25

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES "A R A G O N"

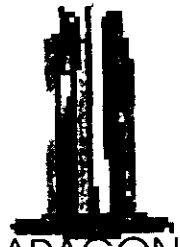
DISEÑO DE UNA INTERFASE USUARIO GRAFICA (GUI), USANDO METODOLOGIA ORIENTADA A OBJETOS

T E S I S

QUE PARA OBTENER EL TITULO DE
INGENIERO EN COMPUTACION
P R E S E N T A :

OSWALDO TOXTLE HERNANDEZ

E.N.E.P



ARAGON

DIRECTOR DE TESIS:
ERNESTO PEÑALOZA ROMERO

MEXICO, D.F. 1999

Dedicatorias:

A mi esposa e hijos:

Dedico este trabajo de tesis a mi chiquita linda y te doy gracias por haber aguantado todo los sacrificios que implicó llevar a su culminación esta tesis y por darme 2 preciosos angelitos ya que son ustedes el motivo de superación en mi vida.

A mis padres:

También dedico con mucho cariño este logro a mis padres, ya que gracias a ustedes he obtenido una buena educación y lo más importante, el privilegio de estar vivo.

Agradecimientos:

Al Ing. Ernesto Peñaloza Romero:

Gracias por dirigir este proyecto de tesis, por siempre apoyarme en todo y por se amigo.

Al Ing. Juan Gastaldi Pérez:

Gracias por apoyarme siempre en todos los trámites escolares y siempre orientarme.

Al Mtro. en Ciencias Luis Nava:

Gracias por enseñarme por todo lo que sé de orientación a objetos, y a tomar un nuevo enfoque sobre como se debe aprender y analizar todos los aspectos relacionados con la informática.

Al matemático Antonio Carrillo:

Un especial agradecimiento, ya que sin su apoyo no hubiese sido posible la conclusión de éste trabajo de tesis.

A mis compañeros de carrera:

Un agradecimiento a mis invaluables amigos y compañeros de carrera: Carlos Paredes, Teresa Jiménez, Mario León, Cecilia Hernández y Angélica Ramírez.

PREFACIO

La interfaz humano-computadora es el conjunto de elementos que permiten a un usuario interactuar con un sistema computarizado; dicha interfaz humano-computadora se integra de elementos de equipo (hardware), como desplegados de rayos catódicos, teclados, ratones (mouse), etc, y elementos de programación (software), como iconos, menús, formas de entrada de datos, etc.

La frustración y ansiedad son parte de la vida diaria para muchos usuarios de sistemas de información computarizados. Ellos se esfuerzan por aprender comandos complejos o por entender sistemas de selección de menú, que se supone son para ayudarlos a hacer su trabajo. Algunas gentes encuentran serios casos de shock (rechazo) de computadora, terror a la terminal, o neurosis de red. Esto hace que ellos eviten usar sistemas computarizados. Mientras que por el contrario los usuarios satisfechos de un sistema computarizado frecuentemente tienen la experiencia de claridad, de sentirse competentes e inclusive de sentirse poderosos al hacer su trabajo con dicho sistema. El diseño de interfaces usuario gráficas, se vio como la pintura que se puso al final de un proyecto, ahora parece ser el marco de acero sobre el cual se construye la estructura. Sin embargo, una conciencia de los problemas y un deseo de hacer el bien no son suficientes. Los diseñadores, administradores y programadores deben estar dispuestos a caminar juntos y pelear por el usuario. Los enemigos a vencer incluyen inconsistentes lenguajes de comandos, secuencias de operación confusas, formatos caóticos de exhibición, terminología inconsistente, instrucciones incompletas, complejos procedimientos de recuperación de error, y mensajes de error extraviados o amenazadores.

Las empresas lideres desarrolladoras de computadoras y de sistemas reconocen que la interfaz humano-computadora, es crítica para determinar el éxito o el fracaso de un producto; y que los sistemas que son más fáciles de usar tendrán una ventaja competitiva en la recuperación de información, la automatización de oficina, y la computación personal.

El éxito de un sistema computacional, es decir, su aplicación para la solución de problemas reales, radica en gran medida en la calidad de su interfaz con el usuario, la cual debe ser capaz de proporcionar mecanismos para una explotación fácil y eficiente del sistema. Por lo anterior, desde hace algunos años se ha venido dedicando cada vez más tiempo y esfuerzo en el análisis, diseño e implementación de interfaces eficaces, lo cual se ha convertido hoy en día en una interesante y relevante área de investigación y desarrollo.

En este trabajo de tesis en el primer capitulo se tratara de la historia del concepto orientado a objetos, conceptos básicos del paradigma orientado a objetos, así como la importancia del concepto paradigma entre otros puntos, el capitulo segundo nos hablara

sobre la complejidad del software, además de cómo y por que se necesita una metodología, y se planteara una metodología de análisis y diseño orientado a objetos. El capitulo tercero versara sobre el análisis de la interface, sus elementos, las teorías y puntos básicos a tomar en cuenta como la teoría del color y el uso de fuentes. El capitulo cuarto planteara el diseño usado para la librería cladiug, la cual es la que permite la construcción de interfaces usuario gráficas. Para terminar con el capitulo quinto en que se muestra lo mas relevante de la codificación.

CONTENIDO

CAPITULO I

CAPITULO II

CAPITULO III -

CAPITULO IV

CAPITULO V

Glosario

Bibliografía

CAPITULO I

HISTORIA Y CONCEPTOS BASICOS DEL PARADIGMA ORIENTADO A OBJETOS

HISTORIA DEL CONCEPTO ORIENTADO A OBJETOS.

El nacimiento de la *tecnología Orientada a Objetos* data de principios de los años 60'S, surgió de la necesidad de describir y simular fenómenos como las redes neuronales, los sistemas de comunicación, el flujo de trafico, los sistemas de producción, etc.

En la primavera de 1961, *Kristen Nygaard* dio origen a las ideas de un lenguaje que serviría al doble propósito de describir el sistema y de programar para la simulación junto con *Ole-Johan Dahl*, Nygaard perfecciono el lenguaje de simulación que ahora se conoce como ***Simula***. Se pretendía que Simula fuese un lenguaje de programación para la descripción y la simulación de sistemas. Sin embargo se dieron cuenta que Simula proporcionaba nuevas y poderosas posibilidades, para fines distintos de la simulación, como la elaboración de prototipos y el diseño de aplicaciones.

A fines de los años sesenta Alan Kay desarrolla otra propuesta de tecnología OO, en la Universidad de Utha, con las ideas centrales de Simula. Dando como resultado el lenguaje ***Smalltalk*** y es de este lenguaje de donde surge el termino orientado a objeto.

CRISIS DEL SOFTWARE.

Durante los primeros años de desarrollo de las computadoras el hardware sufrió continuos cambios (y aún en la actualidad los sigue sufriendo), mientras que el software se veía simplemente como un añadido.

El desarrollo del software se realizaba virtualmente sin ninguna planificación aunque comenzaron a aparecer algunos indicios de los costos o planificación. Durante este periodo se utilizaba en la mayoría de los sistemas una orientación por lotes. Algunas excepciones notables fueron algunos sistemas interactivos tales como el sistema de reserva de billetes de la American Airline *(SABRE)* y los sistemas de tiempo real orientados a la defensa tales como *SAGE*.

La creciente demanda del software y el imperativo aumento de complejidad exige de los desarrolladores la creación de software nuevo en muy cortos periodos de tiempo. La reutilización del software se vuelve imperiosa. El problema estriba en que los desarrolladores de software suelen pensar en función de la aplicación a la que desean dar solución y no en la posibilidad de hacer una rutina que puedan utilizar cada vez que requieran iniciar un nuevo sistema.

Según un pequeño estudio realizado por nosotros nos dio como resultado que los sistemas tengan un tiempo de vida media de dos años que es sólo el tiempo necesario para que el sistema se destruya así mismo agobiado por los múltiples conflictos que las variables generaran o por volverse totalmente incomprensibles, aun para la persona que les da mantenimiento.

Por una parte el software requerido es cada vez más complejo, la calidad de los profesionales a decaído y finalmente el tiempo empleado tanto para el análisis como para el diseño es considerado como una perdida de tiempo que redunda en software de mala calidad con altos costos de mantenimiento y corta vida productiva. Esta fue la llamada **crisis del software.**

La **crisis del software** se refiere a un conjunto de problemas encontrados en el desarrollo del software. Los problemas no están limitados al software que no funciona adecuadamente; sino que la crisis del software abarca los problemas asociados con el desarrollo del software, como mantener un volumen creciente de software existente y como se puede esperar satisfacer la demanda creciente de software.

Los proyectos de desarrollo de software (sistemas) se realizaban y se realizan frecuentemente con sólo una vaga indicación de los requerimientos del cliente normalmente la comunicación entre el cliente y el que desarrolla era muy escasa.

El desafío intelectual del desarrollo del software es una de las causas de la crisis del software, fue así como *Edsger Dijkstra* fijándose en dicho trabajo intelectual sentó las bases para la creación de lo que ahora se conoce con el nombre de Ingeniería de Software, ya que anteriormente el problema de desarrollo de software era hecho por programadores y no por gente con conocimientos en metodologias y técnicas para la creación y mantenimiento de software.

En la actualidad la ingeniería de Software a coadyuvado a dar en parte solución a la crisis del software, sin embargo la ingeniería de software no tiene especializaciones como las hay en medicina, ya que no existe un medico que cure todo, debido a la complejidad del cuerpo humano. Asi tambien es igualmente de complejo el desarrollo del software

En el trabajo realizado en *"Programers at Works"* ha demostrado que el trabajo intelectual para el desarrollo de software es complejo. En este trabajo se compara el diseño del avión ***BOING_747*** como sistema; el cual para su desarrollo se requirió alrededor de 25 especialistas (expertos en aeronáutica, aerodinámica, resistencias de materiales, vuelo, ergonomía etc) con su equivalente en esfuerzo intelectual que es el procesador de textos ***WORD 2.0*** en su versión para Windows, la cual fue diseñada por 8 personas en 10 meses, con una característica en común todas ellas, la ingeniería de software y ciencias de la computación. Lo cual nos da una idea de la responsabilidad de que el sistema sea estable, es decir que no se caiga o falle, ya que no se espera que estando el avión en vuelo una turbina explote, que tenga problemas al despegar o para mantenerse en vuelo. Lo mismo sucedió con el software de WORD tenia la responsabilidad de que no se comportara de manera extraña, fallara o se cayera el sistema.

La crisis del software no desaparecerá de la noche a la mañana, el reconocimiento de los problemas y sus causas, así como el desenmascaramiento de los mitos del software son los primeros pasos hacia la soluciones; luego las soluciones deben dar asistencia practica al desarrollo de software, mejorar la calidad del software y finalmente permitir al mundo del software emparejarse con el mundo del hardware.

No hay un método único mejor que solucione la crisis de software sin embargo el enfoque orientado a objetos ha demostrado ser una muy buena solución

EVOLUCION DE LOS TIPOS DE DATOS.

DATOS TIPO BINARIO.

En un principio, solo un tipo de dato describia el universo de las cadenas de bits, en la memoria de una computadora, el tipo de dato llamado ***palabra***. Las palabras son cadenas de bits con un tamaño fijo que pueden utilizarse como unidades de información. Sin embargo cuando los investigadores diseñaban sus programas se dieron cuenta de la necesidad de representar datos de diferentes formas para propósitos diversos. Cabe mencionar que en este periodo los lenguajes que se utilizaban eran ensambladores. Es así como nace lo que se conoce como ***Tipo de dato***.

TIPOS DE DATOS.

Un tipo de dato describe determinada clase de datos con su representación y un conjunto de operaciones validas que tienen acceso y control sobre los datos. De esta forma cada tipo de datos es un ente conocido, protegido del uso accidental, por ejemplo el tipo de datos carácter describe la clase de datos que puede operar un programa. Además, se dispone de un conjunto de operaciones (funciones) para crear, destruir, y administrar los datos de tipo carácter. Puesto que las operaciones aritméticas como la suma y la resta no están definidas para los datos carácter no se permiten solicitudes de cálculo.

Para esta época se utilizaba, lenguajes de alto nivel con declaración de variables y uso de sentencias. Por ejemplo tipo real, tipo entero, tipo carácter, tipo registro, etc.

TIPOS DE DATOS DEFINIDOS POR EL USUARIO (TDU).

A principio de los años 70, un programador podia hacer referencia solamente a los tipos datos sin poder expresar explícitamente datos tales como: mes, fecha, hora, color, etc. y estructuras de datos complejas como son: pila y árbol entre otros. Estas ideas debían introducirse en forma implícita en alguna parte del código del programa.

Otra característica limitante de los tipos de datos integrados al lenguaje, era su definición mediante la forma como se almacenaba físicamente la información. Esto tenia una relación poco útil con los objetos que se deseaban implementar mediante la aplicación.

Los primeros lenguajes en proporcionar los TDU fueron Pascal y Algol 68, por ejemplo en Pascal un programador podía escribir:

TYPE MES=(ENERO, FEBRERO, MARZO, ABRIL, ...,DICIEMBRE)

Esta expresión definía entonces el TDU MES como el conjunto de doce *cadenas* con dicho conjunto podía definir operaciones relaciónales como comparar dos variables MES para ver cual es menor o si son iguales, otras operaciones podían incluir cálculos del mes anterior o posterior. Los TDU aumentaron el poder de expresión de los lenguajes de programación, permitiendo al programador pasar de los tipos impuestos del fabricante a los tipos impuestos por el usuario; lo que es más importante contribuyeron a traducir los tipos cotidianos en tipos de datos codificados.

TIPOS DE DATOS ABSTRACTOS (TDA).

Las exigencias por representar datos del mundo real complejos llevaron a los lenguajes de programación a crear los tipos de datos abstractos que son antecesores de los objetos.

Los tipos de datos abstractos(TDA) extienden el concepto de los TDU añadiendo el encapsulado. Los TDA contienen la representación y operaciones de un tipo de datos. La característica de encapsulado de los TDA no solo oculta la implantación del tipo de dato, sino que proporciona un muro de protección que impide el uso inadecuado de su estructura. Toda la interface ocurre a través de operaciones definidas dentro del TDA así las operaciones proporcionan un medio bien definido para tener acceso a los datos de la estructura. En resumen los TDA dan a los datos de la estructura una "interface publica" a través de sus operaciones permitidas. Sin embargo, las representaciones y el código ejecutable (método) de cada operación son privadas.

El uso de los TDA apareció por primera vez en SIMULA 67 donde su implementacion se llama "class" (clase). MODULA se refiera a su implementación de los TDA como "module" (modulo), mientras que para ADA utiliza la palabra "package" (paquete). En todos los casos el TDA proporciona una forma más eficiente para que los diseñadores de los sistemas identifique los tipos de datos del mundo real y los empaque en forma conveniente y compacta. De esta forma, los TDA pueden definir cosas tales como: fecha, pedidos, empleado, paneles de pantalla entre otras.

OBJETOS.

Cada objeto se puede considerar como algo independiente, con su propio comportamiento aunque cada objeto debe encapsular un registro fisico de sus propios datos(estruc-

tura), no se requiere el encapsulado de una copia física de cada método operativo como en el TDA pues se desperdicia espacio. Como en cada objeto esta contenido el mismo código, las operaciones de un objeto solo deben ser virtualmente disponibles para un objeto. En otras palabras todas las operaciones que se aplican a un objeto particular también se aplican todos los objetos. Aun cuando dos objetos tengan características idénticas, no son el mismo, al igual que dos gemelos idénticos no son la misma persona.

CONCEPTOS BASICOS DEL PARADIGMA ORIENTADO A OBJETOS.

La frase ***orientado(a) a objetos*** se ha vuelto muy popular en los últimos años se habla de sistemas operativos orientados a objetos, lenguajes orientados a objetos, programación orientada a objetos(POO), hardware orientado a objetos, etc. Sin embargo, el concepto surgió hace ya 25 años con la creación del lenguaje SIMULA. Los conceptos generales más utilizados en el modelo orientado a objetos son:

- **PARADIGMA**: Es una forma conveniente(útil) de ver el mundo real, un enfoque particular o filosofía de percibir el mundo real. En una sección mas adelante se detallara a profundidad el concepto.
- **ABSTRACCION**: Es el proceso de identificar u obtener las características o atributos esenciales de un objeto o sistema, que hace énfasis en algunos detalles o propiedades y suprime otros. Teniendo en cuenta la perspectiva del observador.

  Una buena abstracción es aquella que hace énfasis en los detalles significativos y suprime los irrelevantes. La abstracción es un elemento fundamental en el paradigma orientado a objetos.

