As Double = 0
Dim Result As Integer
Try
    Result = CType(Top / Bottom, Integer)
Catch Exc As System.OverflowException
    MsgBox("Attempt to divide by zero resulted in overflow")
End Try
```

Este ejemplo necesita:

- Una referencia al espacio de nombres System.

El ejemplo de código siguiente implementa un bloque **Try...Catch** que controla Exception, IOException y todas las excepciones que se derivan de **IOException**.

VB

```
Try
    ' Add code for your I/O task here.
Catch dirNotFound As System.IO.DirectoryNotFoundException
    Throw dirNotFound
Catch fileNotFound As System.IO.FileNotFoundException
    Throw fileNotFound
Catch pathTooLong As System.IO.PathTooLongException
    Throw pathTooLong
Catch ioEx As System.IO.IOException
    Throw ioEx
Catch security As System.Security.SecurityException
    Throw security
Catch ex As Exception
    Throw ex
Finally
    ' Dispose of any resources you used or opened in the Try block.
End Try
```

Agregue el código que desee ejecutar al bloque **Try**.

Programación eficaz.

Utilice este bloque de código como punto inicial para ajustar una operación de datos en una instrucción **Try...Catch**. Este bloque **Try...Catch** está diseñado para detectar y volver a producir todas las excepciones. Es posible que no sea la elección más adecuada para su proyecto.

Puede reducir la probabilidad de excepciones utilizando los controles de formularios Windows Forms como el componente OpenFileDialog (Componente, formularios Windows Forms) y los controles de componente SaveFileDialog (Componente, formularios Windows Forms) que limitan las elecciones del usuario a los nombres de archivo válidos. La propiedad FileInfo.Exists puede comprobar si existe un archivo antes de intentar abrirlo. No obstante, el uso de estos controles y clases no es infalible. El sistema de archivos puede cambiar entre el momento en el que el usuario selecciona un archivo y el momento en el que se ejecuta el código. Por ello, cuando se trabaja con archivos es prácticamente obligatorio realizar un control de excepciones.

Seguridad.

Para muchas tareas de archivos, el ensamblado requiere un nivel de privilegios concedido por la clase FileIOPermission. Si realiza una ejecución en un contexto de confianza parcial, el código podría desencadenar una excepción por falta de privilegios.

Solucionar problemas de control de excepciones.

Visual Basic admite el control estructurado de excepciones, que puede utilizar para crear y mantener programas mediante controladores de errores consistentes y exhaustivos. El **control estructurado de excepciones** es un código diseñado para detectar y dar respuesta a los errores que se producen durante la ejecución, mediante la combinación de una estructura de control (similar a **Select Case** o **While**) con excepciones, bloques de código protegidos y filtros.

Excepciones internas

En los casos en los que una excepción se produce como un resultado directo de una excepción anterior, la propiedad InnerException describe el error original. Esta información le ayuda a controlar el error de un modo más eficaz. Si no hay ningún error original, el valor de **InnerException** será una referencia nula o **Nothing** en Visual Basic. Esta propiedad es de sólo lectura.

Para comprobar una propiedad InnerException

- Compruebe la propiedad **InnerException** de la excepción para determinar la causa del error original.

```
Try
    My.Computer.FileSystem.CopyFile("file1", "file2")
Catch ex As System.IO.IOException
    MsgBox(ex.InnerException)
End Try
```

Instrucciones Try...Catch

Su código puede no detectar correctamente las excepciones si ordena los bloques **Catch** incorrectamente. Las instrucciones **Catch** deben ir de lo más específico a lo menos específico. Un bloque **Catch** por sí mismo detectará todas las excepciones derivadas de Exception y, por consiguiente, siempre debe ser el último bloque antes de **Finally**.

En el siguiente ejemplo simplificado se muestra la estructura de la instrucción **Try...Catch...Finally**.

Ejemplo

VB

```
Public Sub TryExample()
    Dim x As Integer = 5   ' Declare variables.
    Dim y As Integer = 0
    Try                    ' Set up structured error handling.
        x = x \ y          ' Cause a "Divide by Zero" error.
    Catch ex As Exception When y = 0        ' Catch the error.
        Beep()
        MsgBox("You tried to divide by 0.") 'Show an explanatory message.
    Finally
        Beep()             ' This line is executed no matter what.
    End Try
End Sub
```

TEMA VI: Técnicas de Validación y Validación

Introducción. Técnicas de prueba. Pruebas de caja blanca. Pruebas de caja negra. Niveles de prueba. Pruebas unitarias. Pruebas de integración. Pruebas implantación. Pruebas de aceptación. Pruebas de regresión. Gestión de los procesos de prueba. Planificación y seguimiento. Herramientas de apoyo. Caso práctico.

1. INTRODUCCION

Nunca se dará suficiente importancia a las pruebas del software y sus implicaciones en la calidad del software. El desarrollo de sistemas de software implica una serie de actividades de producción en las que las posibilidades de que aparezca el fallo humano son enormes. Los errores pueden empezar a darse desde el primer momento del proceso, en el que los objetivos pueden estar especificados de forma errónea o imperfecta, así como en posteriores pasos de diseño y desarrollo. Debido a la imposibilidad humana de trabajar y comunicarse de forma perfecta, el desarrollo de software ha de ir acompañado de una actividad que garantice la calidad.

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. No es raro que una organización de desarrollo de software emplee entre el 30 y el 40 por ciento del esfuerzo total de un proyecto en las pruebas. En casos extremos, las pruebas del software para actividades críticas (por ejemplo, control de tráfico aéreo, control de reactores nucleares) puede costar de tres a cinco veces más que el resto de los pasos de la ingeniería del software juntos!

Las pruebas presentan una interesante anomalía para el ingeniero del software. Durante las fases anteriores de definición y de desarrollo, el ingeniero intenta construir el software partiendo de un concepto abstracto y llegando a una implementación tangible. A continuación, llegan las pruebas. El ingeniero crea una serie de casos de prueba que intentan «demoler» el software construido. De hecho, las pruebas son uno de los pasos de la ingeniería del software que se puede ver (por lo menos, psicológicamente) como destructivo en lugar de constructivo.

Los ingenieros del software son, por naturaleza, personas constructivas. Las pruebas requieren que se descarten ideas preconcebidas sobre la «corrección» del software que se acaba de desarrollar y se supere cualquier conflicto de intereses que aparezcan cuando se descubran errores. Beizer [BE1901 describe eficientemente esta situación cuando plantea:

Existe un mito que dice que si fuéramos realmente buenos programando, no habría errores que buscar. Si tan sólo fuéramos realmente capaces de concentrarnos, si todo el mundo empleara técnicas de codificación estructuradas, el diseño descendente, las tablas de decisión, si los programas *se* escribieran en un lenguaje apropiado, si tuviéramos siempre la solución más adecuada, entonces no habría errores. Así es el mito. Según el mito, existen errores porque somos malos en lo que hacemos, y si somos malos en lo que hacemos, deberíamos sentirnos culpables por ello. Por tanto, las pruebas, con el diseño de casos de prueba, son un

reconocimiento de nuestros fallos, lo que implica una buena dosis de culpabilidad. Y lo tediosas que son las pruebas son un justo castigo a nuestros errores. ¿Castigados por qué? ¿Por ser humanos? ¿Culpables por qué? ¿Por no conseguir una perfección inhumana? ¿Por no poder distinguir en& lo que otro programador piensa y lo que dice? ¿Por no tener telepatía? ¿Por no resolver los problemas de comunicación humana que han estado presentes durante cuarenta siglos? ¿Deben infundir culpabilidad las pruebas? ¿Son realmente destructivas? La respuesta a estas preguntas es: Sin embargo, los objetivos de las pruebas son algo diferentes de lo que podríamos esperar.

2. TECNICAS DE PRUEBA

En un excelente libro sobre las pruebas del software, Glen Myers establece varias normas que pueden servir acertadamente como objetivos de las pruebas:

1. La prueba es el proceso de ejecución de un programa con la intención de descubrir un error.

2. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.

3. Una prueba tiene éxito si descubre un error no detectado hasta entonces.

¿Cuál es nuestro primer objetivo cuando probamos el software?

Los objetivos anteriores suponen un cambio dramático de punto de vista. Nos quitan la idea que, normalmente, tenemos de que una prueba tiene éxito si no descubre errores.

Nuestro objetivo es diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y de esfuerzo.

Si la prueba se lleva a cabo con éxito (de acuerdo con el objetivo anteriormente establecido), descubrirá errores en el software. Como ventaja secundaria, la prueba demuestra hasta qué punto las funciones del software parecen funcionar de acuerdo con las especificaciones y parecen alcanzarse los requisitos de rendimiento. Además, los datos que se van recogiendo a medida que se lleva a cabo la prueba proporcionan una buena indicación de la fiabilidad del software y, de alguna manera, indican la calidad del software como un todo. Pero, la prueba no puede asegurar la ausencia de defectos; sólo puede demostrar que existen defectos en el software.

Principios de las pruebas

Antes de la aplicación de métodos para el diseño de casos de prueba efectivos, un ingeniero del software deberá entender los principios básicos que guían las pruebas del software. Davis sugiere un conjunto' de principios de prueba que se han adaptado para usarlos en este libro:

- *A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente.* Como hemos visto, el objetivo de las pruebas del software es descubrir errores. Se entiende que los defectos más graves (desde el punto de vista del cliente) son aquellos que impiden al programa cumplir sus requisitos.

- Las pruebas deberían planificarse mucho antes de que empiecen. La planificación de las pruebas pueden empezar tan pronto como esté completo el modelo de requisitos. La definición detallada de los casos de prueba puede empezar tan pronto como el modelo de diseño se ha consolidado. Por tanto, se pueden planificar y diseñar todas las pruebas antes de generar ningún código.
- El principio de Pareto es aplicable a la prueba del software. Dicho de manera sencilla, el principio de Pareto implica que al 80 por 100 de todos los errores descubiertos durante las pruebas se les puede hacer un seguimiento hasta un 20 por 100 de todos los módulos del programa. El problema, por supuesto, es aislar estos módulos sospechosos y probarlos concienzudamente.
- Las pruebas deberían empezar por «lo pequeño» y progresar hacia «lo grande». Las primeras pruebas planeadas y ejecutadas se centran generalmente en módulos individuales del programa. A medida que avanzan las pruebas, desplazan su punto de mira en un intento de encontrar errores en grupos integrados de módulos y finalmente en el sistema entero.
- No son posibles las pruebas exhaustivas. El número de permutaciones de caminos para incluso un programa de tamaño moderado es excepcionalmente grande (vea la Sección 17.2 para un estudio más avanzado). Por este motivo, es imposible ejecutar todas las combinaciones de caminos durante las pruebas. Es posible, sin embargo, cubrir adecuadamente la lógica del programa y asegurarse de que se han aplicado todas las condiciones en el diseño a nivel de componente.
- Para ser más eficaces, las pruebas deberían ser realizadas por un equipo independiente. Por «más eficaces » queremos referirnos a pruebas con la más alta probabilidad de encontrar errores (el objetivo principal de las pruebas). Por las razones que se expusieron anteriormente en este capítulo, y que se estudian con más detalle en el Capítulo 18, el ingeniero del software que creó el sistema no es el más adecuado para llevar a cabo las pruebas para el software.

Facilidad de prueba

En circunstancias ideales, un ingeniero del software diseña un programa de computadora, un sistema o un producto con la «facilidad de prueba» en mente. Esto permite a los encargados de las pruebas diseñar casos de prueba más fácilmente. Pero, ¿qué es la «facilidad de prueba» James Bach2 describe la facilidad de prueba de la siguiente manera:

La facilidad de prueba del software es simplemente la facilidad con la que se puede probar un programa de computadora. Como la prueba es tan profundamente difícil, merece la pena saber qué se puede hacer para hacerlo más sencillo. **A** veces los programadores están dispuestos a hacer cosas que faciliten el proceso de prueba **y** una lista de comprobación de posibles puntos de diseño, características, etc., puede ser útil a la hora de negociar con ellos.

Existen, de hecho, métricas que podrían usarse para medir la facilidad de prueba en la mayoría de sus aspectos. **A** veces, la facilidad de prueba se usa para indicar lo adecuadamente que un conjunto particular de pruebas va a cubrir un producto. También es empleada por los militares para indicar lo fácil que se puede comprobar y reparar una herramienta en plenas maniobras. Esos dos significados no son lo mismo que «facilidad de prueba del software». La

siguiente lista de comprobación proporciona un conjunto de características que llevan a un software fácil de probar.

Operatividad. «Cuanto mejor funcione, más eficientemente se puede probar.»

- El sistema tiene pocos errores (los errores añaden sobrecarga de análisis **y** de generación de informes al proceso de prueba).
- Ningún error bloquea la ejecución de las pruebas.
- El producto evoluciona en fases funcionales (permite simultanear el desarrollo y las pruebas).

Observabilidad. «Lo que ves es lo que pruebas.»

- Se genera una salida distinta para cada entrada.
- Los estados y variables del sistema están visibles o se pueden consultar durante la ejecución.
- Los estados **y** variables anteriores del sistema están visibles o se pueden consultar (por ejemplo, registros de transacción).
- Todos los factores que afectan a los resultados están visibles.
- Un resultado incorrecto se identifica fácilmente.
- Los errores internos se detectan automáticamente a través de mecanismos de auto-comprobación.
- Se informa automáticamente de los errores internos.
- El código fuente es accesible.

Controlabilidad «Cuanto mejor podamos controlar el software, más se puede automatizar y optimizar.»

- Todos los resultados posibles se pueden generar a través de alguna combinación de entrada.
- Todo el código es ejecutable a través de alguna combinación de entrada.
- El ingeniero de pruebas puede controlar directamente los estados y las variables del hardware y del software.
- Los formatos de las entradas y los resultados son consistentes y estructurados.
- Las pruebas pueden especificarse, automatizarse y reproducirse convenientemente.

Capacidad de descomposición. «Controlando el ámbito de las pruebas, podemos aislar más rápidamente los problemas y llevar a cabo mejores pruebas de regresión.»

- El sistema software está construido con módulos independientes.
- Los módulos del software se pueden probar independientemente

Simplicidad. «Cuanto menos haya que probar, más rápidamente podremos probarlo.»

- Simplicidad funcional (por ejemplo, el conjunto de características es el mínimo necesario para cumplir los requisitos).
- Simplicidad estructural (por ejemplo, la arquitectura es modularizada para limitar la propagación de fallos).
- Simplicidad del código (por ejemplo, se adopta un estándar de código para facilitar la inspección y el mantenimiento).

Estabilidad. «Cuanto menos cambios, menos interrupciones a las pruebas.»

- Los cambios del software son infrecuentes.
- Los cambios del software están controlados.
- Los cambios del software no invalidan las pruebas existentes.
- El software se recupera bien de los fallos.

Facilidad de comprensión. «Cuanta más información tengamos, más inteligentes serán las pruebas.»

- El diseño se ha entendido perfectamente.
- Las dependencias entre los componentes internos, externos y compartidos se han entendido perfectamente.
- Se han comunicado los cambias del diseño.
- La documentación técnica es instantáneamente accesible.
- La documentación técnica está bien organizada.
- La documentación técnica es específica y detallada.
- La documentación técnica es exacta.

3. PRUEBA DE CAJA BLANCA.

La prueba de caja blanca, denominada a veces *prueba de caja de cristal* es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para obtener los casos de prueba. Mediante los métodos de prueba de caja blanca, el ingeniero del software puede obtener casos de prueba que **(1)** garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo; **(2)** ejerciten todas las decisiones lógicas en sus vertientes verdadera **y** falsa; **(3)** ejecuten todos los bucles en sus límites y con sus límites operacionales; y **(4)** ejerciten las estructuras internas de datos para asegurar su validez.

En este momento, se puede plantear un pregunta razonable: ¿Por qué emplear tiempo **y** energía preocupándose de (y probando) las minuciosidades lógicas cuando podríamos emplear mejor el esfuerzo asegurando que se han alcanzado los requisitos del programa? O, dicho de otra forma, ¿por qué no empleamos todas nuestras energías en la prueba de caja negra? La respuesta se encuentra en la naturaleza misma de los defectos del software:

- Los ***errores lógicos y las suposiciones incorrectas*** son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa. Los errores tienden a introducirse en nuestro trabajo cuando diseñamos e implementamos funciones, condiciones o controles que se encuentran fuera de lo normal. El procedimiento

habitual tiende a hacerse más comprensible (y bien examinado), mientras que el procesamiento de «casos especiales» tiende a caer en el caos.

- **A** menudo creemos que un camino lógico tiene **pocas** posibilidades de ejecutarse cuando, de hecho, se puede ejecutar de forma normal. El flujo lógico de un programa a veces no es nada intuitivo, lo que significa que nuestras suposiciones intuitivas sobre el flujo de control **y** los datos nos pueden llevar a tener errores de diseño que sólo se descubren cuando comienza la prueba del camino.
- ***Los errores tipográficos son aleatorios***. Cuando se traduce un programa a código fuente en un lenguaje de programación, es muy probable que se den algunos errores de escritura. Muchos serán descubiertos por los mecanismos de comprobación de sintaxis, pero otros permanecerán sin detectar hasta que comience la prueba. Es igual de probable que haya un error tipográfico en un oscuro camino lógico que en un camino principal.

4. PRUEBA DE CAJA NEGRA.

Las pruebas de caja negra, también denominada *prueba de comportamiento,* se centran en los requisitos funcionales del software. O sea, la prueba de caja negra permite al ingeniero del software obtener conjuntos de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa. La prueba de caja negra no es una alternativa a las técnicas de prueba de caja blanca. **Más** bien se trata de un enfoque complementario que intenta descubrir diferentes tipos de errores que los métodos de caja blanca.

La prueba de caja negra intenta encontrar errores de las siguientes categorías: **(1)** funciones incorrectas o ausentes, (2) errores de interfaz, **(3)** errores en estructuras de datos o en accesos a bases de datos externas, **(4)** errores de rendimiento y *(5)* errores de inicialización y de terminación.

A diferencia de la prueba de caja blanca, que se lleva a cabo previamente en el proceso de prueba, la prueba de caja negra tiende a aplicarse durante fases posteriores de la prueba (véase el Capítulo 18). Ya que la prueba de caja negra ignora intencionadamente la estructura de control, centra su atención en el campo de la información. Las pruebas se diseñan para responder a las siguientes preguntas:

- *¿Cómo se prueba la validez funcional?*
- *¿Cómo se prueba el rendimiento y el comportamiento del sistema?*
- *¿Qué clases de entrada compondrán unos buenos casos de prueba?*
- *¿Es el sistema particularmente sensible a ciertos valores de entrada?*
- *¿De qué forma están aislados los límites de una clase de datos?*
- *¿Qué volúmenes y niveles de datos tolerará el sistema?*
- *¿Qué efectos sobre la operación del sistema tendrán combinaciones específicas de datos?*

Mediante las técnicas de prueba de caja negra se obtiene un conjunto de casos de prueba que satisfacen los siguientes criterios: ***(1)*** casos de prueba que reducen, en un coeficiente que es mayor

que uno, el número de casos de prueba adicionales que se deben diseñar para alcanzar una prueba razonable y ***(2)*** casos de prueba que nos dicen algo sobre la presencia o ausencia de clases de errores en lugar de errores asociados solamente con la prueba que estamos realizando.

5. NIVELES DE PRUEBA

Las pruebas son un conjunto de actividades que se pueden planificar por adelantado y llevar a cabo sistemáticamente. Por esta razón, se debe definir en el proceso de la ingeniería del software una *plantilla* para las pruebas del software: un conjunto de pasos en los que podamos situar los métodos específicos de diseño de casos de prueba. Se han propuesto varias estrategias de prueba del software en distintos libros. Todas proporcionan al ingeniero del software una plantilla para la prueba y todas tienen las siguientes características generales:

- Las pruebas comienzan a nivel de módulo' y trabajan «hacia fuera», hacia la integración de todo el sistema basado en computadora.
- Según el momento, son apropiadas diferentes técnicas de prueba.
- La prueba la lleva a cabo el responsable del desarrollo del software y (para grandes proyectos) un grupo independiente de pruebas.
- La prueba y la depuración son actividades diferentes, pero la depuración se debe incluir en cualquier estrategia de prueba.

Una estrategia de prueba del software debe incluir pruebas de bajo nivel que verifiquen que todos los pequeños segmentos de código fuente se han implementado correctamente, así como pruebas de alto nivel que validen las principales funciones del sistema frente a los requisitos del cliente. Una estrategia debe proporcionar una guía al profesional y proporcionar un conjunto de hitos para el jefe de proyecto. Debido a que los pasos de la estrategia de prueba se dan a la vez cuando aumenta la presión de los plazos fijados, se debe poder medir el progreso y los problemas deben aparecer lo antes posible.

Verificación y validación.

La prueba del software es un elemento de un tema más amplio que, a menudo, es conocido como verificación y validación (VSrV). La *verificación* se refiere al conjunto de actividades que aseguran que el software implementa correctamente una función específica. La *validación* se refiere a un conjunto diferente de actividades que aseguran que el software construido se ajusta a los requisitos del cliente. Bohem lo define de otra forma:

- ***Verificación****:* « ¿Estamos construyendo el producto correctamente?»
- ***Validación:*** « ¿Estamos construyendo el producto correcto?»

La definición de **V&V** comprende muchas de las actividades a las que nos hemos referido como garantía de calidad del software (SQA*).

La verificación y la validación abarcan una amplia lista de actividades SQA que incluye: 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 [WAL89]. A pesar de que las actividades de prueba tienen un papel muy importante en V&V, muchas otras actividades son también necesarias.

Las pruebas constituyen el último bastión desde el que se puede evaluar la calidad y, de forma más pragmática, descubrir los errores. Pero las pruebas no deben ser vistas como una red de seguridad. Como se suele decir: «No se puede probar la calidad. Si no está ahí antes de comenzar la prueba, no estará cuando se termine.» La calidad se incorpora en el software durante el proceso de ingeniería del software. La aplicación adecuada de los métodos y de las herramientas, las revisiones técnicas formales efectivas y una sólida gestión y medición, conducen a la calidad, que se confirma durante las pruebas.

Miller relaciona la prueba del software con la garantía de calidad al establecer que «la motivación subyacente de la prueba de programas es confirmar la calidad del software con métodos que se pueden aplicar de forma económica y efectiva, tanto a grandes como a pequeños sistemas».

Organización para las pruebas del software.

En cualquier proyecto de software existe un conflicto de intereses inherente que aparece cuando comienzan las pruebas. Se pide a la gente que ha construido el software que lo pruebe. Esto parece totalmente inofensivo: después de todo, ¿quién puede conocer mejor un programa que los que lo han desarrollado? Desgraciadamente, esos mismos programadores tienen un gran interés en demostrar que el programa está libre de erro**res,** que funciona de acuerdo con las especificaciones del cliente y que estará listo de acuerdo con los plazos **y** el presupuesto. Cada uno de estos intereses se convierte en inconveniente a la hora de encontrar errores a lo largo del proceso de prueba.

Desde un punto de vista psicológico, el análisis y el diseño del software (junto con la codificación) son tareas constructivas. El ingeniero del software crea un programa de computadora, su documentación y sus estructuras de datos asociadas. Al igual que cualquier constructor, el ingeniero del software está orgulloso del edificio que acaba de construir y se enfrenta a cualquiera que intente sacarle defectos.

Cuando comienza la prueba, aparece una sutil, aun**que** firme intención de <<romper» lo que el ingeniero del software ha construido. Desde el punto de vista del constructor, la prueba se puede considerar (psicológicamente) destructiva. Por tanto, el constructor anda con cuidado, diseñando y ejecutando pruebas que demuestren que el programa funciona, en lugar de detectar errores. Desgraciadamente, los errores seguirán estando. Y si el ingeniero del software no los encuentra, el cliente **si** lo hará!

A menudo, existen ciertos malentendidos que se puedan deducir equivocadamente de la anterior discusión: **(1)** el responsable del desarrollo no debería entrar en el proceso de prueba; **(2)** el software debe ser «puesto a salvo» de extraños que puedan probarlo de forma despiadada; **(3)** los encargados de la prueba sólo aparecen en el proyecto cuando comienzan las etapas de prueba.

Todas estas frases son incorrectas.

El responsable del desarrollo del software siempre es responsable de probar las unidades individuales (módulos) del programa, asegurándose de que cada una lleva a cabo la función

para la que fue diseñada. En muchos casos, también se encargará de la prueba de integración, el paso de las pruebas que lleva a la construcción (y prueba) de la estructura total del sistema. Sólo una vez que la arquitectura del software esté completa entra en juego un grupo independiente de prueba.

El papel del grupo independiente de prueba **(GIP)** es eliminar los inherentes problemas asociados con el hecho de permitir al constructor que pruebe lo que ha construido. Una prueba independiente elimina el conflicto de intereses que, de otro modo, estaría presente.

Después de todo, al personal del equipo que forma el grupo independiente se le paga para que encuentre errores. Sin embargo, el responsable del desarrollado del software no entrega simplemente el programa al **GIP** y se desentiende. El responsable del desarrollado y el GIP trabajan estrechamente a lo largo del proyecto de software para asegurar que se realizan pruebas exhaustivas.

Mientras se realiza la prueba, el desarrollador debe estar disponible para corregir los errores que se van descubriendo.

El **GIP** es parte del equipo del proyecto de desarrollo de software en el sentido de que se ve implicado durante el proceso de especificación y sigue implicado (planificación y especificación de los procedimientos de prueba) a lo largo de un gran proyecto. Sin embargo, en muchos casos, el **GIP** informa a la organización de garantía de calidad del software, consiguiendo de este modo un grado de independencia que no sería posible si fuera una parte de la organización de desarrollo del software.

6. PRUEBAS DE UNITARIAS

La prueba de unidad centra el proceso de verificación en la menor unidad del diseño del software: el componente software o módulo.

Consideraciones sobre la prueba de unidad

Las pruebas que se dan como parte de la prueba de unidad están esquemáticamente ilustradas en la Figura 18.4. Se prueba la interfaz del módulo para asegurar que la información fluye de forma adecuada hacia y desde la unidad de programa que está siendo probada. Se examinan las estructuras de datos locales para asegurar que los datos que se mantienen temporalmente conservan su integridad durante todos los pasos de ejecución del algoritmo. Se prueban las condiciones límite para asegurar que el módulo funciona correctamente en los límites establecidos como restricciones de procesamiento.

Se ejercitan todos los caminos independientes (caminos básicos) de la estructura de control con el fin de asegurar que todas las sentencias del módulo se ejecutan por lo menos una vez. **Y,** finalmente, se prueban todos los caminos de manejo de errores.

Antes de iniciar cualquier otra prueba es preciso probar el flujo de datos de la interfaz del módulo. Si los datos no entran correctamente, todas las demás pruebas no tienen sentido. Además de las estructuras de datos locales, durante la prueba de unidad se debe comprobar (en la medida de lo posible) el impacto de los datos globales sobre el módulo.

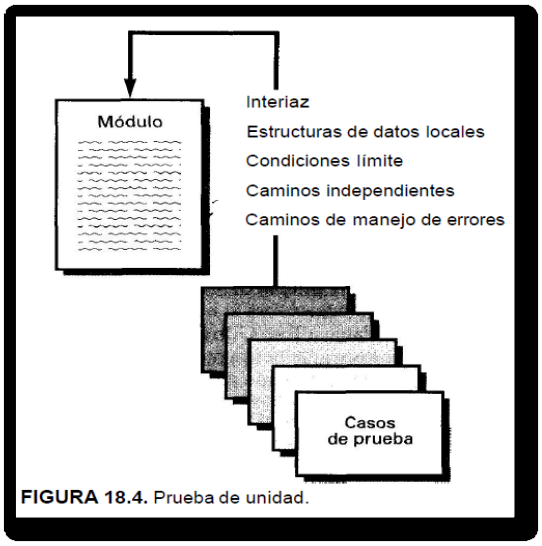


FIGURA 18.4. Prueba de unidad.

Durante la prueba de unidad, la comprobación selectiva de los caminos de ejecución es una tarea esencial. Se deben diseñar casos de prueba para detectar errores debidos a cálculos incorrectos, comparaciones incorrectas o flujos de control inapropiados. Las pruebas del camino básico y de bucles son técnicas muy efectivas para descubrir una gran cantidad de errores en los caminos.

Entre los errores más comunes en los cálculos están: (**1**) precedencia aritmética incorrecta o mal interpretada; **(2)** operaciones de modo mezcladas; **(3)** inicializaciones incorrectas; **(4)** falta de precisión; ***(5)*** incorrecta representación simbólica de una expresión. Las comparaciones y el flujo de control están fuertemente emparejadas (por ejemplo, el flujo de control cambia frecuentemente tras una comparación). Los casos de prueba deben descubrir errores como: **(1)** comparaciones ente tipos de datos distintos; (**2**) operadores lógicos o de precedencia incorrectos; **(3)** igualdad esperada cuando los errores de precisión la hacen poco probable; **(4)** variables o comparaciones incorrectas; ***(5)*** terminación de bucles inapropiada o inexistente; (**6**) fallo de salida cuando se encuentra una iteración divergente, y **(7)** variables de bucles modificadas de forma inapropiada.