  La abstracción debe enfocarse en que es un objeto y que hace antes de pensar en la implementación por ejemplo, un automóvil puede abstraerse como un objeto que sirve para desplazarse a mayor velocidad, sin importar como lo haga. Dentro de la abstracción es característico que un objeto pueda abstraerse de diversas formas, dependiendo del observador. Así el automóvil antes mencionado puede ser visto como un objeto de colección para un coleccionista, una herramienta de trabajo por un corredor profesional o una mercancía por un vendedor, etc.
- **ENCAPSULACION(OCULTAMIENTO DE INFORMACION):** Es el proceso de ocultar la implementación de un objeto, a la vez que permite una interface publica por medio de sus operaciones permitidas. El objeto esconde sus datos de los demás objetos y permite el acceso a los datos mediante sus propios métodos.

  Al encapsular u ocultar la información, se separan los aspectos externos de un objeto (los accesibles para todos) de los detalles de implementación (los

accesibles para nadie). Con esto se trata de lograr que al tener algún cambio en la implementación de un objeto no se tenga que modificar los programas que utilizan tal objeto.

Siguiendo con el ejemplo del automóvil, se sabe que existen diversos mecanismos para que funcione este, en particular se tiene el sistema de frenado que todo mundo sabe que sirve para detener el auto al pisar el pedal del freno (interface publica), pero solo el mecánico sabe los detalles de implementación (parte privada). Por otro lado, si en algún momento se cambia el tipo de frenos (de tambor a discos) para el conductor es transparente.

- **MODULARIDAD:** Consiste en dividir un programa en partes llamadas módulos, los cuales se pueden trabajar por separado. En términos de programación los módulos pueden compilarse por separado y la división no depende de cierto numero de líneas, sino que es una división en términos de integrar en un modulo un conjunto de procedimientos relacionados entre si, junto con los datos que son manipulados por tales procedimientos. El *objetivo de la modularidad es reducir el costo de elaboración de programas* al poder dividir el trabajo entre varios programadores.

Por ejemplo, un automóvil esta constituido por un conjunto de módulos, tales como un sistema eléctrico, uno mecánico y uno de frenado. Cada modulo trabaja por separado y el especialista solo conoce la forma en que se relaciona su modulo con los otros, pero no tiene por que saber los detalles de funcionamiento de otros módulos o sistemas.

- **CLASIFICAR:** Ordenar o disponer por clases, coordinar, catalogar.
- **CLASIFICACION:** Manera de ordenar los conceptos conforme a ciertas relaciones existentes entre ellos; es la distribución de los objetos según sus

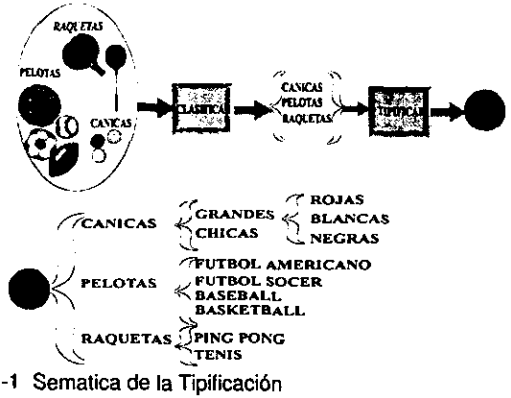


Figura 1-1 Sematica de la Tipificación

semejanzas y diferencias de acuerdo con un común denominador.

- **JERARQUIZACION:** Es el proceso de tipificación en el cual no solamente hay que partir de una semántica, como en el proceso de clasificación, sino hay que partir de un tipo o clase. Dado un elemento de "referencia" o "primario" (clase o tipo) se define un conjunto de objetos que determinan una especialización de dicho elemento. El objetivo de jerarquizar es definir los distintos niveles de abstracción.

  La jerarquizacion es el resultado de ordenar la abstracción. Un conjunto de abstracciones forman una jerarquía, en un sistema complejo. La jerarquizacion tiene relación con la herencia.

- **DOMINIO:** Es un mundo aparte ya sea real, hipotético o abstracto, habitado por un conjunto de objetos distintos que se comportan de acuerdo a reglas y políticas que son características del mismo.

  Existen varios tipos de dominios como por ejemplo:

  - De Aplicación (los objetos que deberán resolver el problema)
  - De Servicio (Utilerias para mantenimiento y desarrollo)
  - De implementacion (redes, lenguajes ,S.O., etc.)
  - Arquitectónicos (control y manejo de todo el sistema)

- **DOMINIO DEL PROBLEMA:** Son todos aquellos elementos que se deben considerar para ofrecer la solución de un problema del mundo real(la creación del sistema de software).

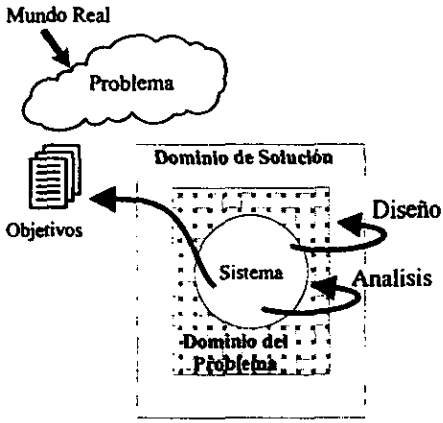


Figura 1-2 -Representación de los Dominios y Herramientas para manejarlos.

- **DOMINIO DE SOLUCION:** Son todos aquellos elementos de software que sirven para implementar el sistema de software(exitosamente). Atacando al sistema por lo más importante y no por lo más urgente.

- **OBJETO:** Un objeto es una entidad tangible, visible o abstracta (cualquier *cosa que pueda ser comprendida intelectualmente o cualquier cosa hacia la* cual una idea o acción es dirigida) que exhibe un comportamiento bien definido. Un objeto tiene estado, comportamiento e identidad.

Además el concepto de objeto no esta acotado a ninguna semántica, es decir, en todos los sistemas de encuentran objetos.

- Propiedades computaciones de los objetos
- Recibir y enviar datos, información (mensajes)
- Ejecutar un proceso de otro objeto o interno
- Puede ser ejecutado por otros objetos
- Un objeto no puede estar en dos estados al mismo tiempo

Si vemos al objeto como modulo se le pueden agregar más comportamientos o propiedades computacionales como ocultar información, ocultar comportamiento o funciones, compartir información o compartir comportamientos.

MODELO CONCEPTUAL DE OBJETO

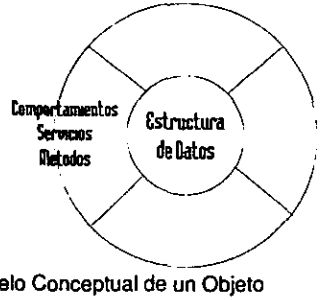


Figura 1-3 Modelo Conceptual de un Objeto

El ciclo de vida de un objeto es el conjunto de estados y eventos que rigen al objeto.

- **IDENTIDAD:** Es la propiedad de un objeto que lo distingue de todos los demás. En un programa se trata generalmente de un identificador.
- **ESTADO:** El estado de un objeto abarca todos los atributos, propiedades o caracteristicas distintivas del comportamiento de un objeto y los valores de cada uno de las propiedades. En términos de programación puede decirse que las propiedades son las variables que sirven para describir dicho comportamiento del objeto.

En un objeto pueden existir varios estados, por ejemplo: el objeto reservación aérea puede tener los siguientes estados:

– reservación solicitada

– reservación en lista de espera

– reservación confirmada

– reservación cancelada

– reservación satisfecha(una vez que el avión a despegado)

– reservación archivada

- **COMPORTAMIENTO:(SERVICIOS, METODOS)** Es la forma como actúa o reacciona un objeto en términos de sus cambios de estado, envío y recepción de mensajes. Esta formado por la definición de las operaciones (funciones y procedimientos) que puede realizar este objeto.

  Cabe mencionar que el termino operación se refiere a una unidad de procesamiento que puede ser solicitada, el procedimiento se implanta mediante un método, es decir, el método es la especificación de como llevar a cabo la operación.

- **EVENTO:** Es una abstracción de un incidente o señal en el mundo real, que nos habla de cosas que se están moviendo a un nuevo estado. Es decir un cambio en el estado de un objeto.

  En el paradigma orientado a objetos el mundo se describe en términos de objetos y sus estados así como los eventos que modifican esos estados. Por ejemplo: un cliente hace una reservación aérea, una maquina se descompone, se termino una tarea, etc. Un estado del objeto reservación aérea xxxxx pasa de ser una reservación de lista de espera a una reservación confirmada. El nombre de este evento podría ser reservación en lista de espera para el objeto xxxxx confirmada.

  – Propiedades computacionales de los eventos:

  – Los eventos ocurren por una operación

  – Un evento puede ocasionar la reacción en cadena de otros eventos

  – Los eventos pueden asociar un objeto con otro

- **MENSAJE:** Es la forma de comunicación entre objetos, para que un objeto haga algo le debemos enviar una solicitud. Esta hace que se produzca una operación, la operación ejecuta el método apropiado y de manera opcional produce una respuesta; el mensaje que constituye la solicitud contiene el nombre del objeto, el nombre de la operación, y a veces un grupo de parámetros.

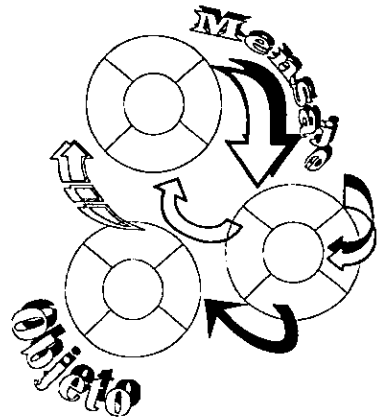


Figura 1-5 Modelo de Conceptual de Mensajes

- **ESCENARIO:** Es la secuencia de eventos que ocurren durente una particular ejecución del sistema.
- **CLASE:** Es un conjunto de objetos similares que comparten una estructura *y comportamientos comunes. Las clases se consideran núcleos que contie-*

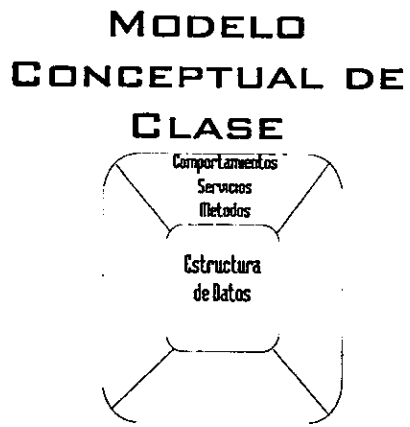
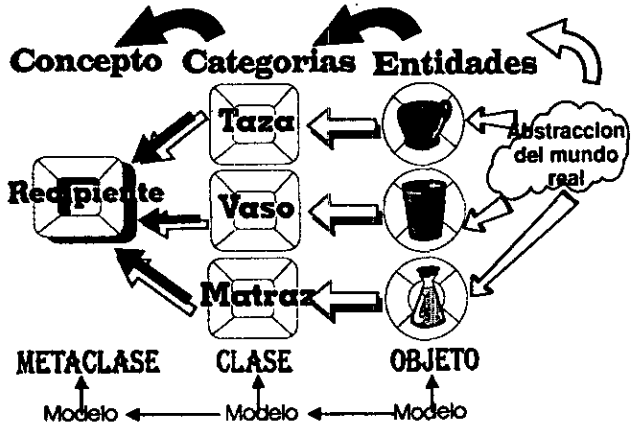


Figura 1-4 Modelo Conceptual de Clase

Figura 1-6
Semantica de la
Metaclase



nen una plantilla de la forma en que se utilizan los datos y código. Es decir es la generalización de los objetos.

- **METACLASE: (CLASES PARAMETRICAS, CLASES POLIMORFICAS):** Son generalizaciones de clase, sus instancias de la metaclase son clases. El diagrama que ejemplifica la metaclase es la figura 1-6:
- **HERENCIA:** La herencia es una relación entre clases del tipo generalización/especialización donde una clase comparte la estructura y/o comportamiento definido en una o más clases. Es decir el reuso de clases por clases; en el que una nueva clase reusa las características de una y solo una clase a lo que se llama herencia simple, dos o más clases herencia múltiple.

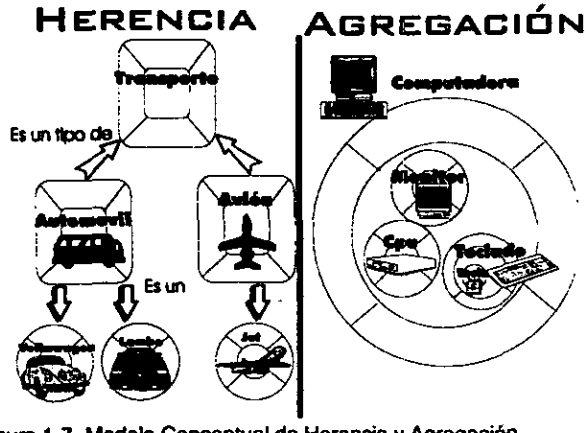


Figura 1-7 Modelo Conceptual de Herencia y Agregación

Definiéndose una jerarquía de clases. En tal jerarquía algunas clases son subordinadas a otras llamadas subclases o clases derivadas; una subclase define el comportamiento de un conjunto de objetos que heredan algunas caracteristicas de la clase padre, pero adquieren caracteristicas especiales no compartidas por el padre(superclase), en este sentido se dice que la subclase es una especialización de la clase padre.

La herencia es la contribución más importante del paradigma orientado a objetos, pues mediante este mecanismo es posible lograr la principal meta del modelo objetual que es la reutilización de código.

- **AGREGACION:** La agregación es una relación entre objetos del tipo todo/parte, es decir el reuso de objetos por objetos; en este sentido es un tipo especializado de asociación

  La agregación encapsula partes como secretos del todo

- **POLIMORFISMO:** Es la propiedad que permite a una operación tener diferente comportamiento en objetos diferentes; en otras palabras, objetos diferentes reaccionan de modo diferente al mismo mensaje.

  Existen dos tipos de polimorfismo: sobrecarga de operadores y sobrecarga de funciones.

PARADIGMA ESTRUCTURADO VS PARADIGMA ORIENTADO A OBJETOS.

La idea fundamental del enfoque estructurado es romper un programa en unidades

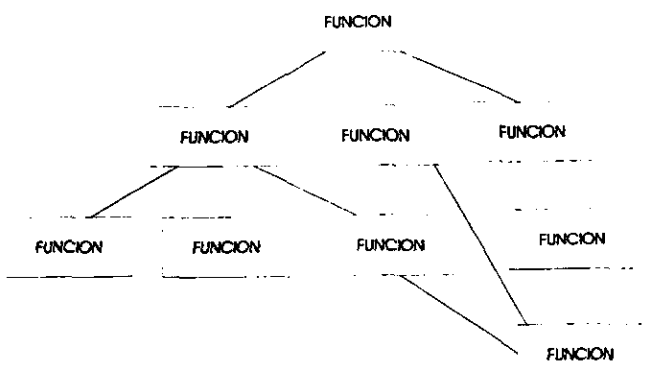


Figura 1-8 Representacion de la División Funcional

más pequeñas denominadas funciones (ver figura 1-8) cada una de las funciones

(almenos idealmente) tiene un propósito claramente definido así como un interfaz a las otras funciones del programa. la información se pasa entre funciones utilizando parámetros. y las funciones pueden tener datos locales a los cuales no se puede acceder fuera del ámbito de la función.

El enfoque estructurado utiliza el método descendente y de refinamiento sucesivo (ciclo de vida de desarrollo en cascada) y cada una de las funciones se invocan sucesivamente. Los programas estructurados son más fáciles de escribir y mantener que los programas no estructurados, sin embargo a medida de que el tamaño de los programas aumenta y se hacen muy grandes y más complejos, el énfasis se pone en los tipos de datos que se procesan. las estructuras de datos en un programa se hacen tan importantes como las operaciones realizadas por ellos. Los tipos de datos se procesan en muchas funciones dentro de un programa estructurado y cuando se producen cambios en esos tipos de datos las modificaciones se deben de hacer en cada posición que actúan sobre esos tipos de datos.

Esta tarea además de consumir gran tiempo puede ser frustrante en programas que contengan millares de líneas de código y centenares de funciones. Esta circunstancia se produce esencialmente porque muchas funciones comparten datos(variables globales,ver figura 1-9) así por ejemplo si se añaden nuevos datos se necesitaran modificar todas las

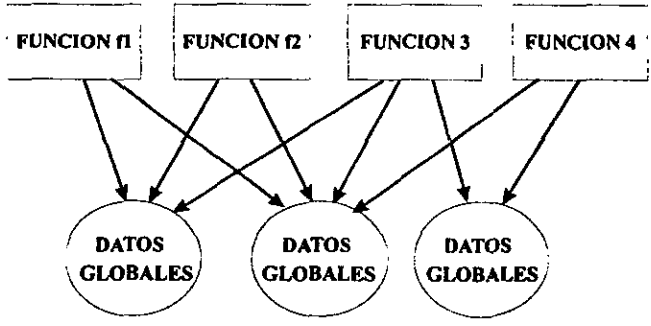


Figura 1-9 Esquema de Acceso a Datos por Funciones

funciones que acceden a los datos, de modo a que ellas también puedan acceder. En el mundo orientado a objetos las estructuras de datos se relacionan con los objetos y solo pueden ser utilizadas mediante los métodos diseñados para ese tipo de objeto.

El enfoque estructurado no ha dado solución ha muchos de los problemas a los que se enfrentan actualmente los desarrolladores una parte de esto es debido al desconocimiento de lo que realmente es la programación estructurada a los enfoques incorrectos o la negligencia en la implantación de los módulos. La programación estructurada nació para evitar los problemas no para resolverlos. La programación estructurada ataca el dominio de solución y no el del problema, la orientación a objetos fue creada para atacar el dominio del problema

Otro problema que suele presentar es que los programas estructurados sean, con frecuencia, difíciles de diseñar ya que funciones y estructuras de datos no modelan muy bien el mundo real.

El ultimo inconveniente que consideramos se produce cuando un equipo de programadores trabajan en una aplicación. En un programa estructurado, a cada programador, se le asigna para construir una tarea especifica de funciones y tipos de datos. Los cambios inocentes en apariencia tienen ramificaciones imperceptibles que pueden provocar una reacción en cadena de errores. Dado que diferentes programadores manipulan funciones independientes que comparten datos, los cambios que un programador hace a los datos deben ser reflejados en el trabajo del resto del equipo, los problemas y los errores de comunicación se reflejan en el diseño final.

Para ayudarnos a lidiar con la complejidad se comenzó a utilizar la programación estructurada la cual redujo el espagueti en el código, pero la programación seguía basándose en una secuencia esperada de instrucciones de ejecución. A mediados de los ochenta, las autoridades de las técnicas estructuradas afirmaban que era imposible construir sistemas de 50 millones de líneas de código.

El paradigma orientado a objetos fue desarrollado, esencialmente, como evolución, por las limitaciones que otros paradigmas tenían y tienen, como en el caso del paradigma estructurado.

El enfoque orientado a objetos enfatiza los datos, al contrario que la programación estructurada que enfatiza los algoritmos.

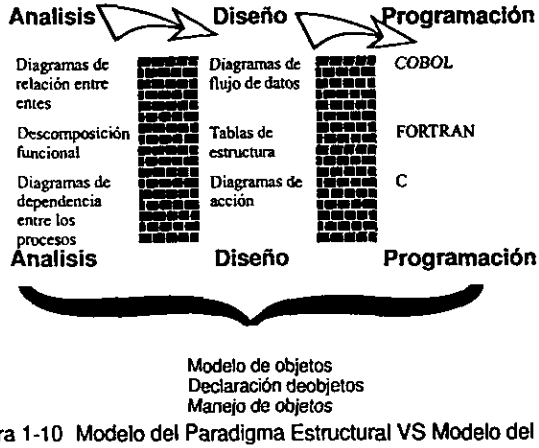


Figura 1-10 Modelo del Paradigma Estructural VS Modelo del Paradigma Objetual

Se ha puesto de relieve que las técnicas orientadas a objetos nos proporcionan una forma más natural de pensar acerca de la complejidad del mundo, a diferencia de las técnicas utilizadas para el análisis, diseño y programación convencional. Con el enfoque orientado a objetos se pueden visualizar con facilidad los procesos que necesitan automatizarse pero

en términos de objetos, reglas de eventos y formas de activación mientras que es más difícil establecer la relación con las tablas de estructura en los diagramas de flujo de datos. Ya que esto solo modela la parte estática de un sistema y no la dinámica.

En el análisis tradicional se utilizan los modelos de relación entre entidades y la descomposición funcional. Los analistas generan matrices que asocian las funciones con los tipos de entidades, el diseño tradicional utiliza diagramas de flujo de datos, tablas de estructura, diagramas de acción. Los lenguajes de programación tienen un modelo conceptual distinto ya que utilizan construcciones de código como: COBOL, FORTRAN, C, etc. En las técnicas orientadas a objetos todos utilizan el mismo modelo conceptual analistas diseñadores, programadores y de modo particularmente importante los usuarios finales(ver figura 1-10).

Para finalizar mencionaremos que al analizar y diseñar orientado a objetos de manera natural podemos manejar esquemas y arquitectura cliente / servidor.

EL FUTURO DEL PARADIGMA ORIENTADO A OBJETOS.

En gran medida, el futuro del software depende de los componentes reutilizables. Una de las mejores vias para conseguir esta caracteristicas es mediante los componentes OO.

Esto nos lleva a un mundo de clases reutilizables, donde la mayor parte del proceso de construcción de software consistirá en ensamblaje de clases ya existentes y probadas.

Las técnicas OO ligadas a las herramientas del CASE con un generador de códigos y un deposito (también orientado a objetos) además del uso de base de datos OO, así como la programación visual constituirán el mejor camino conocido para construir software, es decir una verdadera ingeniería de software.

Tal vez el beneficio mas importante a largo plazo sea el cambio en nuestra forma de pensar.

LA IMPORTANCIA DEL CONCEPTO PARADIGMA.

En parrafos anteriores hemos mencionado varias veces la palabra paradigma y en esta sección analizaremos el uso, sobre uso y abuso del termino paradigma, el significado de la palabra y las varias formas en que se encuentra utilizada. Intentaremos apreciar como el concepto nos ayuda a comprender la actividad del análisis y programación de los sistemas y a evaluar efectivamente las alternativas disponibles para aumentar el desempeño de los que se dedican a esta actividad.

La disciplina de programación de computadoras ha sido por muchos años un lugar de oportunidad para el desarrollo del intelecto humano y también para el placer de los que apreciamos esa labor. Para llevar a cabo dicha tarea existen diversos lenguajes, los cuales están basados en una serie de premisas o postulados que definen las características y limitantes de los mismos. El concepto de paradigma fue introducido por Thomas Kuhn, en su estudio sobre la forma como avanza el pensamiento científico dentro de este contexto un paradigma es mas que un simple modelo de pensamiento, es toda una visión distinta de ver la realidad.

COPERNICO DESPLAZO UN PARADIGMA

El diccionario define paradigma como "ejemplo, modelo, ejemplar", esta definición tiene cierta utilidad en cuanto el termino en el contexto de idiomas, sin embargo, para la especificación de programas y si queremos trazar las raíces etimológicas del termino en nuestro contexto, nos conviene considerar la obra ya clásica de Thomas Kuhn, La estructura de la Revolución científica. En dicha obra Kuhn analiza las revoluciones científicas como lo que el llama (paradigm shifts) los desplazamientos de paradigma que sufre una ciencia sobre el camino de su desarrollo.

Para Kuhn, un cambio de paradigma altera la conceptualización de entidades con que trabaja el profesionista en su campo; si vamos a entender que hay, en este sentido, paradigmas de programación, tendremos que apreciar que el programador, operando bajo *un paradigma nuevo (o al menos para el, diferente)* experimenta una alteración en su conceptualización del proceso de resolución de problemas por medio de la computadora. Hay una modificación esencial de perspectiva sobre lo que es un problema y lo que representa una solución.

Pensemos por un momento en una revolución científica, por tratar un fenómeno accesible a la experiencia de cada quien; La causa que todos a una edad tierna participemos en el desplazamiento de paradigma asociado: la revolución fomentada por la teoría de Nicolai Copernico.

La Astronomía Tolomeica se fundamento en la conceptualización de un universo con la tierra a su centro. Simplificando toda una revolución a una sola afirmación, podemos decir que 1543 Copernico propone que los planetas giran, no alrededor de la Tierra sino alrededor del sol. El cambio de mentalidad que tal postura requiere es impresionante y la agitación social causada por la obra de Copernico fue enorme. (Ver Figura 1-11)

El impacto científico que un cambio de perspectiva conlleva esta muy bien ejemplificado por la vida de Johanes Kepler. Kepler, una generación mas joven que Copernico, dedica su vida a entender los movimientos de los planetas. Armado con los datos voluminosos y precisos acumulados por Tycho Brahe, Kepler finalmente logra ver los planetas girando en órbitas elípticas. Los datos fríos simplemente existieron independiente de perspectiva, predisposición mental, paradigma, *pero la interpretación de los datos, las posibles formas de organizarlos, combinanrlos, y abstraer patrones inherentes en ellos es otra cosa.* Es difícil imaginar que Kepler, por todo el genio que fuera, hubiera tenido la

creatividad necesaria para descubrir las elipses ocultas entre los datos si estuviera buscando dentro del bosque de epiciclos de la astronomía tolomeica . Bien para apreciar el trabajo de Kepler (y Galileo y Newton y tres siglos de científicos más) hay que reconocer el

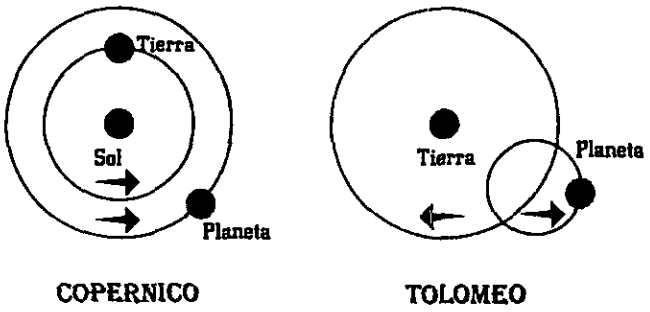


Figura 1-11 Las órbitas de los planetas bajo paradigmas distintos

paradigma establecido por Copernico dentro del cual científicamente hablando por lo menos vivió. Y de igual manera hay que entender el campo de la psicología de este siglo en términos de la revolución de Freud, el de Biología en términos de Darwing y la lingüística de los últimos 40 años en términos de Chomsky, por mencionar unos cuantos ejemplos pero esto ¿qué tiene que ver con el programador/analista, sentado frente a su terminal, encargado del diseño e implantación de un sistema para el control de producción de una línea de ensamblaje? Los que hablan de la actividad de programación en términos de paradigmas creen que mucho.

TAMBIEN LOS PROGRAMADORES SE DESPLAZAN.

La resolución de problemas por medio de la computadora no difiere de otras actividades científicas, la organización de los varios aspectos de un problema, las posibilidades de solución abiertas para consideración, las técnicas utilizadas en las palabras de Kuhn, las "entidades" en términos de las cuales se conceptualizan los problemas y sus soluciones, todos son propiciados (y de igual manera, limitados) por el paradigma vigente. El esquema dominante para la programación de la computadora durante los últimos 20 años ha sido la programación estructurada. Dentro del paradigma que engloba la programación estructurada, que llamaremos "procedimentista", se visualiza la solución de un problema en términos de una jerarquía de procedimientos para la manipulación de los datos dentro de este paradigma, los datos, como entidades, claramente juegan un papel subordinado al de los procedimientos; son receptores pasivos de las manipulaciones. La

organización de un sistema computacional esta inicialmente pensada, después implantada y finalmente en términos de procedimientos.

Hoy la tendencia es hacia un desplazamiento de éste paradigma vivimos el auge de la programación orientada a objetos (POO). El objeto desplaza al procedimiento como el centro del sistema solar. La organización de un sistema gira ahora alrededor de los datos. Las entidades que manipula el programador en el proceso de diseñar, implantar y documentar sus sistemas son los objetos. No hablamos simplemente de las entidades manipuladas por el programa sino que la esencia de las entidades de la POO, o cualquier esquema de programación como paradigma, va más allá de las que manipulan los programas son las entidades, como estructuras mentales compartidas entre los profesionistas trabajando en el campo, las que hacen trascendente un esquema de programación y lo convierten en paradigma.

Tomemos un ejemplo hoy en día virtualmente todos los productos computacionales requieren de interfaces gráficas basadas en el uso de múltiples ventanas. ¿Cómo estructura el manejo de las ventanas un analista en la tradición procedimentista? Su formación, su experiencia, su interacción con sus colegas, el vocabulario (formal e informal) que utiliza: todo opera para conducirlo a plantear la solución del problema como un conjunto de procedimientos jerárquicamente organizados. Su atención está enfocada en los procedimientos. El flujo del control y, más importante, el flujo de la información están contemplados dentro del marco de referencia de una jerarquía de procedimientos. Veamos más concretamente como funciona eso. Imaginémonos una aplicación donde una acción en una ventana resulta en una modificación del contenido de otra. Supongamos, para ser más concretos, que halla una ventana, Vprov, que despliega proveedores y otra, Vpart, que despliega partes, como se ve en la figura 1-12, y que al estar señalada una referencia a una parte dentro de Vprov, Vpart se repocisiona para que aparezca la información pertinente de la parte señalada. Para un analista operando bajo el paradigma procedimentista, es natural diseñar el procedimiento, digamos Pprov, responsable para atender a las acciones dentro

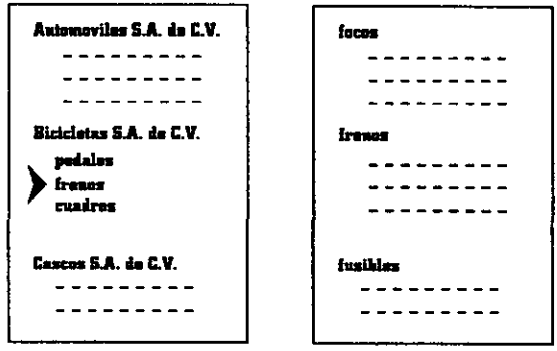


Figura 1-12 Ventanas Interrelacionadas

de Vprov de tal forma que se responsabilice de efectuar las acciones de Vpart. Y como Pprov también tenemos otros procedimientos quienes, respondiendo a sus propias exigencias, efectúan cambio en la ventana Vpart.

En el diseño del sistema completo, el manejo del contenido de Vpart esta disperso, una consecuencia de la técnica natural de descomposición de problemas que es el corazón del enfoque procedimentista. Y este aspecto del control de la ventana nunca llega a ser tan simple como parece a primera vista. Tal vez el sistema permita que el usuario "congele" una ventana y Pprov tiene que averiguar que Vpart no se encuentra en esta condición antes de reposicionarla tal vez la ventana se encuentra "iconizada" y reposicionarla debe diferirse y Pprov debe registrar que el reposisionamiento está pendiente, en vez de efectuarlo en este momento. Y, tal vez, caracteristicas como estas no fueron contempladas en el diseño inicial posiblemente ni la implantación inicial del sistema, sino que fueron incorporadas como resultado de la evolución que acompaña el desarrollo de un sistema real. (Ver figura 1-13)

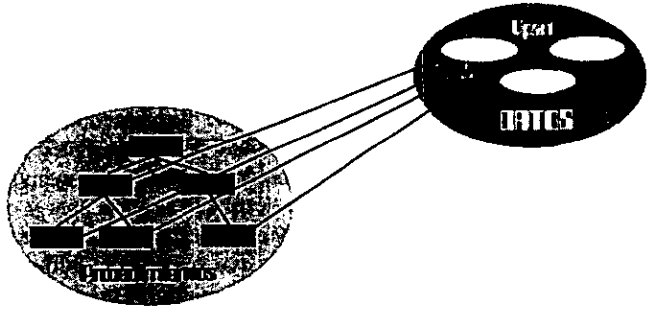


Figura 1-13 Es natural para el programador operando bajo el paradigma estructurado descomponer el problema de tal manera que el control de la ventana se encuentra disperso sobre los procedimientos del sistema.

El analista formado en la tradición de POO conceptualizará el problema de otro modo. Las entidades primordiales de su universo son objetos y la modulación conceptual está en término de ellos. Entonces, para el, las ventanas son agentes activos. Agentes que se responsabilizan por si mismo, actúan en su propio beneficio y se comunican entre si. Conceptualizando el problema así, es natural que el analista estructure su sistema del tal forma que la ventana Vprov manda un mensaje a Vpart pidiéndole que se reposicione sobre una parte especificada y su responsabilidad acaba allí. La venta Vpart efectúa las acciones correspondientes. Tal vez el objeto Vpart se encuentra congelado, algo que el mismo sabe, y no hace nada. Si se encuentra iconizado, recuerda en su propia memoria que tiene que reposisionarse al ser desplegado otra vez.

El diseño resultante es radicalmente diferente. El control de la ventana, el "conocimiento" sobre su funcionamiento, no se encuentra disperso sobretodo el sistema la

modificación de su comportamiento no requiere un análisis exhaustivo de todo el programa. De igual importancia, la codificación de cada operación es más concisa, más clara, más enfocada, más perspicaz. Un segmento de código no se distrae por preocupaciones esencialmente ajenas a su propósito principal. Un diseño que proviene de la programación estructurada resultaría en un procedimiento ostensiblemente dedicado al manejo de la ventana Vprov. "ensuciado" por la codificación del manejo de la ventana Vpart. Y por menores y relevantes (desde el punto de vista del manejo de Vprov) como son el congelamiento, la iconización, etcétera.