Un buen diseño exige que las condiciones de error sean previstas de antemano y que se dispongan unos caminos de manejo de errores que redirijan o terminen de una forma limpia el proceso cuando se dé un error. Yourdon [YOU75] llama a este enfoque *antipurgado.*

Desgraciadamente, existe una tendencia a incorporar la manipulación de errores en el software **y** así no probarlo nunca. Como ejemplo, sirve una historia real:

Entre los errores potenciales que se deben comprobar cuando se evalúa la manipulación de errores están:

1. Descripción ininteligible del error.
2. El error señalado no se corresponde con el error encontrado.
3. La condición de error hace que intervenga el sistema antes que el mecanismo de manejo de errores.
4. El procesamiento de la condición excepcional es incorrecto.

5. La descripción del error no proporciona suficiente información para ayudar en la localización de la causa del error.

La prueba de límites es la última (y probablemente, la más importante) tarea del paso de la prueba de unidad.

El software falla frecuentemente en sus condiciones límite. Es decir, con frecuencia aparece un error cuando se procesa el elemento n-ésimo de un array n-dimensional, cuando se hace la i-ésima repetición de un bucle de i pasos o cuando se encuentran los valores máximo o mínimo permitidos. Los casos de prueba que ejerciten las estructuras de datos, el flujo de control y los valores de los datos por debajo y por encima de los máximos y los mínimos son muy apropiados para descubrir estos errores.

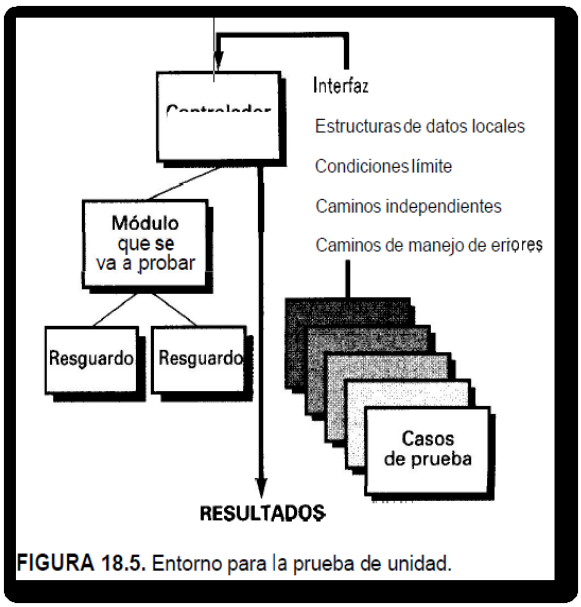


FIGURA 18.5. Entorno para la prueba de unidad.

Procedimientos de Prueba de Unidad

Debido a que un componente no es un programa independiente, se debe desarrollar para cada prueba de unidad un software que controle ylo resguarde. En la Figura 18.5 se ilustra el entorno para la prueba de unidad. En la mayoría de las aplicaciones, un *controlador* no es más que un «programa principal» que acepta los datos del caso de prueba, pasa estos datos al módulo (a ser probado) e imprime los resultados importantes. Los resguardos sirven para reemplazar módulos que están subordinados (llamados por) el componente que hay que probar. Un resguardo o un «subprograma simulado» usa la interfaz del módulo subordinado, lleva a cabo una mínima manipulación de datos, imprime una verificación de entrada y devuelve control al módulo de prueba que lo invocó.

Los controladores y los resguardos son una sobrecarga de trabajo. Es decir, ambos son software que debe desarrollarse (normalmente no se aplica un diseño formal) pero que no se entrega con el producto de software final. Si los controladores y resguardos son sencillos, el trabajo adicional es relativamente pequeño.

Desgraciadamente, muchos componentes no pueden tener una adecuada prueba unitaria con un «sencillo» software adicional. En tales casos, la prueba completa se pospone hasta que se llegue al paso de prueba de integración (donde también se usan controladores o resguardos). descubrir más fácilmente.

La prueba de unidad se simplifica cuando se diseña un módulo con un alto grado de cohesión. Cuando un módulo sólo realiza una función, se reduce el número de casos de prueba y los errores se pueden predecir y descubrir más fácilmente.

7. PRUEBA DE INTEGRACION.

Un neófito del mundo del software podría, una vez que se les ha hecho la prueba de unidad a todos los módulos, cuestionar de forma aparentemente legítima lo siguiente: «Si todos funcionan bien por separado, ¿por **qué** dudar de que funcionen todos juntos?» Por supuesto, el problema es «ponerlos juntos» (interacción). Los datos se pueden perder en una interfaz; un módulo puede tener un efecto adverso e inadvertido sobre otro; las subfunciones, cuando se combinan, pueden no producir la función principal deseada; la imprecisión aceptada individualmente puede crecer hasta niveles inaceptables; y las estructuras de datos globales pueden presentar problemas. Desgraciadamente, la lista sigue y sigue.

La prueba de integración es una técnica sistemática para construir la estructura del programa mientras que, al mismo tiempo, se llevan a cabo pruebas para detectar errores asociados con la interacción. El objetivo es coger los módulos probados mediante la prueba de unidad y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño.

A menudo hay una tendencia a intentar una integración no incremental; es decir, a construir el programa mediante un enfoque de *(Cbig bang».* Se combinan todos los módulos por anticipado. Se prueba todo el programa en conjunto. ¡Normalmente se llega al caos! Se encuentran un gran conjunto de errores. La corrección se hace difícil, puesto que es complicado aislar las causas al tener delante el programa entero en toda su extensión.

Una vez que se corrigen esos errores aparecen otros nuevos y el proceso continúa en lo que parece ser un ciclo sin fin.

La integración incremental es la antítesis del enfoque del «big bang». El programa se construye y se prueba en pequeños segmentos en los que los errores son más fáciles de aislar y de corregir, es más probable que se puedan probar completamente las interfaces y se puede aplicar un enfoque de prueba sistemática. En las siguientes secciones se tratan varias estrategias de integración incremental diferentes.

Integración descendente

La prueba de integración descendente es un planteamiento incremental a la construcción de la estructura de programas. Se integran los módulos moviéndose hacia abajo por la jerarquía de control, comenzando por el módulo de control principal (programa principal). Los módulos subordinados (subordinados de cualquier modo) al módulo de control principal se van

incorporando en la estructura, bien de forma *primero-en-profundidad,* o bien de forma *primero-en-anchura.*

Como se muestra en la Figura 18.6, la integración primero-en-profundidad integra todos los módulos de un camino de control principal de la estructura. La selección del camino principal es, de alguna manera, arbitraria y dependerá de las características específicas de la aplicación. Por ejemplo, si se elige el camino de la izquierda, se integrarán primero los módulos M **1,** M2 y M5. **A** continuación, se integrará M8 o M6 (si es necesario para un funcionamiento adecuado de M2). Acto seguido se construyen los caminos de control central y derecho. La integración primero-en-anchura incorpora todos los módulos directamente subordinados a cada nivel, moviéndose por la estructura de forma horizontal. Según la figura, los primeros módulos que se integran son **M2, M3** y **M4.** A continuación, sigue el siguiente nivel de control, **M5, M6**, etc.

El proceso de integración se realiza en una serie de cinco pasos:

1. Se usa el módulo de control principal como controlador de la prueba, disponiendo de resguardos para todos los módulos directamente subordinados al módulo de control principal.
2. Dependiendo del enfoque de integración elegido (es decir, primero-en-profundidad o primero-en-anchura) se van sustituyendo uno a uno los resguardos subordinados por los módulos reales.
3. Se llevan a cabo pruebas cada vez que se integra un nuevo módulo.
4. Tras terminar cada conjunto de pruebas, se reemplaza otro resguardo con el módulo real.
5. Se hace la prueba de regresión para asegurarse de que no se han introducido errores nuevos.

El proceso continúa desde el paso 2 hasta que se haya construido la estructura del programa entero.

La estrategia de integración descendente verifica los puntos de decisión o de control principales al principio del proceso de prueba. En una estructura de programa bien fabricada, la toma de decisiones se da en los niveles superiores de la jerarquía y, por tanto, se encuentran antes. Si existen problemas generales de control, es esencial reconocerlos cuanto antes. Si se selecciona la integración primero en profundidad, se puede ir implementando y demostrando las funciones completas del software.

Por ejemplo, considere una estructura clásica de transacción (Capítulo **14)** en la que se requiere una compleja serie de entradas interactivas, obtenidas y validadas por medio de un camino de entrada. Ese camino de entrada puede ser integrado en forma descendente. Así, se puede demostrar todo el proceso de entradas (para posteriores operaciones de transacción) antes de que se integren otros elementos de la estructura. La demostración anticipada de las posibilidades funcionales es un generador de confianza tanto para el desabollador como para el cliente.

La estrategia descendente parece relativamente fácil, pero, en la práctica, pueden surgir algunos problemas logísticos. El más común de estos problemas se da cuando se requiere un proceso de los niveles más bajos de la jerarquía para poder probar adecuadamente los niveles

superiores. Al principio de la prueba descendente, los módulos de bajo nivel se reemplazan por ***resguar**dos;* por tanto, no pueden fluir datos significativos hacia arriba por la estructura del programa. El responsable de la prueba tiene tres opciones: **(1)** retrasar muchas de las pruebas hasta que los resguardos sean reemplazados por los módulos reales**; (2)** desarrollar resguardos que realicen funciones limitadas que simulen los módulos reales; o ***(3)*** integrar el software desde el fondo de la jerarquía hacia arriba.

El primer enfoque (retrasar pruebas hasta remplazar los resguardos por los módulos reales) hace que perdamos cierto control sobre la correspondencia de ciertas pruebas específicas con la incorporación de determinados módulos. Esto puede dificultar la determinación de las causas del error y tiende a violar la naturaleza altamente restrictiva del enfoque descendente. El segundo enfoque es factible pero puede llevar a un significativo incremento del esfuerzo a medida que los resguardos se hagan más complejos. El tercer enfoque, denominado prueba ascendente, se estudia en la siguiente sección.

Integración ascendente

La prueba de la integración ascendente, como su nombre indica, empieza la construcción y la prueba con los *módulos atómicos* (es decir, módulos de los niveles más bajos de la estructura del programa). Dado que los módulos se integran de abajo hacia arriba, el proceso requerido de los módulos subordinados siempre está disponible y se elimina la necesidad de resguardos.

Se puede implementar una estrategia de integración ascendente mediante los siguientes pasos:

1. Se combinan los módulos de bajo nivel en ***grupos*** (a veces denominados construcciones) que realicen una subfunción específica del software.

2. Se escribe un controlador (un programa de control de la prueba) para coordinar la entrada y la salida de los casos de prueba.

3. Se prueba el grupo.

4. Se eliminan los controladores y ***se*** combinan los grupos moviéndose hacia arriba por la estructura del programa.

La integración sigue el esquema ilustrado en la Figura 18.7. Se combinan los módulos para formar los grupos **1, 2** y **3.** Cada uno de los grupos se somete a prueba mediante un controlador (mostrado como un bloque punteado). Los módulos de los grupos **1 y 2** son subordinados de M. Los controladores **D**, y **D**, se eliminan y los grupos interaccionan directamente con **M**. De forma similar, se elimina el controlador D, del grupo **3** antes de la integración con el módulo **M**. Tanto **M,** como **M,** se integrarán finalmente con el módulo **M,** y así sucesivamente.

A medida que la integración progresa hacia arriba, disminuye la necesidad de controladores de prueba diferentes. De hecho, si los dos niveles superiores de la estructura del programa se integran de forma descendente, se puede reducir sustancialmente el número de controladores y se simplifica enormemente la integración de grupos.

8. PRUEBAS DE IMPLANTACIÓN.

El objetivo de las pruebas de implantación es comprobar el funcionamiento correcto del sistema integrando el hardware y software en el entorno de operación, y permitir al usuario

que, desde el punto de vista de operación, realice la aceptación del sistema una vez instalado en su entorno real y en base al cumplimiento de los requisitos no funcionales especificados.

Las pruebas de implantación están orientadas a una revisión de requisitos no funcionales del producto en un **entorno de preproducción** de calidad. ¿Por qué de calidad? Un entorno de estas características que sea muy distinto al de producción o cuyos resultados no sean extrapolables al mismo provocará en muchos casos conclusiones erróneas y en otro generará muchas dudas sobre el cumplimiento o no de las especificaciones no funcionales.

De esta forma, se considerarán pruebas de implantación por ejemplo, las pruebas de carga o stress, las relacionadas con la seguridad del sistema, etc…

Si un producto no supera las pruebas de implantación establecidas (para ello se requiere la definición de umbrales de cada métrica cualitativa o cuantitativa que se decida medir) no debería **pasarse a producción** aunque supere las pruebas de aceptación (que se podrían desarrollar en paralelo a las pruebas de sistema), de las cuales también se podría obtener un feedback interesante para las pruebas de implantación, ya que las acciones de los usuarios en las aplicaciones son imprevisibles y el valor de su testing (adicional a la propia validación de requisitos) puede ser muy importante.

9. PRUEBAS DE ACEPTACIÓN.

El objetivo de las pruebas de aceptación es validar que un sistema cumple con el funcionamiento esperado y permitir al usuario de dicho sistema que determine su aceptación, desde el punto de vista de su funcionalidad y rendimiento.

Las pruebas de aceptación son definidas por el usuario del sistema y preparadas por el equipo de desarrollo, aunque la ejecución y aprobación final corresponden al usuario.

La validación del sistema se consigue mediante la realización de pruebas de caja negra que demuestran la conformidad con los requisitos y que se recogen en el plan de pruebas, el cual define las verificaciones a realizar y los casos de prueba asociados. Dicho plan está diseñado para asegurar que se satisfacen todos los requisitos funcionales especificados por el usuario teniendo en cuenta también los requisitos no funcionales relacionados con el rendimiento, seguridad de acceso al sistema, a los datos y procesos, así como a los distintos recursos del sistema.

"Una PA tiene como propósito demostrar al cliente el cumplimiento de un requisito del software"

- Describe un escenario (secuencia de pasos) de ejecución o uso del sistema desde la perspectiva del cliente
- Puede estar asociada a requisitos funcionales o no funcionales
- Un requisito tiene una o más PAs asociadas
- La Pas cubren desde escenarios típicos/frecuentes hasta los más excepcionales.