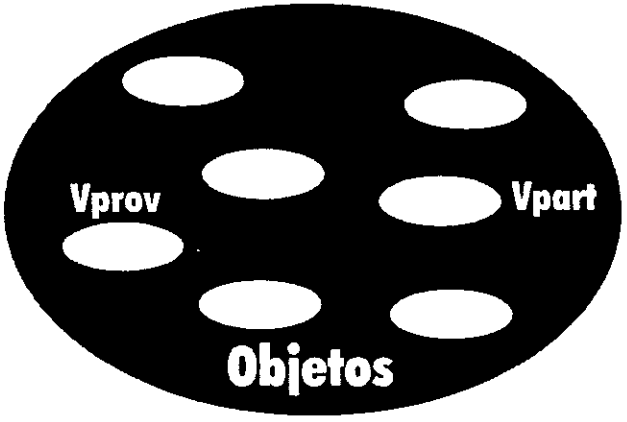
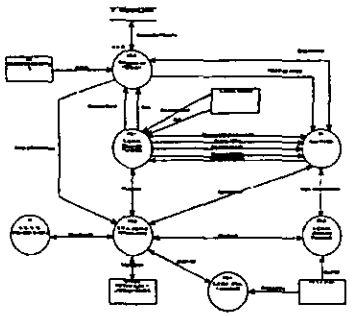
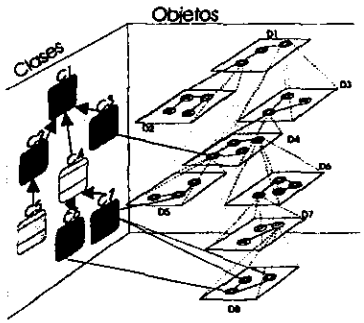


Figura 1-14 Es mas natural que un programador operando bajo el paradigma OO conceptualice el problema como objetos ventanas comunicandose entre sí. El resultado es que el control dela ventana está localizado en un solo lugar.

CAPITULO II



METODOLOGIA DE ANALISIS Y DISEÑO O.O.



COMPLEJIDAD.

En el capítulo anterior se mencionó que la creación software es compleja. Pero ahora veremos ¿Qué es complejidad?, ¿Que relación tiene la complejidad con el software? y ¿Cómo el paradigma orientado a objetos esta involucrado? Así como ¿Por que es necesario una metodología?, ¿Como se logra la comunicación entre desarrolladores?y el planteamiento de una metodología de Análisis y Diseño Orientado a Objetos (AOO y DOO).

¿POR QUE EL SOFTWARE ES INHERENTEMENTE COMPLEJO?

Una estrella moribunda cerca del colapso, un niño aprendiendo como leer, leucocitos apresurándose para atacar un virus; Son solamente unos cuantos objetos del mundo físico que involucran realmente asombrosa complejidad.

El software también comprende elementos de gran complejidad; No obstante, la complejidad que encontramos en el software es de tipo fundamentalmente diferente.

Partiremos de la definición de complejidad tomada del diccionario filosófico abreviado . ***Complejidad.***- *Dícese de lo que se compone de elementos diversos.- en sistemas.- Sistema compuesto de elementos distintos que guardan entre si relaciones determinadas constituyendo un todo cerrado y autónomo.*(ver figura 2-1).

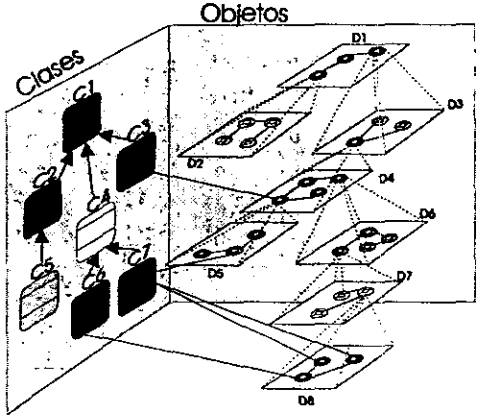


Figura 2-1 Representación Canonica de la Complejidad

Nosotros hemos realizado algunos sistemas de software que no son complejos. Estos son las aplicaciones que fácilmente se pueden olvidar ya que son especificadas, construidas, mantenidas y usadas por la misma persona. Usualmente los programadores novatos o los desarrolladores profesionales trabajan aislados, comúnmente en las aplicaciones antes mencionadas no siendo así, en el desarrollo de aplicaciones de gran escala . Esto no quiere decir que todos los sistemas semejantes son ordinarios y sin elegancia, ni queremos menospreciar a sus creadores sino que; tales sistemas tienden a tener un muy limitado propósito y un muy corto periodo de vida.

Podemos disponer de tirarlos lejos y reemplazarlos con software completamente nuevo, en vez de tratar de reusarlos, de repararlos o tratar de extender su funcionalidad. Tales aplicaciones son generalmente más tediosas, ya que dificultan su desarrollo, consecuentemente el aprender como diseñarlas no es de nuestro interés.

En cambio nos interesamos mucho mas por los retos de desarrollo que llamaremos **software Industrial-Fuerte (o de gran escala)**; Aquí encontramos aplicaciones que exhiben un muy rico conjunto de comportamientos como por ejemplo:

Sistemas reactivos que son manejados por eventos del mundo físico y para cualquier tiempo o espacio son de escasos recursos, las aplicaciones que mantienen la integridad de cientos de miles de registros de información mientras permiten actualizaciones y búsquedas concurrentes; sistemas para el comando y control de entidades del mundo real, tales como el enrutamiento del trafico aéreo o ferroviario. Los sistemas como estos tienden a tener un periodo de vida largo y horas extras de uso, muchos usuarios llegaran a depender de su propio funcionamiento.

Las características distintivas del software Industrial-Fuerte, es que tal software es intensamente difícil, si no es que imposible, para desarrollar individualmente, y comprender todas las sutilezas de dicho diseño.

Hablando en términos francos, la complejidad de tales sistemas excede la capacidad intelectual humana, esta complejidad de la que hablamos parece ser una propiedad esencial. Por esencial queremos decir que podemos lograr dominar esta complejidad, pero no podremos jamas quitarla.

Como sugiere Booch *"La complejidad del software es una propiedad esencial y no una accidental"* por lo que anteriormente mencionamos. Podemos por lo tanto concluir que el software es inherentemente complejo. Observemos que esta complejidad inherente se deriva de 4 elementos:

- La complejidad del domino del problema
- La dificultad de manejar el proceso de desarrollo
- La flexibilidad posible a través del software
- Los problemas de la caracterización del comportamiento de los sistemas discretos

LA COMPLEJIDAD DEL DOMINIO DEL PROBLEMA.

Los problemas que tratamos de modelar y solucionar con el software frecuentemente involucran elementos de inevitable complejidad, en los cuales encontramos innumerables requerimientos quizás todavía contradictorios. por ejemplo: consideremos los requerimientos para un sistema que controle las partes electrónicas de una aeronave, un sistema de cambiado de telefonía celular, o un robot autónomo. El crudo funcionamiento de tales sistemas es lo suficientemente difícil para comprenderlos.

La complejidad usualmente surge desde el mal emparejamiento de ideas que existe entre los usuarios del sistema y los desarrolladores de éste. Los usuarios generalmente encuentran muy difícil de dar expresiones precisas de sus necesidades en una forma que los desarrolladores puedan entender. En casos extremos los usuarios podrían tener solo ideas vagas de que es lo que ellos quieren de un sistema. Esto ocurre, por que en cada grupo generalmente escasean expertos en el dominio de lo demás, es decir un desarrollador no es experto en el problema a atacar y el usuario no es experto en el desarrollo de sistemas, por lo tanto los usuarios y los desarrolladores tienen diferentes perspectivas sobre la naturaleza del problema y hacen diferentes suposiciones con respecto a la naturaleza de la solución. es aquí donde interviene el paradigma orientado a objetos; ya que hay que recordar que en dicho paradigma el usuario final piensa también en términos de objetos, permitiendole modelar y expresar mas claramente sus ideas y percepción de la realidad así como de sus necesidades a los desarrolladores.

LA DIFICULTAD DE MANEJAR EL PROCESO DE DESARROLLO.

La tarea fundamental de un equipo de desarrollo de software es hasta crear la ilusión de simplicidad para proteger a los usuarios de la complejidad; es asi como, en el contexto de la implementación de las aplicaciones computacionales, toman gran importancia los términos computacionales hacia el usuario: Virtual y Transparente.

La orientación a objetos de forma natural nos forza a escribir menos código por ingeniosa inventiva y poderosos mecanismos que nos den la ilusión de simplicidad como por ejemplo usando esqueletos de diseños existentes (Metaclases, Clases abstractas, etc.).

Como ya se mencionó anteriormente una sola persona no puede siempre entender completamente cada sistema, aun si descomponemos nuestra implementación en vías significativas, sin embargo finalizaremos siempre con cientos y algunas veces con miles de módulos. ***Por lo que dicho trabajo requiere de un equipo de desarrolladores y una metodología para coordinar el trabajo así como la comunicación entre desarrolladores.***

LA FLEXIBILIDAD POSIBLE A TRAVES DEL SOFTWARE.

Esto se refiere a que los desarrolladores pueden usar diversas herramientas para expresar sus abstracciones mientras que por ejemplo en la industria de la construcción hay códigos y estándares para calificar la materia prima; pocas veces esos estándares existen en la industria del software, como resultado de lo anterior el desarrollo del software queda con una labor intensiva de negociación

LOS PROBLEMAS DE LA CARACTERIZACION DEL COMPORTAMIENTO DE LOS SISTEMAS DISCRETOS.

Si lanzamos una bola al aire, podremos confiablemente predecir el camino, por que conocemos que bajo condiciones normales se aplican ciertas leyes físicas, nos sería muy sorpresivo si por sólo arrojar la bola un poco más fuerte, en el vuelo a la mitad del camino se detuviera súbitamente y se disparara extrañamente en el aire. En un software de simulación de este movimiento de bola no totalmente depurado; exactamente ese tipo de comportamiento podría ocurrir fácilmente.

Dentro de una aplicación grande, hay quizás cientos o aún miles de variables, además de un hilo de control. La colección entera de esas variables, sus valores actuales y las direcciones actuales de memoria, así como el llamado en pila de cada proceso dentro del sistema constituye el estado presente de la aplicación. Debido a que ejecutamos nuestro software sobre computadoras digitales, tenemos como resultado sistemas con estados discretos; los sistemas discretos por naturaleza tienen un número finito de estados, en sistemas grandes esta es una explosión combinatorial esto hace que ese número sea muy grande.

Nosotros debemos tratar de diseñar nuestros sistemas con una separación de intereses ya que el comportamiento en una parte del sistema tenga un mínimo impacto sobre el comportamiento de otro. De cualquier modo el resto del hecho, es que la fase de transición entre estados discretos no pueden ser modelados por funciones continuas, cada evento externo para un sistema de software tiene el potencial de colocar ese sistema en un nuevo estado, y además el mapeo de estado a estado no siempre es determinístico; en las peores circunstancias, un evento externo podría corromper el estado del sistema (**es decir llevarlo de la complejidad a la perplejidad**). Esto sucede por que los diseñadores fallaron al tomar en cuenta ciertas interacciones entre eventos. Por ejemplo, imaginemos un avión comercial que vuela sobre la superficie y el ambiente de la cabina es manejado por una sola computadora; seriamos muy infelices si como resultado de un pasajero en el asiento 38-J; prendiera fuego sobre su cabeza y el avión inmediatamente ejecutara una repentina bajada en picada. En un sistema contínuo este comportamiento sería improbable, pero en un sistema discreto todos los eventos externos pueden afectar cualquier parte del estado interno del sistema ciertamente este es el principal motivo para hacer pruebas fuertes a nuestros sistemas, pero para todos excepto, la mayoría de los sistemas triviales es imposible hacer

pruebas exhaustivas ya que no tenemos ninguna herramienta matemática, ni la capacidad intelectual de modelar el comportamiento completo de un gran sistema discreto, deberemos entonces conformarnos con niveles aceptables de confianza con respecto a su exactitud.

Centrándonos nuevamente en el hecho de que una sola persona no puede analizar completamente la complejidad de un sistema diremos que ciertamente, siempre habrá entre nosotros gente de extraordinaria técnica o experiencia, quien puede hacer el trabajo de un puñado de simples desarrolladores mortales. El ingeniero de software equivalente a Leonardo Da Vinci. Esta es la gente a quien vemos desplegarse como nuestros arquitectos de sistemas, los primeros quienes idean innovadores lenguajes, mecanismos y armazones que otros pueden usar como fundamentos arquitectónicos de otras aplicaciones; de cualquier modo como Booch observa *"el mundo es solo habitado con genios dispersos no hay razón para creer que la comunidad de ingeniería de software tiene una gran proporción inmoderada de ellos"*.

Si bien hay un toque de genio en todos nosotros, en el reino del software industrial-Fuerte no podemos siempre contar con la inspiración divina para que podamos pensar bien por lo tanto nosotros debemos considerar vías mas disciplinadas para poder lograr dominar la complejidad. Es por esta razón por la que proponemos usar una metodología orientada a objetos.

COMUNICACION ENTRE DESARROLLADORES.

Cuando varias personas laboran en un sistema o en un programa los diagramas son la herramienta esencial para la comunicación. Una técnica formal de diagramación permite a los diseñadores intercambiar ideas y hacer que sus componentes independientes se ajusten de manera precisa.

Al modificarse los sistemas los diagramas claros pueden facilitar el mantenimiento. Con ellos, un equipo nuevo de trabajo puede entender el funcionamiento de los programas y diseñar los cambios. Tales cambios afectan con frecuencia a otras partes del sistema, sin embargo, los diagramas claros de la estructura de un sistema y de un programa, permiten a los programadores de mantenimiento comprender las consecuencias de dichos cambios. Durante la depuración los diagramas claros son una valiosa herramienta para la comprensión de la forma en que los programas deben funcionar y buscar aquello que funciona mal.

Así la diagramación es un lenguaje esencial, tanto para la claridad de pensamiento como para la comunicación. Un sistema necesita estándares para los diagramas que lo describen, así como existen estándares para los planos de ingeniería.

METODOLOGIA

El modelo de objeto nos permite encontrar los objetos que actúan en un sistema, así como sus relaciones con otros objetos. Utilizando las siguientes herramientas. Abstracción (que es la más importante) pues nos permite encontrar los objetos que se encuentran en el dominio del problema, obteniendo sus funciones (responsabilidades), para que posteriormente las funciones se oculten por medio de la encapsulación y poder manejar el objeto como una pequeña caja negra que se comunica con otros objetos por medio de mensajes, teniendo así una interface publica y ocultando la información que sólo él necesita para cumplir con sus responsabilidades.

Siguiendo hacia otros niveles de abstracción creamos clases, que son el conjunto de objetos que tienen comportamientos comunes, estos módulos llamados clases a su vez pueden tener características comunes con otras clases y formar jerarquías de clases.

En resumen podemos definir la Metodología Orientada a Objetos en los siguientes pasos:

- ABSTRAER
- ENCAPSULAR
- JERARQUIZAR
- MODULARIZAR

ABSTRAER

Cada concepto es una idea particular o nuestra comprensión del mundo. Sabemos que tenemos un concepto cuando lo podemos aplicar con éxito a las cosas que nos rodean por ejemplo, decir que tenemos el concepto de automóvil solo requiere que podamos identificar una instancia de dicho concepto. Finalizando un concepto es una idea o noción compartida que se aplica a determinados objetos en forma consiente.

La formación de conceptos nos ayuda a ordenar nuestras vidas. Los psicólogos han propuesto que cuando un bebé inicia su vida, su mundo es una confusión. A edad temprana, desarrolla el concepto de ser alimentado. Poco después, aprende a distinguir los sonidos maternos y a reconocer los distintos objetos que ofrece nuestro mundo, en cierto momento, desarrollamos conceptos tales como azul y cielo y aprendemos a combinar conceptos para formar otros nuevos, como cielo azul, al crecer aún más, hacemos construcciones conceptuales que producen significado, precisión y sutilezas. Por ejemplo el cielo es azul solo en días claros; o bien, el cielo no es realmente azul, solo parece azul en nuestro planeta tierra debido a efectos atmosféricos. Los conceptos que utilizamos pueden variar, puesto que nosotros los elegimos. Los conceptos pueden ser concretos (por ejemplo: persona, lápiz, automóvil), intangibles (por ejemplo: tiempo, calidad, compañía), papeles o roles (por

ejemplo: doctor, paciente, propietario), juicios (por ejemplo: trabajo productivo, salario alto, buen ejemplo), por relación (por ejemplo: matrimonio, sociedad, propiedad) eventos (por ejemplo: venta, compra, falla del sistema etc.)

Los conceptos que adquirimos nos proporcionan una especie de lentes mentales con los que tomamos conciencia y razonamos acerca de los objetos de nuestro mundo, por ejemplo, al poseer el concepto persona podemos razonar acerca de todos o parte de miles de millones de objetos en la tierra. Los conceptos forman nuestra percepción de la realidad, cuando utilizamos un concepto, no servimos de pruebas que determinan si éste se aplica o no a los objetos que nos rodean. Así, cada concepto se basa en pruebas que determinan si se aplica o no. Los objetos que pasan la prueba se realizan como una instancia de un concepto. De esta forma los objetos, se pueden percibir de muchas maneras, de acuerdo a nuestras pruebas conceptuales. Por convención, las ideas o los conceptos poseídos de manera privada se llaman concepciones. Si la comprensión es compartida por varias personas se convierte en un concepto.

El Significado de Abstracción. La abstracción es una de las maneras fundamentales, que nosotros como humanos usamos para hacerle frente a la complejidad. Hoare sugiere que "la abstracción proviene de un reconocimiento de similitudes entre ciertos objetos, situaciones, o procesos en el mundo real, y la decisión de concentrarse en estas similitudes y de ignorar por ahora las diferencias". Shaw define una abstracción como una " descripción simplificada, o especificación, de un sistema que enfatiza algunos de detalles de sistema o propiedades mientras suprime otros. *Una abstracción buena es aquella que enfatiza los detalles que son importantes al lector o al usuario y suprime detalles que son, por lo menos por el momento, sin importancia o diversos*".

Berzins, Gris y Naumann recomiendan que "un concepto califica como una abstracción solo si puede describirse, entenderse, y analizarse independientemente del mecanismo que eventualmente se usará para darse cuenta de él" Combinando estos puntos de vista diferentes, nosotros definimos una abstracción como se indica a continuación:

Una abstracción denota las características esenciales de un objeto que lo distinguen de toda clase de objetos y así definir claramente limites conceptuales, relativos a la perspectiva del espectador.

Una abstracción enfoca la vista externa de un objeto, y así servir para separar el comportamiento esencial de un objeto de su implementación. Abelson y Sussman llaman a esto comportamiento/división de implementación una barrera de abstracción lograda por aplicar el principio del compromiso menor, mediante el cual la interface de un objeto provee su comportamiento esencial, y nada más. Nos gusta usar un principio adicional que nosotros llamamos el principio del menor asombro, mediante el cual una abstracción captura todo el comportamiento de algún objeto, no más y no menos, y así ofrecer que no haya sorpresas o efectos colaterales que van más allá del alcance de la abstracción.

PROCESO DE ABSTRACCION

Abstracción en el dominio del problema.- Se lleva a cabo con los siguientes pasos:

- Identificar Objetos (que se derivan directamente del dominio del problema)
- *Clasificar objetos similares y sus comportamientos*

IDENTIFICANDO OBJETOS: Analizando el dominio del problema mediante la abstracción encontramos los objetos que actúan en dicho dominio. Este análisis depende en gran medida de la habilidad o experiencia del analista para obtener los objetos y sus características que den una solución al problema. Para generar o implementar objetos primero deben existir las clases pero para que existan las clases primero hay que definir los modelos de objetos.

PROCESO DE IDENTIFICACION DE OBJETOS:

1.- Se debe tener un objetivo

2.- Buscar los objetos que satisfacen los requerimientos de los objetivos

Por ejemplo en una interface gráfica se pueden identificar los siguientes objetos.

a) Estructuras de datos: Tales como listas ligadas (simple y dobles), arboles, colas, pilas, buffer. Ya que se pueden encapsular las operaciones necesarias para procesar la estructura de datos con los datos almacenados en la misma estructura.

b) Los componentes de interface de usuario como: ventanas, menús, botones, cuadros de diálogos, barras, iconos. En este tipo de componentes se puede utilizar la herencia.

c) Componentes que simulan objetos del mundo real o eventos tales como: manometros, gráficas, mapas, gente, etc. también pueden ser considerados como objetos.

d) Componentes de hardware como el ratón, el teclado y la pantalla.

e) Algoritmos que se pueden representar como objetos por ejemplo la creación de un objeto para búsqueda, o un analizador de sintaxis general que derive diferentes analizadores a partir del general para manejar lenguajes diferentes.

Y en general todos los objetos que actúan en el dominio del problema

El icono que utilizaremos para representar un objeto es el siguiente: Figura 2-2

Continuando con el ejemplo de la interface gráfica los objetos quedarían representados de la siguiente manera: (Ver Figura 2-3)

El producto que se obtiene al modelar la abstracción de objetos se le llama: Diagrama de Objetos. Un solo diagrama de objetos puede entonces representar un instante en el tiempo

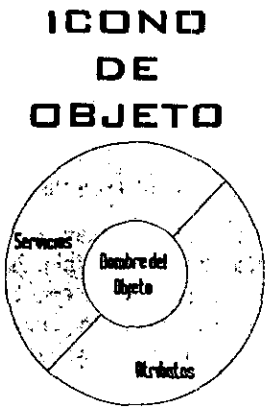


Figura 2-2 Icono de Objeto

transcurrido de un evento transitorio. En este sentido los diagramas de objetos son prototipos, cada uno representa las interacciones que pueden ocurrir entre una colección de

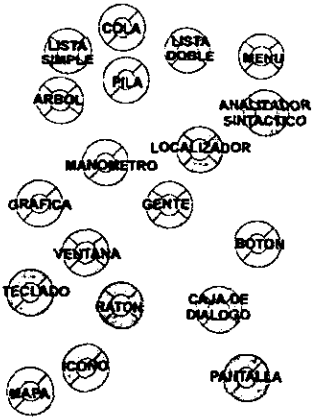


Figura 2-3 Ejemplo de algunos objetos existentes en una GUI y su reprensentacion usando el icono de la metodologia

objetos sin importar que instancias nombradas especificamente participan en el mecanismo. Este diagrama se complementara con el diagrama de transición de estados y el escenario que en secciones posteriores se veran.

Las operaciones utilizadas en un diagrama de objetos deben ser consistentes con las operaciones definidas en las clases por ejemplo si mostramos un objeto R enviando el mensaje M al objeto S, entonces la operación M debe estar definida en la clase del objeto S. Los diagramas de clase documentan las abstracciones clave en nuestro sistema y los diagramas de objetos resaltan los mecanismos importantes que manipulan estas abstracciones.

Los elementos más importantes de un diagrama de objetos son los objetos y las relaciones entre objetos. Los nombres de los objetos no necesariamente deben ser específicos pero si deben ser representativos de la abstracción, las propiedades pueden aparecer en el centro del icono del objeto.

Relaciones entre objetos.- una relación entre dos objetos simplemente significan que los objetos pueden enviarse mensajes el uno al otro. Ver Figura2-4.

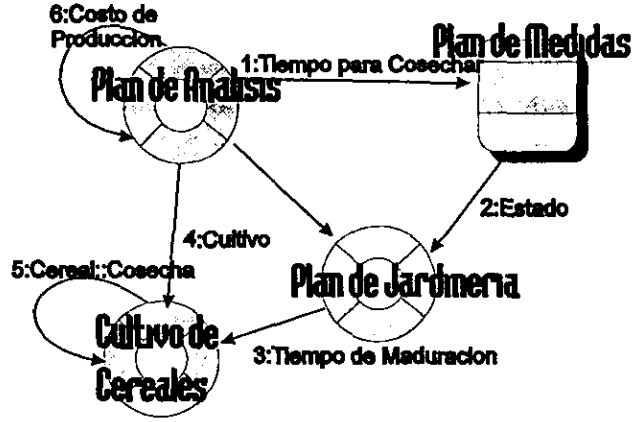


Figura 2-4 Ejemplo de un Diagrama de Objetos de Cultivo Hydroponico

En esta figura se muestra el diagrama de objetos de un sistema de Cultivo hydroponico

CLASIFICACION

La clasificación es el medio por el cual nosotros ordenamos el conocimiento. En el diseño orientado a objetos, reconocemos la igualdad entre cosas esto nos permite exponer el estado común dentro de mecanismos y abstracciones claves, y eventualmente nos conduce a arquitecturas menores y más simples. Desafortunadamente, no hay trayectoria dorada a la clasificación. Para el lector acostumbrado a encontrar respuestas de libro de cocina, nosotros inequívocamente reconocemos que no hay recetas simples para identificar clases y objetos. No hay tal cosa como la estructura de clase "perfecta", ni los "derechos" de un conjunto de objetos. Como cualquier disciplina de ingeniería, nuestras elecciones de diseño son un compromiso formado por muchos factores compitiendo entre ellos.

En una conferencia sobre ingeniería de software, varios desarrolladores fueron cuestionados acerca de que reglas ellos aplicaron para identificar clases y objetos. Stroustrup, el diseñador de C++, respondido "Es un Santo Grial. No hay Panacea". Gabriel, uno de los diseñadores de CLOS, respondió, "Que es una pregunta fundamental para la cual no hay respuesta fácil. Yo trato cosas". Afortunadamente, existe un legado extenso de experiencia con la clasificación en otras disciplinas. Desde enfoques más clásicos, las

técnicas de análisis orientadas a objeto han surgido y ofrecen varias prácticas recomendadas y reglas básicas útiles, para identificar las clases y objetos pertinentes a un particular problema. Estas Heurísticas son foco de esta sección.

CLASIFICANDO OBJETOS: Como se menciono en el capitulo anterior Clasificar es la manera de ordenar conforme a ciertas relaciones existentes entre objetos

PROCESO DE CLASIFICACION:

1.-Definir una semantica (Seleccionar un criterio de clasificación)

2.-Identificar los elementos de un conjunto dado, bajo la semantica establecida.

3.-Agrupar los elementos que definen una unidad homogenea (definir las clases)

FORMAS DE CLASIFICACION

- **Funcional.**- Clasificar los objetos de manera funcional según G. Booch dice que solo hay tres tipos de objetos
  - Servidores (Son operados por otros objetos)
  - Actores (Son los que operan a otros objetos y nunca son operados)
  - Agentes (Pueden operar como ser operados por otros objetos)
- **Ontológica.**- Es la clasificación que se realiza buscando lo que proviene de un mismo módulo
- **Propiedades físicas.**- Extensión, dureza, masa, peso, porosidad, etc.
- **Estructural.**- Es la clasificación que se realiza buscando las relaciones entre sus elementos,por orden o configuración
- **Morfológica.**- Es la clasificación por tamaño, tipo, forma o figura
- **Alfabética.**- Vocales, consonantes
- **Geográfica.**- Clasificación por regiones
- **Ideológica.**- Clasificar por pensamientos (religión, política)
- **Nemotécnica.**-Clasificar por contenido
- **Númerica.**- Clasificación por ciertas propiedades matemáticas (reales, enteros)
- **Cronológica.**- Clasificar por tiempo de vida de algo o de un proceso.
- **Alfanumérica.**- Incluye la clasificación numérica y alfabética

Siendo las clasificaciones más importantes en el paradigma orientado a objetos: la funcional, la de propiedades fisicas, ontológica, estructural y morfológica.

Para que sea útil la clasificación debe cumplir las siguientes propiedades: ser consistente(las categorías distributivas serán excluyentes entre si) y completa (la suma de sus partes distribuidas será igual al todo).Ver figura 2-5

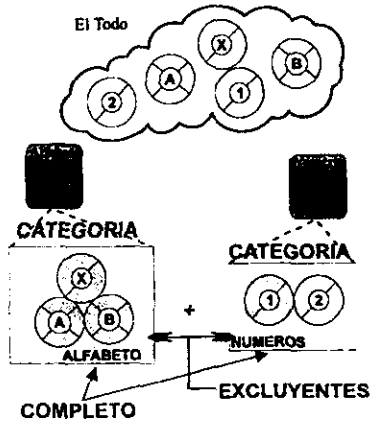


Figura 2-5 Propiedades que debe cumplir una Clasificacón para que sea buena y util.

El icono que utilizaremos para representar un clase es el siguiente: Ver Figura 2-6

Figura 2-6 Icono de Clase

Un diagrama de clases es utilizado para mostrar la existencia de clases y sus relaciones en el diseño lógico de un sistema los tres elementos más importantes de una estructura de clases son: las clases, las relaciones entre clases y las utilidades de clases (para lenguajes que soportan la declaración de subprogramas libres.

Si el nombre de la clase es muy largo debe ser acortado o aumentado el tamaño del icono; El nombre de la clase debe ser único dentro de la categoría de clases.

Relación los tipos de relación entre clases son: La herencia, la instanciación y relación entre metaclases.

En algunas circunstancias especialmente en el modelado de clases que forman parte de una base de datos existe un gran valor en mostrar la cardinalidad entre clases que se utilizan la una a la otra, para cada clase en un diagrama de clases también tenemos una plantilla de clases, una plantilla de clases captura todos los aspectos importantes. Esta plantilla es lo suficientemente detallada pero en fases tempranas del análisis no se espera que sea llenada completamente sino conforme vaya evolucionando el diseño puede ir siendo llenada.

Componentes: **Visibilidad.**-Iindica si la clase es importada, exportada o privada con relación a la categoría de la clase que la contempla.

Cardinalidad.- Especifica cuantas instancias permitirá típicamente los valores de este elemento son: 0/1/n en casos especiales se utiliza rangos específicos.

Jerarquía.- Tiene dos subcomponentes que son: superclase lista de los nombres de la clase o metaclase

Parámetros genéricos.- Si el lenguaje lo permite los tres siguientes campos se repiten hasta cuatro veces tres veces para la interface y uno para la implementación de la clase, si el lenguaje de implementación lo permite puede ser dividida en pública protegida y privada. En cualquier caso esta es la sección más importante de la plantilla de clase porque es aquí donde capturamos la vista exterior de la clase. Esta vista externa incluye cualesquiera variable de la instancia y variables de clase documentadas en los campos de elementos así como todas sus operaciones.

Atributos.- Para cada campo, podemos documentar su nombre, si es una constante o variable su clase y cualesquiera restricción de dominio (por ejemplo L es un entero cuyo valor está restringido para estar en el rango de uno a cien cuando es inicializado el campo).

Operaciones(Servicios).- Es la lista de operaciones.

Acontinuacion un Ejemplo muy simple de un Diagrama de Clases Ver Figura2-7

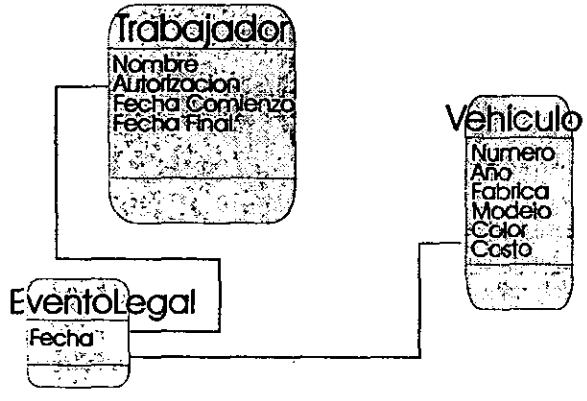


Figura 2-7 Ejemplo de un Diagrama de Clases

Un diagrama de objetos es utilizado para mostrar la existencia de objetos y sus relaciones en el diseño lógico de sistemas. Un solo diagrama de objetos representa todo o parte de la estructura de los objetos de un sistema típicamente el diseño de un sistema requiere de un conjunto de diagramas de objetos. El propósito de cada diagrama de objetos es ilustrar las semánticas de mecanismos clave en el diseño lógico. Las clases por lo regular son más estáticas en comparación con los objetos que son más transitorios.

ENCAPSULACION

El Significado de Encapsulación lo definimos anteriormente. La abstracción de un objeto debe preceder a la decisión sobre su implementación. Una vez seleccionada la implementación, se debe tratar como un secreto de la abstracción y ocultar la implementación a los clientes. La abstracción y encapsulacion son conceptos complementarios: la abstracción enfoca los comportamientos sobresalientes de un objeto, mientras que la encapsulación se enfoca en la implementación que da origen a estos comportamientos. La encapsulación es lograda mediante el ocultamiento de información, que es el proceso de ocultar todos los atributos y comportamientos de un objeto que no contribuye a sus características esenciales; típicamente, la estructura de un objeto se oculta, así como también los implementación de sus métodos.

."Mientras la abstracción ayuda a la gente a pensar sobre lo qué están haciendo, la encapsulación permite cambios confiables a programas con un mínimo de esfuerzo" .

La encapsulación provee barreras explícitas entre abstracciones diferentes y estas barreras permiten una separación clara de intereses. Por ejemplo

Para entender como la fotosíntesis trabaja a un alto nivel de abstracción, nosotros podemos ignorar detalles tales como el funcionamiento de las raíces de la planta o la química de las paredes celulares. De igual forma, en el diseño de una aplicación de base de datos, es práctica común escribir programas que no cuiden de la representación física de los datos, pero si, que dependan solamente de un esquema lógico de datos. En ambos casos, los objetos a un cierto nivel de abstracción son protegidos de los detalles de implementación en los niveles más bajos de abstracción.

En la práctica, esto significa que cada clase debe tener dos de partes: una interface y una implementación. La interface de una clase captura únicamente la vista externa, esta abstracción comprende el comportamiento común de todas las instancias de la clase.