10. PRUEBA DE REGRESIÓN.

Cada vez que se añade un nuevo módulo como parte de una prueba de integración, el software cambia. Se establecen nuevos caminos de flujo de datos, pueden ocurrir nuevas E/S y se invoca una nueva lógica de control.

Estos cambios pueden causar problemas con funciones que antes trabajaban perfectamente. En el contexto de una estrategia de prueba de integración, la *prueba de regresión* es volver a ejecutar un subconjunto de pruebas que se han llevado a cabo anteriormente para asegurarse de que los cambios no han propagado efectos colaterales no deseados.

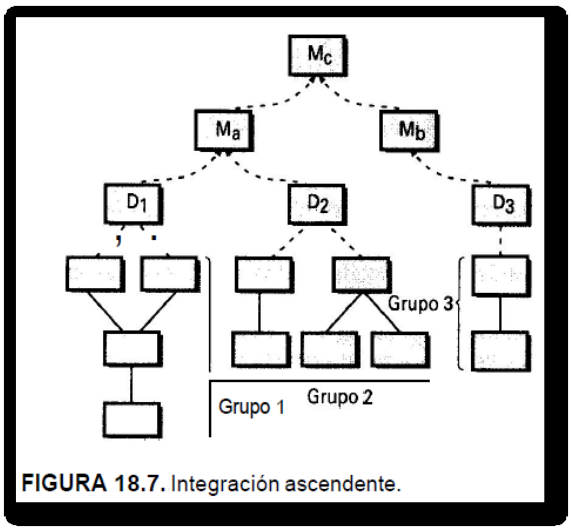


FIGURA 18.7. Integración ascendente.

En un contexto más amplio, las pruebas con éxito (de cualquier tipo) dan como resultado el descubrimiento de errores, y los errores hay que corregirlos. Cuando se corrige el software, se cambia algún aspecto de la configuración del software (el programa, su documentación o los datos que lo soportan). La prueba de regresión es la actividad que ayuda a asegurar que los cambios (debidos a las pruebas o por otros motivos) no introducen un comportamiento no deseado o errores adicionales.

La prueba de regresión se puede hacer manualmente, volviendo a realizar un subconjunto de todos los casos de prueba o utilizando herramientas automáticas de reproducción de captura. Las *herramientas de reproducción de captura* permiten al ingeniero del software capturar casos de prueba y los resultados para la subsiguiente reproducción y comparación. El conjunto de pruebas de regresión (el subconjunto de pruebas a realizar) contiene tres clases diferentes de casos de prueba: una muestra representativa de pruebas que ejercite todas las funciones del software; pruebas adicionales que se centran en las funciones del software que se van a ver probablemente afectadas por el cambio; pruebas que se centran en los componentes del software que han cambiado.

A medida que progresa la prueba de integración, **el** número de pruebas de regresión puede crecer demasiado. Por tanto, el conjunto de pruebas de regresión debería diseñarse para incluir sólo aquellas pruebas que traten una o más clases de errores en cada una de las funciones

principales del programa. No es práctico ni eficiente volver a ejecutar cada prueba de cada función del programa después de un cambio.

Prueba de humo

La prueba de humo es un método de prueba de integración que es comúnmente utilizada cuando se ha desarrollado un producto software «empaquetado».

Es diseñado como un mecanismo para proyectos **crí**ticos por tiempo, permitiendo que el equipo de software valore su proyecto sobre una base sólida. En esencia, la prueba de humo comprende las siguientes actividades:

1. Los componentes software que han sido traducidos a código se integran en una «construcción». Una construcción incluye ficheros de datos, librerías, módulos reutilizables y componentes de ingeniería que se requieren para implementar una o más funciones del producto.
2. Se diseña una serie de pruebas para descubrir errores que impiden a la construcción realizar su función adecuadamente. El objetivo será descubrir errores «bloqueantes» que tengan la mayor probabilidad de impedir al proyecto de software el cumplimiento de su planificación.
3. Es habitual en la prueba de humo que la construcción **se** integre con otras construcciones y que se aplica una prueba de humo al producto completo (en su forma actual). La integración puede hacerse bien de forma descendente *(top-down)* o ascendente *(bottom-up)*.

La frecuencia continua de la prueba completa del producto puede sorprender a algunos lectores. En cualquier caso, las frecuentes pruebas dan a gestores y profesionales una valoración realista de la evolución de las pruebas de integración. McConnell describe la prueba de humo de la siguiente forma:

La prueba de humo ejercita el sistema entero de principio a fin. No ha de ser exhaustiva, pero será capaz de descubrir importantes problemas. La prueba de humo será suficiente si verificamos de forma completa la construcción y podemos asumir que es suficientemente estable para ser probado con más profundidad.

La prueba de humo facilita una serie de beneficios cuando se aplica sobre proyectos de ingeniería del software complejos y críticos por su duración:

- ***Se minimizan los riesgos de integración***. Dado que las pruebas de humo son realizadas frecuentemente, incompatibilidades y otros errores bloqueantes son descubiertos rápidamente, por eso se reduce la posibilidad de impactos importantes en la planificación por errores sin descubrir.
- ***Se perfecciona la calidad del producto final.*** Dado que la prueba de humo es un método orientado a la construcción (integración), es probable que descubra errores funcionales, además de defectos de diseño a nivel de componente y de arquitectura. Si estos defectos se corrigen rápidamente, el resultado será un producto de gran calidad.
- ***Se simplifican el diagnóstico y la corrección de errores***. Al igual que todos los enfoques de prueba de integración, es probable que los errores sin descubrir durante la prueba de humo se asocien a «nuevos incrementos de software» - e s t o es, el software

que se acaba de añadir a la construcción es una posible causa de un error que se acaba de descubrir.

- ***El progreso es fácil de observar.*** Cada día que pasa, se integra más software y se demuestra que funciona. Esto mejora la moral del equipo y da una indicación a los gestores del progreso que se está realizando.

Comentarios sobre la prueba de integración

Ha habido muchos estudios sobre las ventajas y desventajas de la prueba de integración ascendente frente a la descendente. En general, las ventajas de una estrategia tienden a convertirse en desventajas para la otra estrategia. La principal desventaja del enfoque descendente es la necesidad de resguardos y las dificultades de prueba que pueden estar asociados con ellos. Los problemas asociados con los resguardos pueden quedar compensados por la ventaja de poder probar de antemano las principales funciones de control. La principal desventaja de la integración ascendente es que «el programa como entidad no existe hasta que se ha añadido el último módulo». Este inconveniente se resuelve con la mayor facilidad de diseño de casos de prueba y con la falta de resguardos.

La selección de una estrategia de integración depende de las características del software y, a veces, de la planificación del proyecto. En general, el mejor compromiso puede ser un enfoque combinado (a veces denominado *prueba sandwich)* que use la descendente para los niveles superiores de la estructura del programa, junto con la ascendente para los niveles subordinados.

A medida que progresa la prueba de integración, el responsable de las pruebas debe identificar los módulos críticos. Un módulo crítico es aquel que tiene una o más de las siguientes características: **(1)** está dirigido a varios requisitos del software; (**2**) tiene un mayor nivel de control (está relativamente alto en la estructura del programa); (**3**) es complejo o propenso a errores (se puede usar la complejidad ciclomática como indicador); o **(4)** tiene unos requisitos de rendimiento muy definidos. Los módulos críticos deben probarse lo antes posible. Además, las pruebas de regresión se deben centrar en el funcionamiento de los módulos críticos.

11. GESTION DE PROCESOS DE PRUEBA.

Es la gestión enfocada a verificar que toda la implementación del sistema (código), no tenga errores, según el paradigma utilizado (estructurado, orientado a objetos, etc) en las fases anteriores, se utilizan distintos tipos de pruebas como son: pruebas de caja negra, de caja blanca, prueba de ensayos y errores, etc.

Toda esta fase necesita ser gestionada, ya que está sujeto a un cronograma, personal, registro de pruebas, etc., y necesita ser administrado.

Existen herramientas que prestan su asistencia en la planificación, desarrollo y control de las pruebas. Las herramientas de gestión de pruebas se utilizan para controlar y coordinar las pruebas del software por todos y cada uno de los pasos principales de las pruebas. Las herramientas de esta categoría gestionan y coordinan las pruebas de regresiones, efectúan comparaciones que determinan las diferencias entre la salida real y la esperada y realizan pruebas por lotes de programas con interfaces hombre máquina interactivas. Además de las funciones indicadas anteriormente, muchas herramientas de gestión de pruebas sirven también

como controladores de pruebas genéricos. Un controlador de pruebas lee uno o más casos de prueba de algún archivo de pruebas, aplica formato a los datos de prueba para que se ajusten a las necesidades del software que se está probando, e invoca entonces al software que es preciso probar.

12. Planificación y Seguimiento.

Los planes de pruebas deberían incluir una descripción de ios elementos que hay que probar, la agenda de pruebas, los procedimientos para gestionar el proceso de pruebas, los requerimientos hardware y software, y cualquier problema de pruebas que probablemente pueda surgir.

La verificación y validación es un proceso caro. Para algunos sistemas, tales como los sistemas de tiempo real con restricciones no funcionales complejas, más de la mitad del presupuesto para el desarrollo del sistema puede invertirse en V & V. Es necesaria una planificación cuidadosa para obtener el máximo provecho de las inspecciones y pruebas y controlar los costes del proceso de verificación y validación.

Debería comenzarse la planificación de la verificación y validación del sistema en etapas tempranas del proceso de desarrollo. El modelo de proceso de desarrollo del software mostrado en la Figura 22.3 se denomina a veces modelo V. Es una instanciación del modelo genérico en cascada y muestra que los planes de pruebas deberían derivarse a partir de la especificación y diseño del sistema.

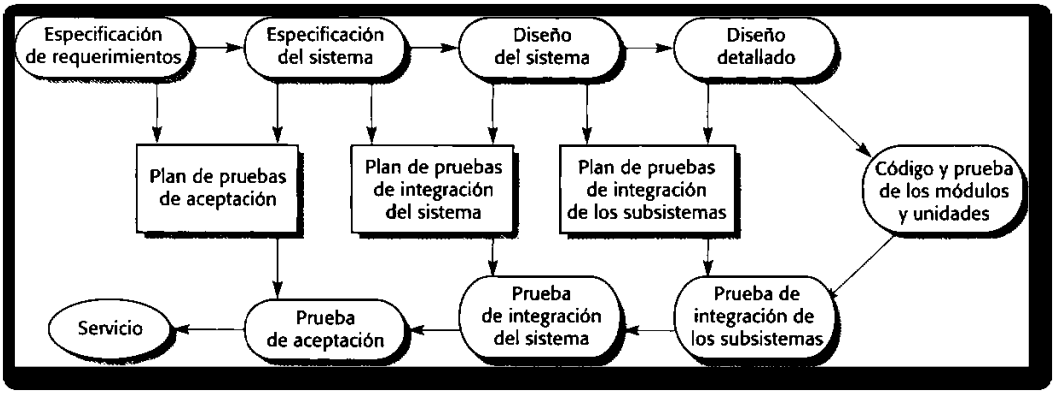


Figura 22.3 Planes de pruebas como un enlace entre las pruebas y el desarrollo.

Este modelo también divide la V & V del sistema en varias etapas. Cada etapa está conducida por las pruebas que tienen que definirse para comprobar la conformidad del programa con su diseño y especificación.

Como parte del proceso de planificación de V & V, habría que decidir un equilibrio entre las aproximaciones estáticas y dinámicas de la verificación y validación, y pensar en estándares y procedimientos para las inspecciones y pruebas del software, establecer listas de comprobación para conducir las inspecciones de programas (véase la Sección 22.3) y definir el plan de pruebas del software.

El relativo esfuerzo destinado a las inspecciones y las pruebas depende del tipo de sistema a desarrollar y de los expertos de la organización en la inspección de programas. Como regla

general, cuanto más crítico sea el sistema, debería dedicarse más esfuerzo a las técnicas de verificación estáticas.

La planificación de las pruebas está relacionada con el establecimiento de estándares para el proceso de las pruebas, no sólo con la descripción de los productos de las pruebas. Los planes de pruebas, además de ayudar a los gestores a asignar recursos y estimar el calendario de las pruebas, son de utilidad para los ingenieros del software implicados en el diseño y la realización de las pruebas del sistema. Estos ayudan al personal técnico a obtener una panorámica general de las pruebas del sistema y ubicar su propio trabajo en este contexto. Una buena descripción de los planes de pruebas y su relación con los planes de calidad más generales se proporciona en Frewin y Hatton (Frewin y Hatton, 1986). Humphrey (Humphrey, 1989) y Kit (Kit, 1995) también incluyen estudios sobre la planificación de las pruebas.

El proceso de Prueba

Una descripción de las principales fases del proceso de prueba. Éstas podrían describirse como se hizo anteriormente en este capítulo.

Trazabilidad de Requerimientos

Los usuarios son los más interesados en que el sistema satisfaga sus requerimientos y las pruebas deberían planificarse para que todos los requerimientos se prueben individualmente.

Elementos Probados

Deberían especificarse los elementos del proceso del software que tienen que ser probados.

Calendario de Pruebas

Un calendario de todas las pruebas y la asignación de recursos para este calendario se enlaza, obviamente, con la agenda general del desarrollo del proyecto.

Procedimiento de Registro de las Pruebas

No es suficiente ejecutar simplemente las pruebas; los resultados de las pruebas deben ser registrados sistemáticamente. Debe ser posible auditar d proceso de pruebas para comprobar que se ha llevado a cabo correctamente.

Requerimientos de Hardware y Software

Esta sección deberla determinar las herramientas software requeridas y la utilización estimada del hardware.

Restricciones

En esta sección deberían anticiparse las restricciones que afectan al proceso de Pruebas como la escases de Personal.

FIGURA 22.4 – La Estructura de un Plan de Pruebas.

Los principales componentes de un plan de pruebas para un sistema grande y complejo se muestran en la Figura 22.4. Además de determinar el calendario y procedimientos de las pruebas, el plan de pruebas define los recursos hardware y software que se requieren. Este es útil para los gestores del sistema que son los responsables de asegurar que estos recursos están disponibles para el equipo de pruebas, Los planes de pruebas normalmente deberían incluir cantidades significativas de contingencias para que los desajustes en la

implementación y el diseño puedan solucionarse y el personal pueda ser reasignado a otras actividades.

Para sistemas más pequeños, se puede utilizar un plan de pruebas menos formal, pero sigue siendo necesario un documento formal para soportar la planificación del proceso de pruebas. Para algunos procesos ágiles como la programación extrema, las pruebas son inseparables del desarrollo. Al igual que otras actividades de planificación, la planificación de las pruebas también es incremental. En XP, el cliente es el último responsable de decidir cuánto esfuerzo debería dedicarse a la prueba del sistema.

Los planes de pruebas no son documentos estáticos, sino que evolucionan durante el proceso de desarrollo. Los planes de pruebas cambian debido a retrasos en otras etapas del proceso de desarrollo. Si parte de un sistema está incompleto, el sistema no puede probarse como un todo. Entonces tiene que revisarse el plan de pruebas para volver a desplegar y a asignar a los encargados de las pruebas a alguna otra actividad, y recuperarlos cuando el software vuelva a estar disponible.

13. HERRAMIENTAS DE GESTIÓN Y MANEJO DE PRUEBAS

- **kSETT**

Tabla 7.12: kSETT.

Clase de herramienta	Gestión de pruebas. Gestión y seguimiento de casos de pruebas.
Organización	Software SETT Corporation. http://www.softsett.com/
Descripción	kSETT soluciona el problema de mantener un gran volumen de pruebas, seguimiento y reporte de resultados de pruebas manuales y automáticas para todos los miembros del equipo y gerencia. kSETT proporciona una solución Web detallada para problemas cotidianos de crear, manejar, seguir y ejecutar casos de prueba. Usando un estándar HTML para la interfaz y los informes en línea el kSETT proporcionan una intefaz amigable al usuario.
Plataforma	Solución Hosted

- **rth**

Tabla 7.13: rth.

Clase de herramienta	Gestión de pruebas. Gestión de pruebas/requerimientos - seguimiento de defectos (Freeware)
Organización	RTH project. https://sourceforge.net/projects/rth/
Descripción	Requerimientos: almacena registros o archivos de requerimientos bajo control de versiones (upload documentos o inserción de registros). Links al requerimiento, a subrequerimientos o link a requerimientos a probar. Cerrar o abrir requerimientos. Notificaciones por e-mail cuando un requerimiento cambia. Vista en tabla o vista en árbol de requerimientos. **Pruebas**: almacenamiento de casos de prueba o documentación de soporte bajo control de versiones. Capacidad de atar pruebas a los requisitos y de definir cobertura. Asignar las pruebas a usuarios y al progreso de seguimiento, de la creación o de la automatización de las pruebas. Resultado de las pruebas: atar todos los resultados de la prueba a Release, Build, y Test Set. Almacenar los planes de prueba bajo control de la versión. **Defectos:** atar defectos a otros defectos. Asignar un defecto a un Desarrollador o a un Probador. Adicionar notas a un defecto. Mostrar el historial de bugs. Notificaciones por e-mail de cambios en los defectos. **rth** es diseñado para ser simple y correr en cualquier sistema. La disposición y la configuración son simples y funciona en los servidores de Linux y de Windows.
Plataforma	Windows y Linux.

- **QaTraq**

Tabla 7.14: QaTraq.

Clase de herramienta	Gestión de pruebas. Herramienta de gestión de casos de prueba. (Freeware)
Organización	testmanagement.com http://www.testmanagement.com/
Descripción	QaTraq es un sistema de gestión de casos de prueba, con capacidad de realizar reportes y seguimiento de los resultados de las pruebas. QaTraq está disponible como Open Source Software.
Plataforma	Windows, Linux y Solaris.

- **TestDirector**

Tabla 7.15: TestDirector.

Clase de herramienta	Gestión de pruebas. Herramienta de gestión de pruebas automatizada.
Organización	Mercury Interactive. http://www.mercuryinteractive.com/
Descripción	Gestiona y automatiza el proceso entero de QA.
Plataforma	MS Windows.

- **QADirector®#circledR;**

Tabla: QADirector®#circledR;.

Clase de herramienta	Herramienta de gestión de pruebas.
Organización	Compuware Corporation. www.compuware.com/qacenter
Descripción	QADirector®#circledR; es una herramienta de gestión de procesos de pruebas base-Windows que es parte de QACenterTM de Compuware de la familia de productos de pruebas para aplicaciones. Provee a los encargados de la aplicación y del sistema, a los desarrolladores y de los grupos de trabajo de QA un solo punto del control para manejar todas las fases de pruebas. QADirector integra la gestión de pruebas con las pruebas automatizadas para proporcionar un framework que maneja el proceso de pruebas entero, planeación y diseño, ejecución y análisis. QADirector también permite hacer el mejor uso de los artefactos existentes en las pruebas (planes de prueba, casos, scripts), de las metodologías y de las herramientas de prueba de aplicaciones.
Plataforma	Windows.

- **T-Plan Professional**

Tabla 7.17: T-Plan Professional.

Clase de herramienta	Herramienta de gestión de pruebas.
Organización	T-Plan. www.t-plan.co.uk

Descripción	T-Plan permite que se maneje cada aspecto del proceso de pruebas, proporcionando un acercamiento consistente y estructurado. Proporcionando orden, estructura y visibilidad a través del ciclo de vida de desarrollo, desde la planeación a ejecución.
Plataforma	Windows.

15. CASO PRÁCTICO

El diseño de pruebas para el software o para otros productos de ingeniería puede requerir tanto esfuerzo como el propio diseño inicial del producto. Sin embargo, los ingenieros del software, por razones que ya hemos tratado, a menudo tratan las pruebas como algo sin importancia, desarrollando casos de prueba que «parezcan adecuados», pero que tienen poca garantía de ser completos.

Recordando el objetivo de las pruebas, debemos diseñar pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible.

Cualquier producto de ingeniería (y de muchos otros campos) puede probarse de una de estas dos formas: (1) 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;(2) 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. El primer enfoque de prueba se denomina prueba de caja negra y el segundo, prueba de caja blanca.

Cuando se considera el software de computadora, la prueba de caja negra se refiere a las pruebas que se llevan a cabo sobre la interfaz del software. O sea, los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce un resultado correcto, así como que la integridad de la información externa (por ejemplo, archivos de datos) se mantiene. Una prueba de caja negra examina algunos aspectos del modelo fundamental del sistema sin tener mucho en cuenta la estructura lógica interna del software.

La prueba de caja blanca del software se basa en el minucioso examen de los detalles procedimentales. Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles. Se puede examinar el «estado del programa» en varios puntos para determinar si el estado real coincide con el esperado o mencionado.

A primera vista parecería que una prueba de caja blanca muy profunda nos lleva a tener «programas cien por cien correctos». Todo lo que tenemos que hacer es definir todos los

caminos lógicos, desarrollar casos de prueba que los ejerciten y evaluar los resultados, es decir, generar casos de prueba que ejerciten exhaustivamente la lógica del programa.

Desgraciadamente, la prueba exhaustiva presenta ciertos problemas logísticos. Incluso para pequeños programas, el número de caminos lógicos posibles puede ser enorme. Por ejemplo, considere un programa de 100 líneas de código en lenguaje C. Después de la declaración de algunos datos básicos, el programa contiene dos bucles que se ejecutan de 1 a 20 veces cada uno, dependiendo de las condiciones especificadas en la entrada. Dentro del bucle interior, se necesitan cuatro construcciones **if-then else.**

Para poner de manifiesto el significado de este número, supongamos que hemos desarrollado un procesador de pruebas mágico («mágico» porque no existe tal procesador) para hacer una prueba exhaustiva. El procesador puede desarrollar un caso de prueba, ejecutarlo y evaluar los resultados en un milisegundo. Trabajando las 24 horas del día, 365 días al año, el procesador trabajaría durante 3170 años para probar el programa. Esto irremediablemente causaría estragos en la mayoría de los planes de desarrollo. La prueba exhaustiva es imposible para los grandes sistemas software.

La prueba de caja blanca, sin embargo, no se debe desechar como impracticable. Se pueden elegir y ejercitar una serie de caminos lógicos importantes.

Se pueden comprobar las estructuras de datos más importantes para verificar su validez. Se pueden combinar los atributos de la prueba de caja blanca así como los de caja negra, para llegar a un método que valide la interfaz del software y asegure selectivamente que el funcionamiento interno del software es correcto.

TEMA VII: Implementación de la Aplicación

Documentación del código. Automatización de la instalación del sistema desarrollado. Manuales de documentación técnica de Instalación. Manuales del usuario. Manual de Administración y Mantenimiento del Sistema.

Documentación del Código

a.- Documentación Externa

La **documentación en un proyecto de software** consiste en información tanto dentro del código fuente como fuera de él generalmente en forma de documentos separados.

En los proyectos formales de desarrollo de software la mayoría de la documentación es de éste último tipo. En esta categoría están no sólo los más visibles para los usuarios tales como los manuales de operación y de instalación del software, sino también los documentos de diseño utilizados durante el desarrollo del software: copia de los requerimientos, copia del algoritmo empleado así como de las alternativas consideradas, diagramas de flujo, copia de los documentos que se utilizan para el diseño de las entradas y salidas visuales o impresas, copia de los estándares de desarrollo, listado más actual del código fuente, documentos relacionados con las modificaciones hechas al proyecto, así como notas importantes de diseño usadas por los desarrolladores durante el diseño y la implementación, entre otras.

Por lo general estos documentos se agrupan en una carpeta especial y es empleada solamente para fines de uso interno. Puede existir un documento de diseño más detallado a nivel de código que describe decisiones de diseño a nivel de unidad de programación (una clase, una rutina o un programa), las alternativas que fueron consideradas durante la codificación y las razones por las que se seleccionaron las aproximaciones elegidas. Algunas veces toda esta información está contenida en un documento formal, pero a menudo está en una carpeta de uso personal del programador. En ambos casos, esta información de nivel tan detallado está separada del código. Pero a veces sucede que existe solamente en el código mismo.

b.- Documentación Interna

A diferencia de la documentación externa, la documentación interna se encuentra dentro del código mismo y es la más detallada de las dos. Puesto que es la más cercana al código, la documentación interna es la que se suele mantener más actualizada y correcta a medida que el código se modifica.

La documentación interna es muy importante puesto que facilita grandemente la lectura y comprensión del código, tanto para el propio programador como para todos los que necesiten leerlo, y es especialmente útil en las fases de prueba y mantenimiento de los programas.

Todos los lenguajes de programación ofrecen secuencias de caracteres que permiten la inclusión de comentarios en el código. En este artículo usaremos la simbología de C estándar y C++ (/* */ y //), para los comentarios. Si usas otro lenguaje, utiliza su simbología correspondiente.

c.- Listas de Chequeo para el Código Autodocumentado

Las siguientes preguntas, agrupadas por temas, son de gran ayuda para determinar si un código en particular es auto explicativo. Todas ellas representan aspectos importantes del buen estilo de programación.

Clases

- ¿Presenta la interfaz de la clase una abstracción consistente?
- ¿Tiene la clase un nombre adecuado, y este nombre describe su propósito principal?
- ¿La interfaz de la clase hace obvio cómo se le debería usar?
- ¿Es la interfaz de la clase lo suficientemente abstracta, tal que no tienes que pensar sobre cómo se implementan sus servicios? ¿Puedes tratar a la clase como una caja negra?

Rutinas

- ¿El nombre de cada rutina describe exactamente lo que hace?
- ¿Desempeña cada rutina una tarea bien definida?
- ¿Los componentes de cada rutina que se beneficiarían de colocarse en sus propias rutinas, se han puesto en sus propias rutinas?
- ¿Es obvia y clara la interfaz de cada rutina?

Nombres de Datos

- ¿Los nombres de tipos de datos son lo suficientemente descriptivos, tal que ayudan a documentar las declaraciones de datos?
- ¿Se han nombrado adecuadamente las variables?
- ¿Son las variables usadas únicamente para el propósito para el cual fueron nombradas?
- ¿Se le han asignado nombres más informativos a los contadores de ciclos, en lugar de i, j y k?
- ¿Se utilizan constantes nombradas en lugar de constantes literales?
- ¿La convención de nombres de identificadores empleada distingue entre los nombres de tipos de datos, los tipos enumerados, las constantes nombradas, las variables locales, las variables de clases, y las variables globales?
- ¿Se utilizan tipos enumerados bien nombrados en lugar de corrimientos de bits o variables booleanas?

Organización de Datos

- ¿Se utilizan variables extras para efectos de claridad cuando es necesario?
- ¿Están las referencias a variables lo más cercanas a su declaración?
- ¿Son los tipos de datos lo suficientemente simples, tal que minimizan la complejidad?
- ¿Son accedidos los datos a través de rutinas de acceso abstractas (tipos de datos abstractos)?

Control

- ¿Es claro el flujo de ejecución a través del código?
- ¿Están las sentencias de código relacionadas agrupadas o cercanas entre sí?
- ¿Se han empacado los grupos independientes de sentencias en sus propias rutinas?
- ¿Se ha colocado el caso más común de entre dos posibilidades en la parte if en lugar de en la parte else de una estructura condicional simple?
- ¿Son las estructuras de control lo suficientemente simples, tal que minimizan la complejidad?
- ¿Cada ciclo desempeña una y solo una función, como debería hacerlo una rutina bien diseñada?
- ¿Se ha minimizado el anidamiento (de ciclos y rutinas por ejemplo)?
- ¿Han sido simplificadas las expresiones booleanas usando variables booleanas adicionales, funciones booleanas y tablas de decisión?

Esquema

- ¿El esquema del programa muestra su estructura lógica?

Diseño

- ¿Es el código directo, y evita los hackeos habilidosos?
- ¿Se han ocultado al máximo los detalles de la implementación?
- ¿Está el programa escrito en términos del dominio del problema tanto como es posible en lugar de en términos de las ciencias computacionales o de las estructuras del lenguaje de programación?

AUTOMATIZACIÓN DE LA INSTALACIÓN DEL SISTEMA DESARROLLADO.

Un proyecto de software que necesita implantar un software en ambientes de homologación y producción o que necesita distribuirlo para un gran número de máquinas de usuario.

Problema

El proceso de implantación de un software en ambientes de homologación, de producción y en estaciones de clientes generalmente es hecho en forma manual, con apoyo de guías de instrucciones ofrecidas por el equipo de desarrollo a los equipos de soporte. Errores en la secuencia de tareas y en la ejecución de todos los procedimientos pueden atrasar la implantación de software críticos o generar errores para los usuarios del software.

Objetivos

- Es preciso garantizar que una instalación sea hecha sin errores.