La implementación de una clase comprende la representación de la abstracción así como también los mecanismos que logran el comportamiento deseado. La interface de una clase es un lugar donde declaramos todas las suposiciones que un cliente puede hacer acerca de cualquier instancia de la clase (atributos). Y en la implementación se encapsulan los detalles acerca de los cuales el cliente no puede hacer suposiciones. *En una implementación en el paradigma O.O que se considera típicamente bien hecha, es una clase que generalmente deberá ser pequeña, por que se puede volver a utilizar en los recursos de otras clases*

por medio de mecanismos como la agregación o la herencia. También debemos proveer operaciones y metaoperaciones, específicamente, constructores y destructores, operaciones que inicializen y destruyen instancias de estas clases, respectivamente.

La capacidad para cambiar la representación de una abstracción sin perturbar cualquiera de sus clientes es el beneficio principal de la encapsulación. C++ ofrece un control aun más flexible sobre la visibilidad de miembros, objetos y funciones miembro. Específicamente, los miembros pueden ponerse en tres secciones pública, privada, o protegida en la definición de una clase. Los miembros declarados en las partes públicas son visibles a todos los clientes; los miembros declarados en las partes privadas son totalmente encapsulados, y los miembros declarados en las partes protegidas son visibles únicamente a la clase misma y a las subclases.

El encapsulamiento es un concepto relativo, ya que los miembros qué se pueden ocultar a un nivel de abstracción pueden representarse como vista externa a otro nivel de abstracción. Es decir no basta con saber que existe 3 secciones en el lenguaje, ya que esto no implica que se vayan colocar los miembros en la sección adecuada para la clase

PROCESO DE ENCAPSULACION

El proceso de encapsulación es el proceso más largo debido a que se realiza la definición para concretar y complementar las clases y objetos antes abstraídos, además se modela la parte dinámica de los sistemas dicho proceso se lleva a cabo definiendo los siguientes elementos.

- ¿Que es un atributo?
- Definición de atributo
- Pasos para la abstracción de atributos
- ¿Como se identifican o establecen?
- Identificar instancias de conexión (Cardinalidad o Asociación)
- Buscar servicios
- Relación entre objetos y servicios en base a la función (Asociación)
- Modelar los ciclos de vida del objeto

Atributo.- Es una propiedad, cualidad o característica que puede ser asignada a una persona o cosa. Es una abstracción de una sola característica, o algunos datos (información de estado) la cual es común a todas las entidades que fueron abstraídas como objeto, para que cada objeto en una Clase tenga su valor propio.

Nuestro modelo AOO ahora es más específico y más detallado. Los atributos describen más a detalle, una Clase y un Objeto especificandolo. ¿Por que hacer énfasis en los atributos? Hay que hacer énfasis, debido a que los servicios utilizaran estos atributos (información de estado), ya que los atributos describen valores o estados guardados dentro de un objeto para ser exclusivamente manipulados por los servicios de ese objeto; Los atributos deberán ser tratados como partes privadas de la especificación total del objeto, si

otra parte del sistema necesita acceder o manipular los valores de un objeto tendrá que hacerlo a través de un mensaje que establezca una conexión a un servicio definido.

COMO IDENTIFICAR ATRIBUTOS.: Para identificar atributos se deberán aplicar las siguientes preguntas, estas preguntas deberán ser hechas desde la perspectiva de un solo objeto.

- ¿Como se describe el objeto en general?
- ¿Como se describe el objeto dentro del domino del problema?
- ¿Como se describe en el contexto dentro de las responsabilidades del sistema?
- ¿Que necesito saber?

Seguido por:

- *¿Como se comporta el objeto en general?*
- ¿Que información de estado necesito recordar a través del tiempo?
- ¿En que estado puede estar el objeto?
- ¿Que interacciones ocurren entre objetos?

Las anteriores preguntas servirán para modelar la parte de comportamientos que se vera en una sección mas adelante.

Algunos atributos serán bastante directos por ejemplo los atributos posibles para una clase supervisor serán nombre, dirección, titulo,etc. Para una aeronave los atributos posibles serán altura, velocidad, plan de vuelo,etc. Se deberá poner el atributo que mejor describe al objeto de acuerdo al domino del problema; hay que hacer notar que las responsabilidades del objeto dentro del domino del problema dictara los atributos apropiados para ese objeto. Por ejemplo la clase automóvil puede ser especificada de diversas formas dependiendo del domino del problema; si se observa desde el dominio del problema del automovilismo, del comercio, del histórico, o del coleccionista, etc.

Haga cada atributo un **concepto atómico** lo que significa tener un valor único o una agrupación estrecha de valores. Este concepto puede ser un elemento informático individual por ejemplo numero de licencia, RFC, edad, peso, etc; o puede ser un grupo natural de elementos informáticos (p. ej., nombre legal: compuesto por nombre, apellido paterno y apellido materno; o dirección: compuesto de calle, ciudad, estado, código postal, y país), el motivo para expresar un concepto atómico está en producir un modelo más simple para la revisión humana, con pocos nombres de atributos, y agrupaciones naturales de datos es más fácil para su asimilación. Los atributos se deberán poner de la siguiente forma: Por ejemplo la especificación de un sensor se vería como en la figura 2-8

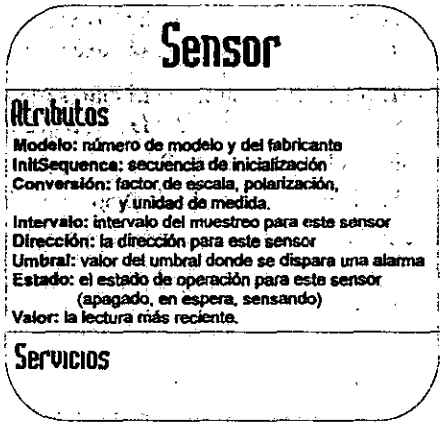


Figura 2-8 Especificación de la Clase Sensor.

Si en diagrama de clases no alcanza debido a que la abstracción de la clase es muy grande se deberá usar una plantilla de clase para documentar el resto a continuación se muestra la plantilla ver figura 2-9

PLANTILLA PARA CLASE

CLASE: NOMBRE DE LA CLASE
DOCUMENTACIÓN: TEXTO
VISIBILIDAD: IMPORTADA, EXPORTADA O PRIVADA.
JERARQUIA: NOMBRE DE LA SUPERCLASE
PARAMETROS GENERICOS: PUBLICA PROTEGIDA Y PRIVADA
ATRIBUTOS: TODAS LAS CARACTERISTICAS O CAMPOS
SERVICIOS: TODOS LOS COMPORTAMIENTOS

Figura 2-9 Propiedades que debe cumplir una Clasificacón para que sea buena y util.

TIPOS DE ATRIBUTOS

Los atributos que se encuentran mas a menudo en los objetos del mundo real son de tres tipos:

- **Descriptivos**.-Son los que proveen hechos intrínsecos. p. ej. **Gato**.peso, **Bateria**.polaridad, Informe.balance.
- **De Nombramiento**.-Sirven para "etiquetar" (identificar) objetos p.ej. **Informe**.numero, **Parte**.no_serie.

– **Referenciales.**-Se usan para encadenar instancias de objetos diferentes. P.ej. **Gato**.nombre_propietario, **Informe**.clave_contador, Avión_piloto.Los Atributos de Identificación proveen medios convenientes de referenciar un objeto y sus conexiones a otros objetos.

Nota: *los nombres en negrilla son los objetos y los subrayados son los atributos.*

INSTANCIAS DE CONEXION (CARDINALIDAD O ASOCIACION).

Los atributos retratan el estado de un objeto. Las instancias de conexión agregan a esa información, la información necesaria para que un objeto cumpla sus responsabilidades. Las Instancias de Conexión modelan la asociación, que es un principio para administrar la complejidad. Webster's define "asociación" como sigue: **Asociación.- Es la unión o conexión de ideas.**

La gente utiliza la asociación para juntar ciertas cosas que suceden en algún punto en el tiempo o bajo circunstancias similares. Para AOO, el término "Instancia de Conexión o *también llamado* restricciones de cardinalidad "se define como un modelo que refleja la representación en el dominio del problema, de la información que un objeto necesita de otro objeto, *a fin de cumpla sus responsabilidades;* Dicho de una manera mas coloquial, se refiere a la restricción de la cantidad de elementos que se pueden asociar con otro, Por *ejemplo una cardinalidad puede restringirse* como una cardinalidad uno a uno, o bien uno a muchos. En ciertos casos se pueden utilizar números para indicar los limites superior e *inferior de la cardinalidad.*

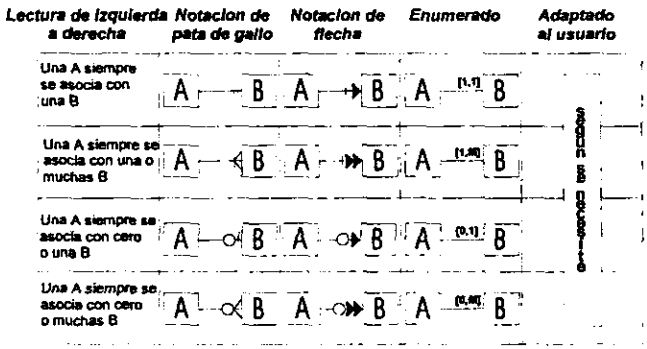


Figura 2-10 Notaciones Comunes para la Cardinalidad

Una instancia de conexión se representa con una línea dibujada entre objetos. Los puntos finales de una línea de instancia de conexión pueden representarse de varias formas como en la figura. 2-10

Cada instancia de conexión deberá estar etiquetada en un esquema de objetos, por lo que se debe tener una notación uniforme. La técnica de etiquetado que se usara en esta tesis es la siguiente. Cuando se representa una instancia que asocia un nodo de la izquierda con un nodo de la derecha, la etiqueta aparecerá sobre la línea horizontal, al leerse se hará de izquierda a derecha, si la etiqueta se posiciona abajo se leerá de derecha a izquierda. Al

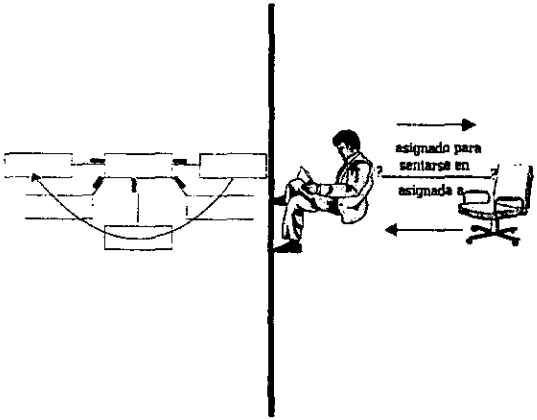


Figura 2-11 Etiquetado de una Instancia de Conexión durante la Rotación y Forma de Lectura de la etiqueta

rotar la línea la etiqueta permanece del mismo lado de la línea como se muestra en la figura 2-11. Así la etiqueta a la derecha de una línea vertical se lee cuando la función realiza la asociación hacia abajo en la línea, la etiqueta a la izquierda de la línea vertical se lee cuando la asociación es hacia arriba en la lineal.

Para formalizar la manera en que se hacen estas asociaciones, empezaremos por definir algunos términos o conceptos que se utilizaran:

- **Función.-** Es un proceso de asociación que, dado un objeto dentro de un conjunto, regresa un conjunto de objetos de un tipo determinado.
- **Dominio.-** El dominio de una función es la colección de todos los objetos asociados por la función.
- **Rango.-** El rango de una función es la colección de todos los objetos con los que se asocia la función.
- **N-ada.-** Es una relación formada por una composición inmutable de varios objetos.

En la figura 2-12 se puede observar un claro ejemplo de una instancia de conexión al encontrar la función entre las dos clases, ya que para definir una asociación se necesita una función. La función **emplea** asocia cada objeto de la clase Organización con un

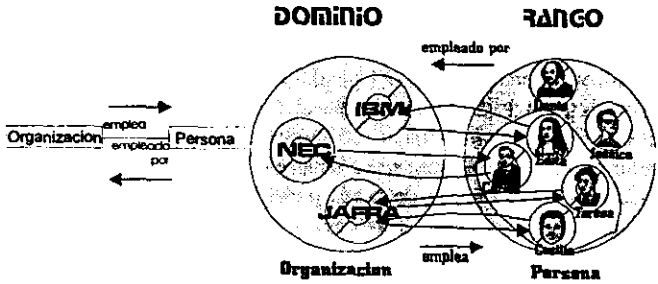


Figura 2-12 Ejemplo de una Instancia de Conexión, la Función emplea define una Asociación

conjunto de objetos de la clase Persona. En este ejemplo la función asocia las personas empleadas con cada organización que los emplea, en este caso Edith se asocia con su patrón IBM, Carlos con Nec, etcétera. Las funciones son analizadas en términos de una asociación desde un conjunto de objetos hacia un conjunto de objetos; los términos convencionales para esto son: domino y rango, respectivamente. De esta forma, las funciones se pueden describir en forma mas breve como asociaciones de un dominio aun rango.

En la figura 2-12, varios objetos Organización son asociados mediante la función **emplea**: ACME, IBM, NEC, JAFRA, etc. El conjunto organización es entonces el **dominio** de la función emplea; La función emplea asocia hacia la colección de objetos Persona que emplea una Organización: Edith, Carlos, Cecilia, Teresa. Esta colección de objetos persona es el **rango** de la función emplea. Todos los demás objetos Persona como Dante y Jessica,

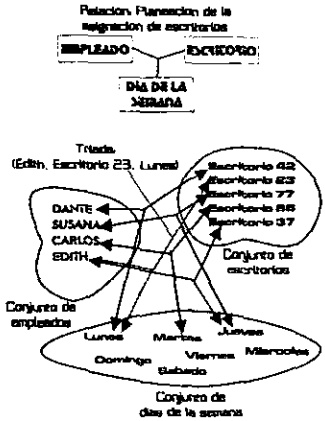


Figura 2-13
Instancia de Conexión llamada Triada

no son empleados por el momento, y por lo tanto, no forman parte del rango de esta función. En otras palabras, el rango de emplea solo incluye a las personas empleadas.

Las instancias de conexión suelen representar relaciones, sobre todo las relaciones binarias (también pueden ser vistas como funciones). Sín embargo las relaciones pueden ser de tres objetos, como en la figura 2-13 que definen una relación *empleado-escritorio-día* de la semana; un ejemplo de dicha relación seria Edith en el escritorio 23 el lunes, este tipo de asociación se le conoce como triadas (terciarias o ternarias); También las relaciones se pueden extender mas allá de las relaciones binarias y trinarias a las asociaciones **n-arias o n-adas** (de n-lugares), un ejemplo de n-ada podría consistir en un objeto proveedor, un objeto cliente, uno o mas objetos productos y uno o mas servicios. La n-ada de esta configuración podría llamarse contrato.

ASPECTOS DEL COMPORTAMIENTO

En el paradigma orientado a objetos se describe a menudo en terminos de la estructura y el comportamiento, el termino estructura es una metáfora visual y espacial que se refiere a la forma en que se distribuyen los objetos en el espacio. La estructura puede especificar distintas configuraciones de los objetos, como empleados, documentos y diseños de ingeniería. En contraste, el comportamiento se refiere a la forma en que cambian los objetos en el tiempo dentro de sus estructuras definidas. Los aspectos del comportamiento incluyen los eventos, estados, operaciones(métodos), eventos (transiciones) y formas de activación (triggers)(reglas de activación y condiciones de control). Por ejemplo, el comportamiento puede especificar la forma de contratar empleados, conectar circuitos o añadir diagramas a un documento.

SUCESIÓN DE CAMBIOS DE ESTADO CON EL TIEMPO

El comportamiento de un objeto se desarrolla como una red de cambios de estado del **tipo causa-efecto.** Por ejemplo un llamado al procesamiento de pedidos, comenzaría con un objeto **pedido solicitado**. Con este objeto, se llamaría una operación **llenar pedido**, con lo que se obtiene el objeto que ahora se clasifica como **pedido llenado**. Esto llevaría a otro cambio de estado, donde el objeto sería clasificado como **pedido embarcado**. Esto continuaría hasta que el pedido se considere completo y el objeto terminado. El cambio de estado de un objeto también puede conducir a un cambio de estado en un objeto diferente. Por ejemplo una vez que el estado del objeto se modifica a **pedido embarcado**, podría requerirse la creación de un objeto **factura**. A la aparición del objeto **factura** le seguiría un cambio de estado que clasificara el nuevo objeto factura como **factura enviada** .

CICLO VITAL DE UN OBJETO

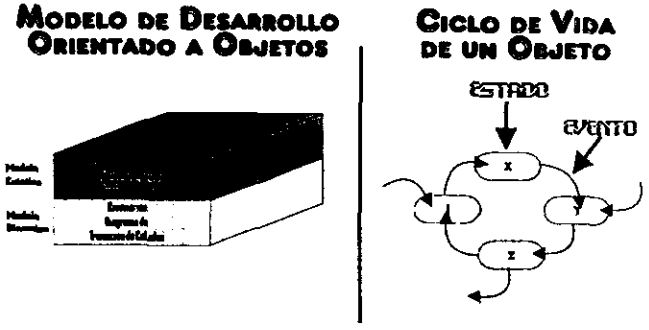


Figura 2-14 Modelo de Desarrollo Oriendo a Objetos y Ciclo de Vida de un Objeto

Cada objeto y cada relación tiene un ciclo de vida. Hasta el momento solo se ha modelado la parte estática de los sistemas pero en el paradigma orientado a objetos tiene una ventaja muy importante sobre el paradigma estructurado la cual es; el modelado de la parte dinámica de los sistemas, dicho modelado se lleva acabo mediante los ciclos vitales de los objetos. (ver figura 2-14)

Un ciclo de vida de objeto, se define como un patrón ordenado de comportamiento dinámico, es decir que una sucesión de eventos que pueden ocurrirle y cada uno de éstos *eventos modifica su estado. Para formalizar el modelado dinámico de los sistemas se usa* el análisis de comportamiento de objetos. En el análisis del comportamiento de objetos se realizan escenarios, traza de eventos y diagramas de transición de estados que muestran eventos, la secuencia en que ocurren y cómo los eventos cambian el estado de los objetos.

ESTADOS DE UN OBJETO

Como se menciono en el capitulo I el estado de un objeto es la colección de los atributos de comportamiento que se aplican a él. En un objeto pueden existir varios estados, pero nunca un objeto puede estar en dos estados al mismo. Por ejemplo, el *objeto reservación aérea puede ser una instancia de alguno de los siguientes atributos de* comportamiento

- Reservación solicitada
- Reservación en lista de espera
- Reservación confirmada
- Reservación cancelada
- Reservación satisfecha (una vez que el avión ha despegado)
- Reservación archivada

Los estados pueden ser de 3 categorías

- **Estados de Creación.**- Algunos modelos de estados tiene uno mas estados donde una instancia primero se crean. Estos son semejantes a los constructores en C++.
- **Estados Finales.**- Uno o mas estados sirven como final del ciclo de vida de una instancia, un estado final también puede representar 2 situaciones, cuando la instancia se vuelve irrelevante o deja de existir. Estos son semejantes a los destructores en C++.
- **Estados Actuales.**- Son atributos que se deben predicar y su dominio es equivalente al numero de estados del objeto al que pertenece.

En el lenguaje OO, las solicitudes se envían en forma de mensajes y provocan la activación de los métodos. Los métodos cambian el estado del objeto y el estado se registra en los datos del objeto.

MENSAJES

Es la forma de comunicación entre objetos, para que un objeto haga algo le debemos enviar una solicitud. Esta hace que se produzca una operación, la operación ejecuta el método apropiado y de manera opcional produce una respuesta; el mensaje que constituye la solicitud contiene el nombre del objeto, el nombre de la operación, y a veces un grupo de parámetros.

Los mensajes pueden ser de 3 tipos.

- **Mensaje de Petición.** Este tipo de mensajes son generados por el cliente y se envían al servidor. La longitud del mensaje es variable
- **Mensaje de Respuesta.** Son mensajes que en algunas ocasiones envía el servidor al cliente. la longitud es variable.

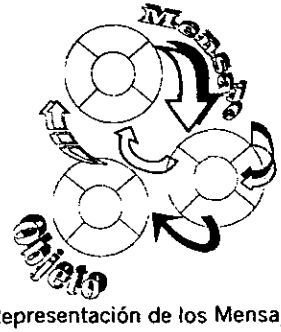


Figura 2-15 Representación de los Mensajes

- **Mensaje de Evento**. Mensajes que envía el servidor al cliente indicando que algún dispositivo realizo alguna acción o que una operación tuvo efectos colaterales a una petición realizando un cambio de estado.
- **Mensaje de Error**. Son mensajes como el de evento, solo que el cliente los maneja generalmente mediante una rutina de errores.

OPERACIONES

En el análisis OO, *el término operación se refiere a una unidad de procesamiento que puede ser solicitada. El procedimiento se implanta mediante un método. El método es la especificación de como llevar a cabo la operación.* Dicho de otra forma es el guión de la operación. A nivel programa, el método es el código que implanta la operación.

Las operaciones se invocan a través de mensajes. Una operación puede o no cambiar el estado de un objeto. Si lo cambiara ocurriría un evento o también llamado transición.

ESCENARIOS

Escenario es una secuencia de eventos que ocurren durante una particular ejecu-

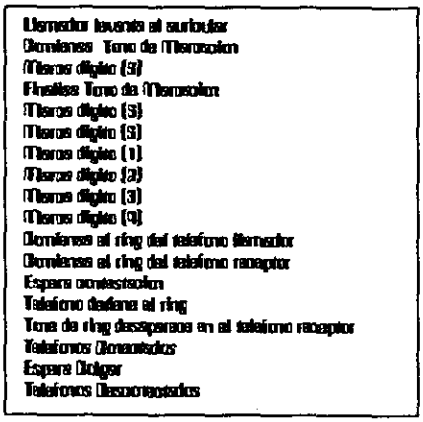


Figura 2-16 Escenario para una llamada Telefonica

ción de un sistema, el ámbito de un escenario puede variar, el escenario puede incluir todos los eventos en el sistema o puede solo incluir algunos eventos involucrados o generados por ciertos objetos en el sistema. Un escenario puede ser el registro histórico de la ejecución de un sistema o un punto de vista experimental de la ejecución de un sistema propuesto. Ver Figura 2-16 que ejemplifica un escenario para una llamada telefónica.

TRAZA DE EVENTOS

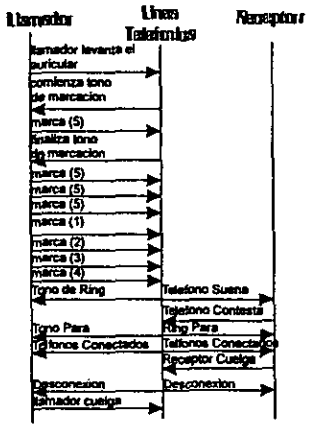


Figura 2-17 Traza de eventos para una llamada Telefonica

La traza de eventos es un escenario aumentado que muestra cada objeto como una línea vertical y cada evento como una línea horizontal. Con este diagrama podemos observar los objetos que están involucrados en el escenario así como su relación con los eventos. Ver figura 2-17 el ejemplo de una traza de eventos

EVENTOS

Ya en el capitulo I se menciono que en el paradigma orientado a objetos el mundo se describe en términos de los objetos y sus estados, así como los eventos que modifican esos estados. **Un evento es un cambio en el estado de un objeto.** Los eventos pueden asociar un objeto con otro. Por ejemplo, en la mayoría de las organizaciones, cuando un objeto se clasifica como **empleado**, debe estar asociado con un **departamento**. Un evento clasificara al objeto como empleado. Otro evento creara la asociación entre el objeto empleado y un objeto departamento. Un evento puede provocar la reacción en cadena de otros eventos. **Una operación hace que los eventos ocurran.** Para poder modelar los eventos y los estados se hace mediante los diagramas de transición de estados. **Los diagramas de transición de estados son útiles para expresar el ciclo vital de un objeto particular.** Sin embargo, la mayoría de los procesos requieren la interacción de varios objetos.

Un Diagrama de Transición de Estados (DTE) relaciona eventos y estados además especifica la secuencia de estados causada por una secuencia de eventos, también se le conoce popularmente como Maquina de Estados Finito (MEF). Cuando un evento es recibido, el siguiente estado depende del estado actual así como del propio evento; **un cambio de estado causado por un evento se llama transición.** El diagrama de transición de estados es una gráfica cuyos nodos son estados y cuyos arcos dirigidos son **transiciones etiquetadas por nombres de eventos.** Un estado se dibuja como una caja redondeada

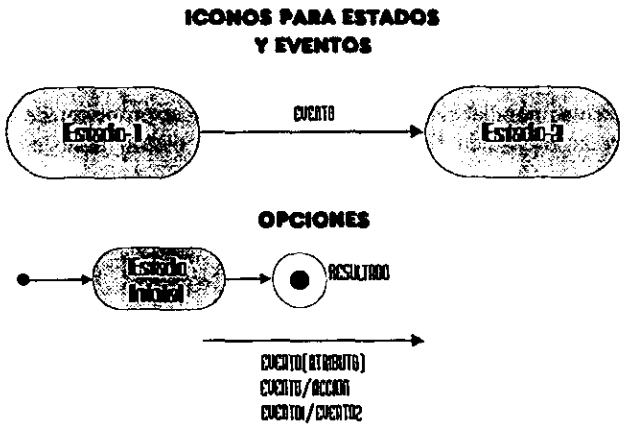


Figura 2-18 Iconografía para Estados y Eventos

conteniendo un nombre opcional, una transición es dibujada por una flecha desde el estado

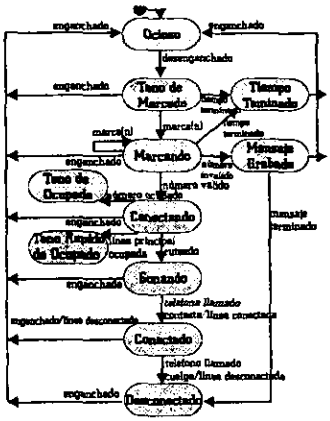


Figura 2-19 *Diagrama de Transición de Estados de una* llamada Telefonica

recibido al estado destino; la etiqueta sobre la flecha es el nombre del evento que causa la transición. La iconografía que usaremos para representar un estado y un evento es la siguiente: ver figura 2-18.

ver en la figura 2-19 un ejemplo de un diagrama de *transición de estados*

MAQUINA DE ESTADOS FINITOS MEF (DIAGRAMA DE TRANSICION DE ESTADOS)

Una técnica popular adoptada para la especificación orientada a objetos es la máquina de estado finito. Una máquina de estado finito es una invención

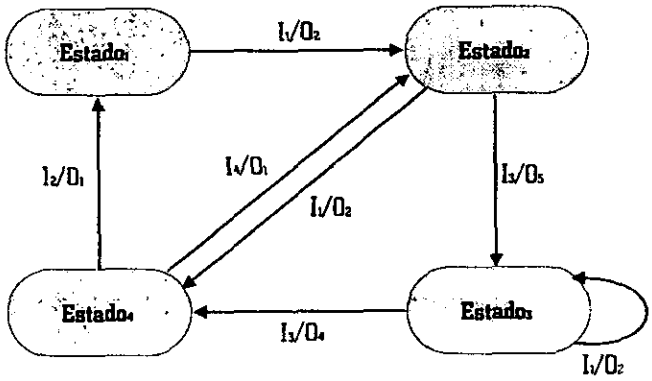


Figura 2-20 En una MEF los cambios de estado dependen del evento anterior

hipotética en la que solo puede existir una cantidad finita de estados en un momento dado. Por ejemplo, el diagrama de transición de estados que se muestra en la figura 2-20 indica que I_3 es un estímulo para uno de los dos posibles cambios de estado. Si la MEF está en el Estado$_2$,un evento de la máquina hace una transición(cambio de estado) al Estado$_3$. Si está la MEF en el Estado$_3$, el cambio es al Estado$_4$. De esta manera, se debe conocer el estímulo y el estado actual antes de que se dé cualquier cambio de estado o transición de estado. Cada máquina en respuesta a un estímulo del ambiente externo, cambia su estado y produce una salida. Por ejemplo en en la figura 2-20, si la máquina está en el Estado$_4$, el evento I_3 produce una transición el Estado$_2$ y una respuesta O_1. La razón de la popularidad de las MEF en el análisis orientado a objetos es que la metáfora de la máquina se puede interpretar fácilmente para incluir las transiciones del estado de un objeto. Para llevar a cabo esto, cada máquina describe el comportamiento de un solo conjunto de objetos. En resumen, cada máquina es un contexto limitado que especifica la forma en que un objeto debe comportarse.

Cada máquina de estados finito tiene un conjunto finito de estímulos o tipos de entrada que pueden llegar a ella y activar el cambio de estado. Un cambio de estado en una MEF es una función de su estado actual y su estímulo dado. En otras palabras, un estado y un tipo de entrada específicos describen las condiciones bajo las cuales hay un cambio de estado. Cuando la acción de una máquina se lleva con éxito ocurren dos cosas: un cambio de estado en el objeto y la generación de una o más respuestas.

COMPONENTES PRINCIPALES DE UN DTE O UNA MEF

1. Un evento en el ambiente externo estimula o activa una operación dentro de una máquina.
2. Cada operación es llamada para cambiar el estado de un objeto exactamente.
3. Además cada operación se especifica en términos de aquellos estados que deben aplicarse a un objeto antes de la operación y los que se le aplican después. Estos preestados y postestados quedan garantizados por cada operación.
4. Dentro de cada MEF se selecciona una operación especifica basada en la forma de activación y las condiciones anteriores del estado.
5. Antes de llamar a la operación, ciertas variantes de MEF requieren la evaluación de una condición de control. Sólo en el caso de que la condición sea cierta se llamará a la operación Una condición de control es un proceso que evalúa si determinadas condiciones son verdaderas o falsas, y solo el procesamiento continua mas allá de este punto solo cuando el resultado es verdadero. Una regla de activación es un proceso que llama a una operación especifica cuando ocurre un tipo especifico de evento. También especifica la forma en que los objetos se transfieren como argumentos a la operación.
6. Cuando la operación llamada se termine exitosamente (es decir, cuando se realice un cambio de estado), ocurrirá un evento o transición.
7. La ocurrencia de un evento indica que se debe enviar una respuesta al ambiente externo. Puesto que el ambiente externo actúa sobre los resultados de la máquina (en caso contrario, la máquina no tendría fin alguno), esta respuesta es en realidad una activación de otra operación de la máquina, aunque externa. El ambiente externo no es más que otra máquina. Sin embargo, no conocemos su especificación. Podría ser una persona, software o una máquina automática, igual que la MEF especificada.

Se puede definir entonces cuatro nociones importantes de comportamiento en las maquinas de estado finito.

- Una **operación llamada** es una unidad de procesamiento encargada de cambiar el estado de un objeto.
- Un **evento** es la terminación con éxito de una operación llamada, es decir, un cambio notable de estado.
- Una **activación (trigger)** liga la causa con el efecto. Responde a eventos y determina los objetos necesarios como argumentos para la operación que llama.
- Una **evaluación de la condición de control** garantiza que determinadas condiciones son ciertas.

En resumen

Los componentes principales de la especificación de una MEF son los tipos de eventos, reglas de activación , condiciones de control y operaciones (que cambian de estado). Las instancias correspondientes son eventos, activaciones evaluaciones de condiciones de control y operaciones llamadas respectivamente. Los eventos son cambios de estado de los objetos. Las activaciones responden a eventos al llamar operaciones con los objetos necesarios como argumentos. Las evaluaciones de condiciones de control actúan como vigilantes para garantizar que una operación se ejecutara solo cuando su prueba de condición es cierta. Cada operación llamada es un proceso encargado en forma activa de cambiar el estado de un objeto al implantar un método especifico. Un método es el guión de una operación. Esta distinción entre operación llamada y evento es la distinción entre el cambio potencial y real del estado de un objeto, las activaciones ligan los cambios reales con los nuevos cambios potenciales.

JERARQUIZAR

En el capitulo I se menciono que la jerarquización es el resultado de ordenar la

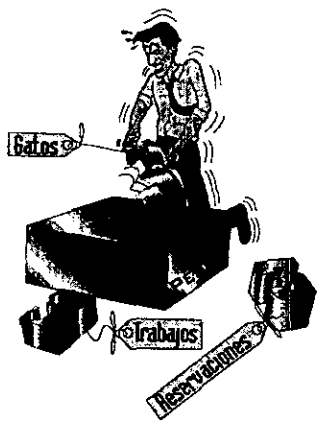


Figura 2-21 La tipificación previene la mezcla de abstracciones

abstracción y que un conjunto de abstracciones forman una jerarquía. El significado de jerarquización es el concepto de tipo, que deriva principalmente de las teorías de tipos de datos abstractos. Como Deutsch sugiere, *"Un tipo es una caracterización precisa de propiedades estructurales o de comportamiento que un conjunto de entidades comparten"*. Para nuestros propósitos, usaremos los términos tipificación y jerarquización intercambiablemente. Aunque que tipificación, es el proceso en el cual no solamente hay que partir de una semántica, como en el proceso de clasificación, sino hay que partir de una clase o tipo; dado un elemento de referencia o primario (tipo o clase) se define un conjunto de objetos que determinan una especialización de dicho elemento. Específicamente, nosotros afirmamos lo siguiente. La tipificación es la aplicación de la clase a un objeto, tal que,

los objetos de tipos diferentes no pueden ser intercambiados, o que la mayoría de ellos puedan ser intercambiados únicamente de maneras muy restringidas. Ver Figura 2-21 donde la tipificacion previene la mezcla de abstracciones

Vivimos en un mundo complejo. Incluso las tareas más sencillas nos confundensi no contamos con las herramientas mentales adecuadas. Los conceptos surgen de la capacidad de abstraer. Una vez formadas estas abstracciones, podemos continuar organizando nuestro mundo para distinguir si una abstracción es mas general que otra, esto nos permite construir *jerarquías de generalizaciones* (Herencia). *Por ultimo, podemos componer un objeto* mediante una configuración de otros objetos (Agregación). Estos tres mecanismos son parte de nuestro patrimonio humano y nos dan la capacidad de percibir y controlar la complejidad de nuestro mundo.