- Una implantación y distribución manual de un software aumentan la probabilidad de riesgo de ocurrencia de errores humanos.

Solución

- Utilice una herramienta para automatizar el proceso de distribución, implantación e instalación, que se pueda ejecutar de manera simple y con un único comando.

- La distribución e instalación de un software son tareas repetitivas y deben ser hechas frecuentemente para disponibilidad el software en ambientes de Test, de Homologación, de producción y en máquinas de clientes. La realización manual de una distribución y de una instalación puede generar errores de trabajo.
- La automatización reduce el tiempo que el equipo de producción y soporte técnico gasta en tareas repetitivas y aumenta el tiempo de respuesta de todos los involucrados.

Herramientas

Existen actualmente en el mercado una enorme cantidad de herramientas para realizar automatización de Instalación:

NSIS: es un programa de Software Libre que permite la creación de instaladores para ambientes Windows

Ant: es una herramienta de build de Software Libre especializada en construir proyectos Java. Ant utiliza archivos XML para expresar las instrucciones de ejecución de build. Ella posee una gran variedad de tareas que permiten la realización de actividades como generación de pruebas unitarias, publicación de información, entre otras.

Capistrano: es una Herramientas para Automatización de Instalación, que permite automatizar tareas en uno o más servidores remotos. Ejecuta comandos en forma paralela en todas las máquinas destino, y proporciona un mecanismo para hacer rollback de los cambios a través de múltiples máquinas.

MANUALES DE DOCUMENTACION TECNICA DE INSTALACION.

Una instalación exitosa es una condición necesaria para el funcionamiento de cualquier software. Mientras más complejo sea el software, es decir, entre otras características, mientras más archivos contenga, mientras mayor la dispersión de los archivos y mientras mayor sea la interdependencia con otros software, mayor es el riesgo de alguna falla durante la instalación. Si la instalación falla aunque sea solo parcialmente, el fin que persigue la instalación posiblemente no podrá ser alcanzado. Por esa razón, sobre todo en casos de software complejo, el desarrollo de un proceso de instalación confiable y seguro es una parte fundamental del desarrollo del software.

Pasos a Tener en Cuenta para la instalación y documentación de la misma:

a- Verificación de la compatibilidad: Se debe comprobar si se cumplen los requisitos para la instalación en cuanto a hardware y software. A veces es necesario desinstalar versiones antiguas del mismo software.

b- Verificación de la integridad: Se verifica que el paquete de software es el original, esto se hace para evitar la instalación de programas maliciosos.

c- Creación de los directorios requeridos: Para mantener el orden en el directorio cada sistema operativo puede tener un estandar para la instalación de ciertos archivos en ciertos directorios. Ver por ejemplo Linux Standard Base.

d- Creación de los usuarios requeridos: Para deslindar responsabilidades y tareas se pueden o deben usar diferentes usuarios para diferentes paquetes de software.

e- Concesión de los derechos requeridos: Para ordenar el sistema y limitar daños en caso necesario, se le conceden a los usuarios solo el mínimo necesario de derechos.

f- Copia, desempaque y decompresión de los archivos desde el paquete de software: Para ahorrar Ancho de banda y tiempo en la transmisión por internet o espacio de Disco duro, los paquetes vienen empacados y comprimidos.

g- Archivos principales, sean de fuente o binarios.

h- Archivos de datos, por ejemplo datos, imágenes, modelos, documentos XML-Dokumente, etc.

Documentación

a- *Archivos de configuración.*

b- *Bibliotecas.*

c- *Enlaces duros o enlaces simbólicos a otros archivos.*

d- *Compilación y enlace con las bibliotecas requeridas:* En algunos casos no se puede evitar el complicado paso de la compilación y enlace que a su vez tiene severos requerimientos de software al sistema. El enlace con bibliotecas requeridas puede ser un problema si en su instalación no se acataron los estándares establecidos.

e- *Configuración:* Por medio de archivos de configuración se le da a conocer al software con que parámetros debe trabajar. Por ejemplo, los nombres de las personas que pueden usar el software, como verificar su clave de ingreso , la ruta donde se encuentran los archivos con datos o la dirección de nuestro proveedor de correo electrónico. Para sistemas complejos s e debe desarrollar el Software Configuration Management.

f- *Definir las variables de entorno requeridas*: Algunos comportamientos del software solo pueden ser determinados por medio de estas variables. Esto es parte de la configuración, aunque es más dinámica.

g- *Registro ante el dueño de la marca:* Para el Software comercial a veces el desarrollador de software exige el registro de la instalación si se desea su servicio.

Si un sistema de gestión de paquetes realiza la instalación, entonces este se ocupa de llevar la contabilidad de las versiones, (des-) instalaciones y cambios en los paquetes de software del sistema.

En caso de que bibliotecas hayan sido cambiadas por la instalación, es necesario arrancar el sistema operativo o el software nuevamente para hacer efectivos los cambios en todos los programas.

MANUALES DEL USUARIO

Expone los procesos que el usuario puede realizar con el sistema implantado. Para lograr esto, es necesario que se detallen todas y cada una de las características que tienen los programas y la forma de acceder e introducir información. Permite a los usuarios conocer el detalle de qué actividades ellos deberán desarrollar para la consecución de los objetivos del sistema. Reúne la información, normas y documentación necesaria para que el usuario conozca y utilice adecuadamente la aplicación desarrollada.

Objetivos

- Que el usuario conozca cómo preparar los datos de entrada.
- Que el usuario aprenda a obtener los resultados y los datos de salida.
- Servir como manual de aprendizaje.
- Servir como manual de referencia.
- Definir las funciones que debe realizar el usuario.
- Informar al usuario de la respuesta a cada mensaje de error.
- Pasos a seguir para definir como desarrollar el manual de usuario.
- Identificar los usuarios del sistema: personal que se relacionará con el sistema.
- Definir los diferentes tipos de usuarios: se presentan los diferentes tipos de usuarios que usarían el sistema. Ejemplo: usuarios directos, indirectos.
- Definir los módulos en que cada usuario participará: Se describen los módulos o procesos que se ejecutarán por cada usuario en forma narrativa breve y clara.

Importancia del Manual De Usuario

- El Manual de Usuario facilita el conocimiento de:
- Los documentos a los que se puede dar entrada por computadora.
- Los formatos de los documentos.
- Las operaciones que utiliza de entrada y salida de los datos.
- El orden del tratamiento de la computadora con los datos introducidos.
- El momento en que se debe solicitar una operación deseada.
- Los resultados de las operaciones realizadas a partir de los datos introducidos.
- Al elaborar el Manual de Usuario, hay que tener en cuenta a quién va dirigido es decir, el manual puede ser manejado desde el director de la empresa hasta el introductor de datos. Por consiguiente, debe redactarse de forma clara y sencilla para que lo entienda cualquier tipo de usuario.

Contenido

a) Diagrama general del sistema.

Muestra en forma condensada el flujo general de la información y de las actividades que se realizan en el sistema. Proporciona una visión general del sistema. Representar los diagramas utilizando para ello diagramas de bloques.

b) Diagrama particular detallado.

Presentar gráficamente todos los pasos que se efectúen dentro del departamento usuario a quien está dirigido este manual. Deben especificarse los archivos de entrada, salida, los resultados, revisiones y procesos manuales.

c) Explicación Genérica de Las Fases Del Sistema.

En este punto se explica en forma específica y detallada todas las operaciones que aparecen representadas en forma gráfica en el diagrama particular. Se analizan cada una de las fases señalando:

c.1) El proceso principal que se desarrolla.

c.2) La entrada de la información.

c.3) La obtención de un resultado parcial.

c.4) El envío de información a otra dependencia.

d) Instalación del Sistema.

La instalación del sistema proporciona detalles completos sobre la forma de instalar el sistema en un ambiente *particular.*

e) Iniciación al Uso Del Sistema

En este punto se explica cómo iniciarse en el sistema y cómo se pueden utilizar sus cualidades comunes. Esta documentación debe decir al usuario cómo salir de un problema cuando las cosas funcionan mal.

f) Manual de Referencia

Es el documento definitivo de cara al usuario y debe ser completo. Describe con detalle las cualidades del sistema y su uso, los informes de error generados y las situaciones en que surgen esos errores.

Dependiendo del sistema, los documentos al usuario se pueden proporcionar por separado o reunidos en varios volúmenes. Los sistemas de ayuda en línea evitan que el usuario pierda tiempo en consultas manuales.

g) Caducidad de Documento Fuente y Destino Final

Como el usuario trabajará con documentos fuentes, éstos podrán tener un período de retención y un destino especificado.

MANUAL DE ADMINISTRACIÓN Y MANTENIMIENTO DEL SISTEMA.

El conjunto de herramientas que se pueden emplear en el mantenimiento de sistemas es tan amplio o más que la diversidad misma de los sistemas susceptibles de ser mantenidos. Incluso, de forma recursiva, buena parte de las herramientas utilizadas en el mantenimiento requieren a su vez de mantenimiento para continuar siendo operativas. Por citar un ejemplo curioso, la herramienta más adecuada para identificar un problema en un analizador lógico podría ser, según el caso, otro analizador lógico.

De este modo, incluso restringiéndonos a los sistemas informáticos, la diversidad de las herramientas y la complejidad de las más especializadas hacen que intentar describirlas o simplemente presentarlas todas sea una tarea imposible. Desde elementos tan simples como unas pinzas, un destornillador o un soldador para reparar circuitos o cableados hasta una cámara blanca para la fabricación de una nueva versión de un circuito integrado, pasando por un sistema JTAG compuesto de hardware específico, software y un ordenador, todo son herramientas validas para mantener sistemas electrónicos digitales.

En el caso de los sistemas informáticos –sobre todo, aunque no solamente, en los ordenadores– se da además la circunstancia de que tanto el sistema a mantener como la herramienta pueden ser hardware, software –y recordemos que el software no tiene existencia física– o una mezcla de ambos, lo que da todavía mayor diversidad a las herramientas.

Como ejercicio que ejemplifica esta última frase es interesante intentar encontrar –si acaso existen–, dentro del mantenimiento de ordenadores:

- herramientas hardware para mantener hardware,
- herramientas software para mantener software,
- herramientas hardware para mantener software,
- herramientas software para mantener hardware,
- . . .

El presente tema pues se dedica a dar una visión global de las herramientas que se aplican al mantenimiento, centrándose en el mantenimiento de sistemas informáticos y en las herramientas de diagnostico de averías – la fase comúnmente más difícil del mantenimiento correctivo. Se presentan algunas clasificaciones, ejemplos y técnicas. Al final se incluye un

apartado acerca de las herramientas informáticas –aplicaciones– que ayudan a la gestión del mantenimiento en general.

1.2. Según concepción.

Como se vio en el tema uno, durante las fases de estudio y producción del ciclo de vida de las instalaciones se pueden desarrollar herramientas que se vayan a utilizar en su mantenimiento. Sin embargo hemos citado antes una serie de herramientas para el mantenimiento que no es necesario diseñar ni fabricar puesto que son de uso general o validas para un amplio conjunto de sistemas. Atendiendo a si las herramientas se deben crear –concebir– ad hoc para un sistema o son de aplicación más genérica, se pueden clasificar como específicas o generales.

1.2.1. Específicas.

Son aquellas herramientas que se diseñan para una instalación o sistema. Su uso es privativo de y suele estar limitado al sistema para el cual se han diseñado, siendo por ello muy potentes y eficaces en su mantenimiento.

Como contrapartida, su coste es elevado y su uso acaba cuando termina el ciclo de vida del sistema al cual se aplican. Están pues justificadas únicamente para sistemas de elevado coste que no puedan ser mantenidos de forma comparable con herramientas genéricas.

1.2.2. Generales.

Son herramientas cuyo campo de aplicación incluye todo tipo de sistemas o una clase susficientemente grande de estos. Se diseñan y fabrican pensando en una aplicabilidad general y no en un sistema en particular. La totalidad de las herramientas que se usaran como ejemplos en las siguientes secciones pertenecen a este tipo.

Hoy en día, debido al extendido uso de los sistemas empotrados y a la aplicación de estándares en muchos ámbitos de la industria, las herramientas específicas se diseñan utilizando componentes mas genéricos y software de personalización y configuración, reduciendo sensiblemente su coste y su tiempo de desarrollo. La introducción de estándares en el diseño de los sistemas, muchos de los cuales tienen que ver con su mantenibilidad, permite que las herramientas puedan ser diseñadas conociendo a priori las especificaciones físicas, eléctricas, de protocolo, etcétera de los sistemas a mantener. De este modo los sistemas empotrados que constituirán las herramientas pueden incluir los componentes hardware necesario para comunicarse mediante estos estándares y el software que interprete los datos obtenidos sin que el sistema a mantener haya sido fabricado todavía. Finalmente el carácter específico lo dará una capa de configuración para adaptarse al sistema y posiblemente la interfaz de usuario. Un ejemplo común de este tipo de herramientas se puede encontrar en los sistemas de diagnostico de averías en vehículos.

1.3. Según nivel de aplicación.

Si atendemos a la extensión del sistema sobre el que se pueden aplicar las herramientas, estas se pueden clasificar en instrumentos, equipos y sistemas.

1.3.1. Instrumentos.

Se trata de herramientas de extensión reducida, que tienen acceso local a un solo elemento de la instalación que se está manteniendo y de la cual pueden tomar un número limitado de muestras o medidas. Ejemplos simples del ámbito del mantenimiento de sistemas electrónicos serían los polímetros, osciloscopios, etcétera.

1.3.2. Equipos.

Son conjuntos de herramientas o herramientas múltiples que permiten tomar muestras de un sistema completo o de varios sistemas suficientemente próximos. Las medidas así tomadas deben estar sincronizadas o coordinadas de alguna manera. Ejemplos serían un analizador mixto –osciloscopio y analizador lógico en un mismo bastidor de sistema, el uso de software específico y un osciloscopio para depurar un sistema de comunicación RS232 .

1.3.3. Sistemas.

Un sistema está formado por un conjunto de equipos interconectados y sincronizados entre sı que permite monitorizar una instalación completa. Ejemplos de sistemas los tenemos en los conjuntos de instrumentos conectados por un bus GPIB, en las aplicaciones que monitorizan coordinadamente diversos ordenadores en red.

TEMA VIII: Fundamentos de la Ingeniería

Concepto

¿Qué es Ingeniería Inversa?

La **ingeniería inversa** se define como el **proceso** de construir especificaciones de un mayor nivel de abstracción partiendo del código fuente de un sistema software o cualquier otro producto (se puede utilizar como punto de partida cualquier otro elemento de diseño, etc.).

Esto también aplica a dispositivos electrónicos, donde la ingeniería inversa sirve para recuperar diagramas esquemáticos, levantar circuitos impresos, hasta el recupero del código fuente de microcontroladores o microprocesadores, para con esta información, desarrollar nuevos dispositivos, productos equivalentes o ampliar prestaciones a los mismos.

Estas especificaciones pueden volverse a usar para construir una nueva implementación del sistema utilizando, por ejemplo, técnicas de ingeniería directa.

La ingeniería inversa nos permite obtener la base de fabricación, programación, instalación o concepción de cualquier objeto, software o proceso.

Ingeniería inversa: análisis de un sistema para identificar sus componentes y las relaciones entre ellos, así como para crear representaciones del sistema en otra forma o en un nivel de abstracción más elevado.

Objetivos

El objetivo de la **ingeniería inversa** es obtener información o un diseño a partir de un producto accesible al público, con el fin de determinar de qué está hecho, qué lo hace funcionar y cómo fue fabricado.

Hoy en día (principios del siglo XXI), los productos más comúnmente sometidos a ingeniería inversa son los programas de computadoras y los componentes electrónicos, pero, en realidad, cualquier producto puede ser objeto de un análisis de Ingeniería Inversa.

El **método** se denomina así porque avanza en dirección opuesta a las tareas habituales de ingeniería, que consisten en utilizar datos técnicos para elaborar un producto determinado. En general, si el producto u otro material que fue sometido a la ingeniería inversa fue obtenido en forma apropiada, entonces el proceso es legítimo y legal. De la misma forma, pueden fabricarse y distribuirse, legalmente, los productos genéricos creados a partir de la información obtenida de la ingeniería inversa, como es el caso de algunos proyectos de Software libre ampliamente conocidos.

La **ingeniería inversa** es un método de resolución. Aplicar ingeniería inversa a algo supone profundizar en el estudio de su funcionamiento, hasta el punto de que podamos llegar a entender, modificar y mejorar dicho modo de funcionamiento.

Pero este término no sólo se aplica al software, sino que también se considera ingeniería inversa el estudio de todo tipo de elementos (por ejemplo, equipos electrónicos, microcontroladores, u objeto fabril de cualquier clase). Diríamos, más bien, que la ingeniería inversa antecede al nacimiento del software, tratándose de una posibilidad a disposición de las empresas para la producción de bienes mediante copiado desde el mismo surgimiento de la ingeniería.

En el caso concreto del **software**, se conoce por ingeniería inversa a la actividad que se ocupa de descubrir **cómo funciona** un programa, función o característica de cuyo código fuente no se dispone, hasta el punto de poder modificar ese código o generar código propio que cumpla las mismas funciones. La gran mayoría del software de pago incluye en su licencia una prohibición expresa de aplicar ingeniería inversa a su código, con el intento de evitar que se pueda modificar su código y que así los usuarios tengan que pagar si quieren usarlo.

La ingeniería inversa nace en el transcurso de la Segunda Guerra Mundial, cuando los ejércitos enemigos incautaban insumos de guerra como aviones u otra maquinaria de guerra para mejorar las suyas mediante un exhaustivo análisis.

Beneficios:

La aplicación de la **Ingeniería inversa** nunca cambia la funcionalidad del software sino que permite obtener productos que indican como se ha construido el mismo. Los beneficios son:

- ✓ *Reducir la complejidad del sistema*: Al intentar comprender el software se facilita su mantenimiento y la complejidad existente disminuye.
- ✓ *Generar diferentes alternativas*: del punto de partida del proceso, principalmente código fuente, se generan representaciones gráficas lo que facilita su comprensión.
- ✓ *Recuperar y/o actualizar la información perdida (cambios que no se documentaron en su momento):* en la evolución del sistema se realizan cambios que no se suele actualizar en las representaciones de nivel de abstracción más alto, para lo cual se utiliza la recuperación de diseño.
- ✓ *Detectar efectos laterales*: Los cambios que se pueden realizar en un sistema puede conducirnos a que surjan efectos no deseados, esta serie de anomalías pueden ser detectado por la ingeniería inversa.
- ✓ *Facilitar la reutilización*: por medio de la ingeniería inversa se pueden detectar componentes de posible reutilización de sistemas existentes, pudiendo aumentar la productividad, reducir los costes y los riesgos de mantenimiento.

Aplicaciones:

La finalidad de la ingeniería inversa es la de desentrañar los misterios y secretos de los sistemas en uso a partir del código. Para ello se emplean una serie de herramientas que extraen información de los datos, procedimientos y arquitectura del sistema existente.

En muchos casos se ha asociado a la Ingeniería Inversa, como el método para obtener copias, muchas veces sin licencia, de objetos o piezas que se hallen en el mercado. Este sería uno de sus múltiples usos, pero en realidad, la aplicación de estas tecnologías va mucho más allá.

El principal cometido de la ingeniería inversa (en lo que a captación de formas se refiere), es el de obtener un archivo CAD 3D del objeto o pieza de muestra, para con él poder:

- Fabricar de nuevo dicha pieza (única o en serie).
- Analizar o estudiar dicha pieza, para su mejora, o para el diseño de una pieza de características similares.
- Crear un archivo informatizado.
- Obtener un CAD de una pieza creada a mano o que haya sufrido alteraciones. (Prototipos).
- Obtener modelos informatizados del terreno. (Arquitectura, geología, minería, etc.)
- Utilizar el CAD de una pieza validada, para realizar informes dimensionales de producciones en serie.
- Diseñar piezas u objetos que deban encajar, alojarse, fijarse, etc. en las piezas digitalizadas.
- Obtener datos informatizados de escenarios o piezas implicadas en accidentes o crímenes.
- Fabricar piezas relacionadas con la pieza de muestra, que de algún modo guarden similitud con la original. (Prótesis médicas o dentales de cualquier tipo).
- Fabricar envases, blisters, protectores, etc., para el "packaging" de los objetos escaneados.

Comprender el Funcionamiento del Programa

La primera actividad real de la ingeniería inversa comienza con un intento de comprender y extraer después abstracciones de procedimientos representados por el código fuente. Para comprender las abstracciones de procedimientos, se analiza el código en distintos niveles de abstracción: sistema, programa, componente, configuración y sentencia.

La funcionalidad general de todo sistema de aplicaciones deberá ser algo perfectamente comprendido antes de que tenga lugar un trabajo de ingeniería inversa mas detallado. Esto es lo que establece un contexto para un análisis posterior y se proporciona ideas generales acerca de los problemas de interoperabilidad entre aplicaciones dentro del sistema. Cada uno de los programas de que consta el sistema de aplicaciones representara una abstracción funcional con un elevado nivel de detalle. También se creara un diagrama de bloques como representación de la iteración entre estas abstracciones funcionales. Cada uno de los componentes representa una sub-función y representa una abstracción definida de procedimientos. En cada componente se crea una narrativa de procedimientos. En algunas situaciones ya existen especificación de sistema, programa y componente.

Cuando ocurre tal cosa, se revisan las especificaciones para evaluar si se ajustan al código existente. Todo se complica cuando se considera el código que reside en el interior del componente. El ingeniero busca las secciones de código que representan las configuraciones

genéricas de procedimiento. En casi todos los componentes existe una sección de código que prepara los datos para su procesamiento (dentro del componente), una sección diferente de código que efectua el procesamiento y otra sección de código que prepara los resultados del procesamiento para exportarlos de ese componente. En el interior de cada una de estas secciones se encuentran configuraciones más pequeñas.

Para los sistemas grandes la ingeniería inversa suele efectuarse mediante el uso de un enfoque semi-automatizado. Las herramientas CASE se utilizan para "analizar" la semántica del código existente. La salida de este proceso se pasa entonces a una herramienta de reestructuración y de ingeniería directa que completarían el proceso de reingeniería.

Identificación y recopilación de componentes funcionales.

1. Identificación y recopilación de los componentes funcionales del sistema.
    - rutinas, variables, constantes, tipos de datos, TAD, objetos, llamadas a funciones, etc.
    - Muy subjetiva e intuitiva.

Algunas ideas:

- cada componente suele ocupar un módulo, o bien aparecen próximos unos a otros.
- las series de componentes funcionales suelen aparecer junto a muchos comentarios.
- los identificadores de los componentes funcionales suelen constar de muchos caracteres.

Algunos problemas:

- sinonimia
- polisemia
- comentarios no actualizados

2. Asignar significado a los componentes "sustanciales" anteriores. Se recorren las sentencias del componente, para confirmar su calidad de componente funcional.
    - Análisis estático (creación de DFDs...)
    - Análisis dinámico (ejecución del programa)
    - Detectar y registrar bucles infinitos, código inalcanzable, etc.

    Y que no se corrigen en esta fase, sólo se documentan, se corrigen después, en la fase de reingeniería

Tipos de Ingeniería Inversa

La ingeniería inversa puede ser de varios tipos:

Ingeniería inversa de datos: Se aplica sobre algún código de bases de datos (aplicación, código SQL, etc) para obtener los modelos relacionales o sobre el modelo relacional para obtener el diagrama entidad relación.

- **Ingeniería inversa de lógica o de proceso**: Cuando la ingeniería inversa se aplica sobre código de un programa para averiguar su lógica o sobre cualquier documento de diseño para obtener documentos de análisis o de requisitos.
- **Ingeniería inversa de interfaces de usuario**: Se aplica con objeto de mantener la lógica interna del programa para obtener los modelos y especificaciones que sirvieron de base para la construcción de la misma, con objeto de tomarlas como punto de partida en procesos de ingeniería directa que permitan modificar dicha interfaz.
- **Ingeniería inversa de protocolos de comunicación**: Independientemente del medio físico, se analiza el comportamiento de la comunicación, analizan tramas y documentan protocolos, para la implementación de nuevos sistemas, replicas o interfaces.

Ingeniería inversa de procesos:

La primera actividad real de la **ingeniería inversa** comienza con un intento de comprender y posteriormente, extraer las abstracciones de procedimientos representadas por el código fuente. Para comprender las abstracciones de procedimientos, se analiza el código en distintos niveles de abstracción: sistema, programa, componente, configuración y sentencia.

Antes de iniciar el trabajo de ingeniería inversa detallado debe comprenderse totalmente la funcionalidad general de todo el sistema de aplicaciones sobre el que se está operando. Esto es lo que establece un contexto para un análisis posterior, y proporciona ideas generales acerca de los problemas de interoperabilidad entre aplicaciones dentro del sistema.

Cada uno de los programas de que consta el sistema de aplicaciones representará una abstracción funcional con un elevado nivel de detalle, creándose un diagrama de bloques como representación de la iteración entre estas abstracciones funcionales. Cada uno de los componentes de estos diagramas efectúa una subfunción, y representa una abstracción definida de procedimientos. En cada componente se crea una narrativa de procesamientos. En algunas situaciones ya existen especificaciones de sistema, programa y componente. Cuando ocurre tal cosa, se revisan las especificaciones para apreciar si se ajustan al código existente, descartando posibles errores.

Todo se complica cuando se considera el código que reside en el interior del componente. El ingeniero busca las secciones del código que representan las configuraciones genéricas de procedimientos. En casi todos los componentes, existe **una sección de código que prepara los datos para su procesamiento** (dentro del componente), una sección

diferente de **código que efectúa el procesamiento** y otra sección de **código que prepara los resultados del procesamiento** para exportarlos de ese componente. En el interior de cada una de estas secciones, se encuentran configuraciones más pequeñas. Por ejemplo, suele producirse una verificación de los datos y una comprobación de los límites dentro de la sección de código que prepara los datos para su procesamiento.

Para los sistemas grandes, la ingeniería inversa suele efectuarse mediante el uso de un enfoque semi-automatizado.

Las herramientas CASE se utilizan para "analizar" la semántica del código existente. La salida de este proceso se pasa entonces a unas herramientas de reestructuración y de ingeniería directa que completarán el proceso de reingeniería.

Cuándo aplicar ingeniería inversa de procesos.

Cuando la ingeniería inversa se aplica sobre código de un programa para averiguar su lógica o sobre cualquier documento de diseño para obtener documentos de análisis o de requisitos se habla de ingeniería inversa de procesos.

Habitualmente, este tipo de ingeniería inversa se usa para:

- Entender mejor la aplicación y regenerar el código.
- Migrar la aplicación a un nuevo sistema operativo.
- Generar/completar la documentación.
- Comprobar que el código cumple las especificaciones de diseño.

La información extraída son las especificaciones de diseño: se crean modelos de flujo de control, diagramas de diseño, documentos de especificación de diseño, etc. y pudiendo tomar estas especificaciones como nuevo punto de partida para aplicar ingeniería inversa y obtener información a mayor nivel de abstracción.

La información extraída son las especificaciones de diseño: se crean modelos de flujo de control, diagramas de diseño, documentos de especificación de diseño, etc. y pudiendo tomar estas especificaciones como nuevo punto de partida para aplicar ingeniería inversa y obtener información a mayor nivel de abstracción.

¿Cómo hacemos la ingeniería inversa de procesos?

A la hora de realizar ingeniería inversa de procesos se suelen seguir los siguientes pasos:

1. Buscamos el programa principal.
2. Ignoramos inicializaciones de variables, etc.
3. Inspeccionamos la primera rutina llamada y la examinamos si es importante.
4. Inspeccionamos las rutinas llamadas por la primera rutina del programa principal, y examinamos aquellas que nos parecen importantes.
5. Repetimos los pasos 3-4 a lo largo del resto del software.
6. Recopilamos esas rutinas "importantes", que se llaman componentes funcionales.

7. Asignamos significado a cada componente funcional, esto es (a) explicamos qué hace cada componente funcional en el conjunto del sistema y (b) explicamos qué hace el sistema a partir de los diferentes componentes funcionales.

A la hora de encontrar los componentes funcionales hay que tener en cuenta que los módulos suelen estar ocupados por componentes funcionales. Además, suele haber componentes funcionales cerca de grandes zonas de comentarios y los identificadores de los componentes funcionales suelen ser largos y formados por palabras entendibles.

Una vez encontrados los posibles componentes funcionales, conviene repasar la lista teniendo en cuenta que un componente es funcional cuando Un componente es funcional cuando su ausencia impide seriamente el funcionamiento de la aplicación, diculta la legibilidad del código, impide la comprensión de todo o de otro componente funcional o cuando hace caer a niveles muy bajos la calidad, habilidad, mantenibilidad, etc.

Vamos a ver cómo a partir de un *código java* cómo se puede realizar Ingeniería Inversa de Procesos.

Tenemos dos clases *(Persona y Trabajador*)

```
class Persona {
  protected String nombre;
  protected int edad;
  protected int seguroSocial;
  protected String licenciaConducir;
  public Persona(String nom, int ed, int seg, String lic) {
  set(nom, ed); seguroSocial = seg; licenciaConducir = lic; }
  public Persona() {
  Persona(null, 0, 0, null); }
  public int setNombre(String nom) {
  nombre = nom; return 1; }
  public int setEdad(int ed) {
  edad = ed; return 1; }
  public void set(String nom, int ed) {
  setNombre(nom); setEdad(ed); }
  public void set(int ed, String nom) {
  setNombre(nom); setEdad(ed); }
  }
  class Trabajador extends Persona {
  private String empresa;
  private int salario;
```

```
public Trabajador(String emp, int sal) {
empresa = emp; salario = sal; }
public Trabajador() {
this(null,0); }
public int setEmpresa String emp) {
empresa = emp; return 1; }
public int setSalario(int sal) {
salario = sal; return 1; }
public void set(String emp, int sal) {
setEmpresa(emp); setSalario(sal); }
public void set(int sal, String emp) {
setEmpresa(emp); setSalario(sal); }
}
```

Si realizamos Ingeniería Inversa, el diagrama UML sería el siguiente:

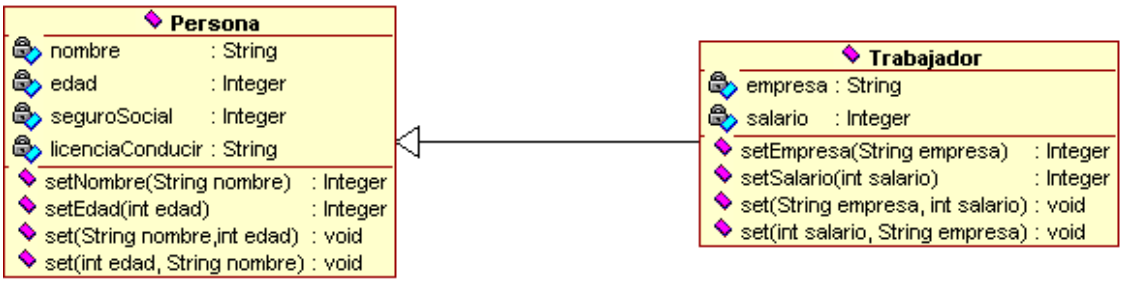
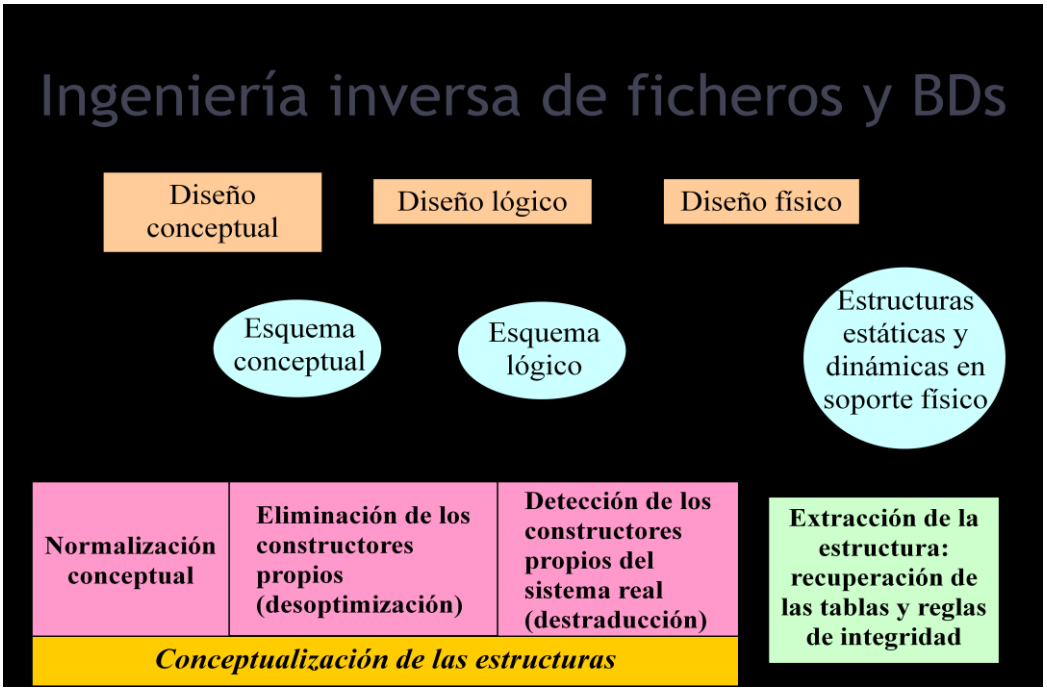


Figura 1. Diagrama de clase generado a partir del código Java

Ingeniería inversa de ficheros y BDs



Ingeniería inversa de ficheros - Extracción de la estructura

- Considerar cada fichero como una posible tabla, y cada campo del fichero como un campo de la tabla.
- Determinar un conjunto de campos que puedan ser clave primaria de sus respectivos ficheros (buscar ID, #).
- Determinar las claves ajenas.
- Determinar los ficheros que no pueden tratarse como tablas (aquellos sin claves).
- Buscar generalizaciones:
  - grandes grupos de claves ajenas.
  - valores repetidos de atributos en una tabla.
  - datos con valores mutuamente excluyentes.
- Encontrar asociaciones.

INGENIERÍA INVERSA EN BASE DE DATOS

Es el proceso de construir especificaciones de un mayor nivel de abstracción partiendo del código fuente de un sistema software o cualquier otro producto (se puede utilizar como punto de partida cualquier otro elemento de diseño, etc.).

Estas especificaciones pueden volver ser utilizadas para construir una nueva implementación del sistema utilizando, por ejemplo, técnicas de ingeniería directa. Estas técnicas que permite la obtención de una representación conceptual de un esquema de base de datos a partir de su codificación.

Aplicaciones:

Sus aplicaciones son múltiples. Re-documentar, reconstruir y/o actualizar documentación perdida o inexistente de bases de datos, servir como pivote en un proceso de migración de datos, y ayudar en la exploración y extracción de datos en bases poco documentadas.

Beneficios de Ingeniería Inversa

• Reducir la complejidad del sistema: al intentar comprender el software se facilita su mantenimiento y la complejidad existente disminuye.

• Generar diferentes alternativas: del punto de partida del proceso, principalmente código fuente, se generan representaciones gráficas lo que facilita su comprensión.

• Recuperar y/o actualizar la información perdida (cambios que no se documentaron en su momento): en la evolución del sistema se realizan cambios que no se suele actualizar en las representaciones de nivel de abstracción más alto, para lo cual se utiliza la recuperación de diseño.

INGENIERÍA INVERSA Y REINGENIERÍA DE INTERFACES DE USUARIO

Adaptar aplicaciones a las necesidades de los usuarios, respetando su lógica anterior:

1. Recopilación de documentación, manuales de usuario, etc.

2. Entrevistas a distintos grupos de usuarios, y observación de sus métodos de trabajo.

3. Uso del sistema por el propio equipo de mantenimiento. *Se puede modificar el código para, p.ej., introducir contadores.*

4. Reconstrucción y re-documentación de la interfaz.

Reconstrucción de programas

- A partir de los productos de ingeniería inversa se construye el programa mediante técnicas de ingeniería directa.
- Reestructuración de datos
  - eliminar sinonimias y polisemias
- Reestructuración de procesos
  - transformar el código no estructurado en código estructurado
  - *en un diagrama de flujo estructurado, es posible hacer transformaciones sucesivas hasta que su complejidad ciclomática se iguale a 1* (Piattini et al. 96) p.547

MANTENIBILIDAD O FACILIDAD DE MANTENIMIENTO DEL SOFTWARE

Medida cualitativa de la facilidad de comprender, corregir, adaptar y/o mejorar el Software. (Pressman 98)

"Facilidad con que un sistema o componente sw. puede ser modificado para corregir defectos, mejorar el rendimiento u otros atributos, o adaptarse a un cambio de entorno".

Muy ligada a la calidad del Software métricas de mantenibilidad = métricas de calidad

También ligada a la complejidad del Software.

HERRAMIENTAS PARA LA INGENIERÍA INVERSA

3.1 Los Depuradores

Un depurador es un programa que se utiliza para controlar otros programas. Permite avanzar paso a paso por el código, rastrear fallos, establecer puntos de control y observar las variables y el estado de la memoria en un momento dado del programa que se esté depurando. Los depuradores son muy valiosos a la hora de determinar el flujo lógico del programa.

Un **punto de ruptura** (**breakpoint**) es una instrucción al depurador que permite parar la ejecución del programa cuando cierta condición se cumpla. Por ejemplo, cuando un programa accede a cierta variable, o llama a cierta función de la API, el depurador puede parar la ejecución del programa.

Algunos depuradores de Windows son:

OllyDbg: es un potente depurador con un motor de ensamblado y desensamblado integrado. Tiene numerosas otras características incluyendo un precio de 0 $. Muy útil para parcheado, desensamblado y depuración.

WinDBG:: es una pieza de software gratuita de Microsoft que puede ser usada para depuración local en modo usuario, o incluso depuración remota en modo kernel.

3.2 Las Herramientas de Inyección de Fallos

Las herramientas que pueden proporcionar entradas malformadas con formato inadecuado a procesos del software objetivo para provocar errores son una clase de herramientas de inserción de fallos. Los errores del programa pueden ser analizados para determinar si los errores existen en el software objetivo. Algunos fallos tienen implicaciones en la seguridad, como los fallos que permiten un acceso directo del asaltante al ordenador principal o red. Hay herramientas de inyección de fallos basados en el anfitrión que funcionan como depuradores y pueden alterar las condiciones del programa para observar los resultados y también están los inyectores basados en redes que manipulan el tráfico de la red para determinar el efecto en el aparato receptor.

3.3 Los Desensambladores

Se trata de una herramienta que convierte código máquina en lenguaje ensamblador. El lenguaje ensamblador es una forma legible para los humanos del código máquina. Los desensambladotes revelan que instrucciones máquinas son usadas en el código. El código máquina normalmente es específico para una arquitectura dada del hardware. De forma que los desensambladores son escritor expresamente para la arquitectura del hardware del software a desensamblar.

Algunos ejemplos de desensambladores son:

IDA Pro: es un desensamblador profesional extremadamente potente. La parte mala es su elevado precio.

PE Explorer: es un desensamblador que _se centra en facilidad de uso, claridad y navegación. No es tan completo como IDA Pro, pero tiene un precio más bajo.

IDA Pro Freeware 4.1: se comporta casi como IDA Pro, pero solo desensambla código para procesadores Intel x86 y solo funciona en Windows.

Bastard Disassembler: es un potente y programable desensamblador para Linux y FreeBSD.

Ciasdis: esta herramienta basada en Forth permite construir conocimiento sobre un cuerpo de código de manera interactiva e incremental. Es único en que todo el código desensamblado puede ser reensamblado exactamente al mismo código.

3.4 Los compiladores Inversos o Decompiladores

Un decompilador es una herramienta que transforma código en ensamblador o código máquina en código fuente en lenguaje de alto nivel. También existen decompiladores que transforman lenguaje intermedio en código fuente en lenguaje de alto nivel. Estas herramientas son sumamente útiles para determinar la lógica a nivel superior como bucles o declaraciones if-then de los programas que son decompilados. Los decompiladores son parecidos a los desensambladotes pero llevan el proceso un importante paso más allá.

Algunos decompiladores pueden ser:

DCC Decompiler: es una excelente perspectiva teórica a la descompilación, pero el descompilador sólo soporta programas MSDOS.

Boomerang Decompiler Project: es un intento de construir un potente descompilador para varias máquinas y lenguajes.

Reverse Engineering Compiler (REC): es un potente _descompilador_ que descompila código ensamblador a una representación del código semejante a C. El código está a medio camino entre ensamblador y C, pero es mucho más legible que el ensamblador puro.

3.5 Las Herramientas CASE

Las herramientas de ingeniería de sistemas asistida por ordenador (Computer-Aided Systems Engineering CASE) aplican la tecnología informática a las actividades, las técnicas y las metodologías propias de desarrollo de sistemas para automatizar o apoyar una o más fases del ciclo de vida del desarrollo de sistemas.

En el caso de la ingeniería inversa generalmente este tipo de herramientas suelen englobar una o más de las anteriores junto con otras que mejoran el rendimiento y la eficiencia.

Referencias

Especificación de requerimientos y requisitos:

http://www.mitecnologico.com/Main/EspecificacionesDeRequerimientos

Análisis de Sistemas. ¿Qué tiene que hacer un sistema?

http://alarcos.inf-cr.uclm.es/doc/ISOFTWAREI/Tema06.pdf

Especificación de Requisitos Software según el estándar de IEEE 830

Encontrar errores: introducción a la depuración en Visual Basic.

http://msdn.microsoft.com/es-es/library/kz97zky6%28v=vs.80%29.aspx

Los tres tipos de errores en programación:

http://msdn.microsoft.com/es-es/library/s9ek7a19%28v=vs.80%29.aspx

Encontrar y corregir errores mediante la depuración:

http://msdn.microsoft.com/es-es/library/9y92594t%28v=vs.80%29.aspx

Encontrar y corregir errores del compilador:

http://msdn.microsoft.com/es-es/library/32w91tab%28v=vs.80%29.aspx

Try..Catch..Finally: Visual Basic 2008 By Everts Garay Gaitan

Ingeniería Inversa:

http://www.iisa.com.ar/index.php/off-shore/217

http://www.xing.com/net/ingenieriainversa/que-es-la-ingenieria-inversa-aplicaciones-y-tecnologias-326226/que-es-la-ingenieria-inversa-19511031

http://www.kaslab.net/Telematicas_2006/Telematicas_2006-Ingenieria_Inversa-ramiro.pdf

http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/lopez_a_aa/capitulo4.pdf

http://cnx.org/content/m17432/latest/

http://cnx.org/content/m17433/latest/

http://ing-de-sistemas-unefa.lacoctelera.net/post/2010/06/14/exposicion-diseno-sistemas-diseno-archivos-y-base-de

http://es.scribd.com/doc/95726294/Ingenieria-de-Sofware