Los objetos no forman conjuntos por si mismos, si no que esto lo hace el proceso de abstracción; *dos cosas que tienen la misma forma abstracta son análogas, estas formas* abstractas se llaman conceptos. *La jerarquización (tipificación) es entonces una forma de manejar la complejidad de los objetos en el mundo, ya que el objetivo principal de la jerarquización es definir los distintos niveles de abstracción.*

La abstracción es una buena cosa, pero en la mayoría de las aplicaciones excepto en las triviales, nosotros podemos encontrar muchas abstracciones diferentes, que podemos comprender a la vez. La encapsulación ayuda a manejar la complejidad por ocultarla dentro de sus abstracciones. La modularidad como veremos mas adelante ayuda también, por que nos da una forma de agrupar abstracciones relacionadas lógicamente. Pero aun así estas no son suficientes; un conjunto de abstracciones frecuentemente forma una jerarquía.

Las dos más importantes jearquizaciones en un sistema complejo son: la estructura clase ("es un" Herencia) y la estructura objeto ("parte de" Agregación)

HERENCIA

Herencia (o generalización) cuando miramos dentro de nuestro ropero, reconocemos los objetos que vemos: pantalones, camisas, abrigos, zapatos, patines, etc. Sin una capacidad bien desarrollada para generalizar a esta área del almacenamiento lo podríamos llamar mueble de zapatos-pantalones-camisas-abrigos-patines. Cuantas más cosas tengamos allí, más complicado será el nombre. La generalización nos permite examinar si esos conceptos tienen algo en común. ¿Existe un concepto más general que abarque a los conceptos como zapatos, pantalones y camisa? En general, se les conoce como ropa; ropa a su vez, se puede comparar a una categoría más general llamada mercancía o artículo de almacén, **con la herencia podemos construir jerarquías de concepto** para poder formar cada vez conceptos más generales lo opuesto a la generalización es la especialización (o particularización). Por ejemplo, humano se puede especializar como humano del sexo femenino o humano del sexo masculino; también como infante, adolescente o adulto, o como bueno, malo o feo. Estos conceptos especializados son subconceptos.

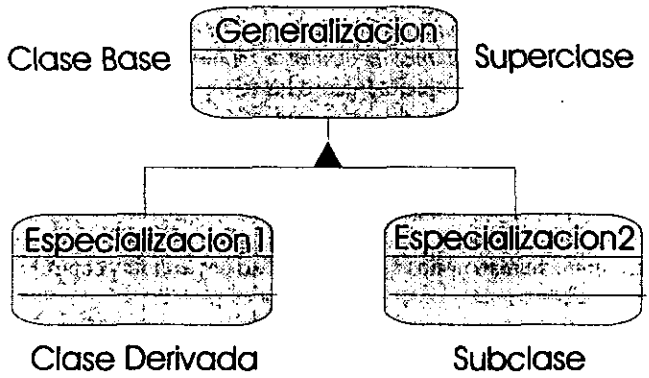


Figura 2-22 Icono de Herencia y nombres mas usados para refenciar las clases de la herencia

Básicamente la herencia es una relación entre clases del tipo generalización /especialización donde una clase comparte la estructura y/o comportamiento definido en una o más clases. **Es decir el reuso de clases por clases;** en el que una nueva clase reusa las características de una y solo una clase a lo que se llama herencia simple, dos o más clases herencia múltiple respectivamente. **Semánticamente la herencia denota una relación "es un"**. Por ejemplo un oso es un tipo de mamífero, una casa "es un" tipo de bien tangible, y un quick sort "es un" tipo de algoritmo de ordenamiento. La iconografía que usaremos en esta tesis es la siguiente: La notación es direccional; esta usa una línea o líneas, dibujadas hacia fuera desde un triángulo sólido, hasta el punto de la generalización o generalizaciones en caso de la herencia múltiple; **la relación generalización/especialización** puede ser dibujada en cualquier ángulo. Sin embargo, ubicando consistentemente los iconos con nuestra forma de pensar en los lugares altos deberá ir la generalización y en los lugares bajos la especialización, ya que esto producirá un modelo fácil de entender. Ver figura 2-22

La forma de nombrar los componentes de la relación, tiene varias formas que son comúnmente usadas a travez de varias metodologias y lenguajes de programación OO cada una se utiliza de la siguiente manera (superclase con subclase), (clase padre con clase hija) y (clase base con clase derivada), estos términos se pueden utilizar indistintamente para referirse a lo mismo.

La herencia entonces implica una jerarquía generalización/especialización, en donde una subclase especializa la estructura o comportamiento mas general(es) de su(s) superclase(s); las clases base representan abstracciones generalizadas, y las clases derivadas representan especializaciones en las cuales los campos y métodos de la clase base son agregados, modificados, o igualmente ocultados. Por lo tanto los comportamientos de clases más específicos adquieren un contexto más abstracto y los comportamientos de clases más generales adquieren un contexto menos abstracto.

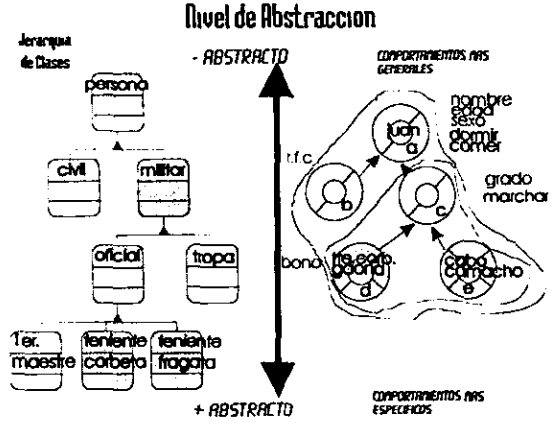


Figura 2-23 La herencia y los niveles de Abtracción

Una clase de alto nivel de abstracción puede especializarse en clases de bajo nivel. Una clase puede tener subtipos. Por ejemplo el tipo persona puede tener subtipo civil y militar. Militar puede tener subtipos oficial y tropa. Oficial puede tener subtipos 1er Maestre, Teniente de corbeta, Teniente de fragata. como se puede observar en la figura 2-23 en donde la clase base más genérica es persona que tiene comportamientos de un contexto menos abstractos como son el dormir y el comer, mientras militar tiene un comportamiento mas especializado que es marchar, esto hace que el contexto del comportamiento marchar sea más abstracto

PROCESO DE HERENCIA

Para saber cuando ocupar la herencia se tiene que aplicar la siguiente regla de herencia sean A y B dos clases diferentes y 'q' un enunciado referente a una semántica por ejemplo:

- q= "Se comporta como" (enunciado de semántica funcional)
- q= "Proviene de Z al igual que" (enunciado de semántica ontológica)
- Si B 'q' A entonces B se debe heredar de A.

Para evolucionar nuestra jerarquía de herencia, la estructura y comportamientos que son comúnes para clases diferentes tenderá a emigrar a una superclase.

Siempre que se construya una nueva clase, se debe determinar si se puede usar una clase preexistente, como clase base. A menudo se encuentran clases que proporcionan casi el comportamiento indicado. Estos son buenos candidatos ya que se puede heredar todas las características deseadas y deshabilitar las no deseadas para deshabilitar una función de la clase base, se emplea la técnica sobrecarga de funciones.

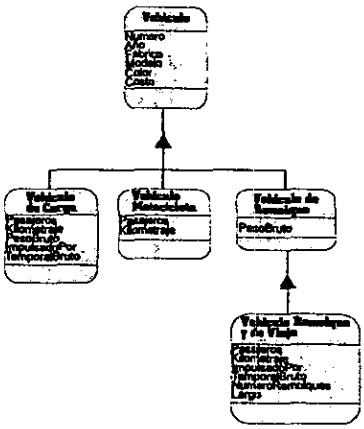


Figura 2-24 Diagrama de jerarquia de clases de vehiculos

Intente heredar el mayor número posible de características de las clases preexistentes lo que esto reduce el tamaño del código y el tiempo que se requiere en la depuración.

Si encontramos que existen subclases entonces se puede definir una jerarquía de clases. Como producto de diagramar las diferentes tipificaciones se obtiene lo que se llama un diagrama de jerarquía de clases. Ver la figura 2-24

Como se menciono en párrafos anteriores se le llama herencia simple aquella herencia que reusa una y solo una clase, y al reuso de dos o más clases se le llama herencia múltiple. La herencia múltiple es conceptualmente directa, pero introduce algún grado de complejidad en los lenguajes de programación, ya que estos deberán poner atención a dos puntos. Los conflictos entre dos nombres de diferentes superclases y repeticiones de herencia: El

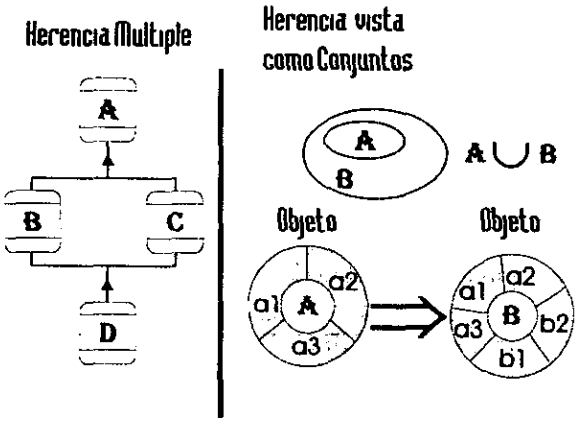


Figura 2-25 Conflicto de la Herencia Multiple en repetición de herencia y La Herencia Vista como Logica de Conjutos

conflicto ocurrirá cuando dos o más superclases provean un campo u operación con el mismo nombre.

La herencia repetida ocurre cuando dos o más superclases comparten una superclase común. En tal situación, el árbol de herencia será en forma de diamante como el de la Figura 2-25, y entonces surge la pregunta ¿ ver que clase tiene una copia o múltiples copias de la estructura de la superclase compartida? Algunos lenguajes prohiben herencia repetida, algunos unilateralmente escogen un enfoque, y otros, tal como C++, permiten al programador decidir.

Otra forma de pensar en la herencia es dibujando conjuntos de objetos dentro de otros conjuntos de objetos. Por ejmplo Ver la Figura 2-25 Sean A y B objetos, y ademas a1,a2,a3 y b1 y b2 comportamientos de dichos objetos respectivamente; en donde el objeto B hereda de A sus comportaminentos y estructura. Visto desde logica de conjuntos B es un Superconjuto de A, es decir se suman sus propiedades como si fuera una union.Dando como resultado que el objeto B tenga los compotamientos b1,b2 mas a1,a2,a3.

En la herencia además existen tecnologías (como tecnología nos referimos a los lenguajes de programación OO) y técnicas muy poderosas que apoyan a dar gran énfasis al paradigma OO. Como técnicas mencionaremos el polimorfismo y como tecnología los prototipos (También conocidos como frameworks, Marcos o Patrones)

POLIMORFISMO O (SOBRECARGA)

Polimorfismo es la capacidad para que diferentes objetos respondan a ordenes similares de modo distinto. O visto de otra forma, mediante el polimorfismo, es posible diseñar funciones con igual nombre, que al actuar sobre diferentes objetos se comportan de modo distinto. A los objetos que tienen esta propiedad se les llama objetos polimorficos.

El polimorfismo trata los objetos de las clases relacionadas de una manera genérica, un buen ejemplo de polimorfismo es un programa de dibujo, en el que las figuras: círculo, rectángulo y líneas se representan por una clase, y cada clase desciende de una clase base llamada figura. Una figura puede ser una instancia de muchos tipos derivados, por ejemplo: un círculo, un rectángulo, u otro tipo de gráfico. En tiempo de ejecución, una función llamada dibujar, en cada clase ejecuta las instrucciones necesarias para dibujar la figura especificada en la pantalla. . El polimorfismo se realiza por uno de los dos métodos siguientes: **sobrecarga de operadores y sobrecarga de funciones**

Al polimorfismo también se le conoce como sobrecarga, de esta forma podemos utilizar indistintamente dichos términos. La sobrecarga se refiere a la práctica de cargar una función con mayor significado. Básicamente, el término expresa que se cargan uno o más identificadores de funciones sobre un identificador previo.

¿PORQUE SE DEBE USAR LA SOBRE CARGA DE OPERACIONES?

En el castellano, como probablemente en la mayoría de los idiomas, algunos verbos se utilizan en diferentes contextos para expresar acciones distintas. Consideremos el verbo tomar, el cual se puede utilizar en todo tipo de situaciones. Se puede:

- Tomar un refresco
- Tomar medidas pertinentes
- Tomar un descanso
- Tomar el pelo
- Tomar descuidado

Esta lista ni siquiera incluye usos de la palabra tomar como sustantivo, como en "una toma de película", o "la toma del agua" etcétera. *En términos de un lenguaje de programación orientado a objetos, se dice que la palabra tomar, cuando se utiliza como verbo, está sobrecargada.* Cada contexto está asociado con un significado distinto; pero todos los usos del verbo están relacionados conceptualmente de alguna manera. El uso repetido del mismo verbo en situaciones diferentes pero similares no solo crean una pista importante para nuestro proceso de razonamiento sino que reduce también el número de palabras requeridas en el idioma castellano.

Cuando se trabaja con POO (programación orientada a objetos), la simpleza es importante. Entre menos necesite saber acerca de un sistema para utilizarlo es mejor. Lo mismo se aplica al código.

La sobrecarga no es un concepto nuevo en los lenguajes de programación. Por ejemplo, el operador = esta sobrecargado en muchos lenguajes de alto nivel, y se utilizan en instrucciones de asignación y en expresiones condicionales, como

- A=B
- if a = b

La sobrecarga permite a las personas utilizar código con menos esfuerzo, ya que extiende operaciones que son conceptualmente similares en la naturaleza. La ventaja que se obtiene por medio de la sobrecarga no es gratuita, ya que con sus ventajas el compilador experimenta mayor complejidad ya que ahora tiene que discernirlo todo. El problema con las palabras sobrecargadas incluso en un idioma natural como el español, es que la semántica depende del contexto. En otras palabras, el significado exacto de un identificador sobrecargado solo se puede entender considerando algunas palabras circunvecinas; las palabras circunvecinas m, son el número y tipo de los argumentos que se utilizan en las llamadas a funciones

Habiendo establecido que la sobrecarga es benéfica quizá sea tiempo de ahondar en algunos detalles. Como se dijo antes, todas las funciones sobrecargadas utilizan el mismo identificador, y se distinguen por el número y tipo de sus argumentos.

Con la sobrecarga, puede surgir todo tipo de problemas, en especial cuando existen conversiones del usuario e implícitas. Si se requieren conversiones, si existen argumentos definidos o si se utilizan listas de argumentos de longitud variable, y se pueden esperar que surjan otros tipos de problemas.

SOBRECARGA DE FUNCIONES

Una función sobrecargada es una función que tiene mas de una definición. Aunque cada definición utiliza el mismo nombre, las definiciones operan como funciones diferentes.

Propiedades de la sobrecarga de funciones.

- Las funciones sobrecargadas tienen el mismo nombre pero deben tener
- un número diferente de argumentos,
- Diferentes tipos de argumentos o ambos.
- Al menos uno de los argumentos debe ser un tipo definido por el usuario.

¿POR QUE SE DEBE USAR LA SOBRE CARGA DE OPERADORES?

La programación orientada a objetos es una metodología que obtiene mucho de su poder haciendo que los objetos se controlen por sí solos al momento de la ejecución. El uso de funciones sobrecargadas no solo da uniformidad de expresión en llamadas a funciones de diferentes objetos sino mayor intuitividad en nombres de funciones.

La sobrecarga de operadores no es un concepto tan nuevo como podría pensarse ha sido parte de los lenguajes naturales por miles de años. Considere la operación genérica de adición podemos:

- Sumar dos números
- Sumar naranjas
- Sumar naranjas a manzanas
- Agregar azúcar al agua o bien, de manera un poco más abstracta
- Agregar madera al fuego.
- Agregar suspenso a una historia.

La adición se puede aplicar de manera indiferente a objetos que se puedan contar, como manzanas y naranjas, y a cosas abstractas como el suspenso, por lo tanto, el concepto de adición es intuitivo en el sentido de que se utilizan para denotar una acción de aumento

de un número, intensificación de un atributo etcétera. Lo importante aquí es que el concepto de adición es más bien vago a menos que se le asocie con un contexto.

El dualismo entre operadores y llamadas a funciones sugiere la posibilidad de sobrecargar operadores, habilitando diferentes tipos de objetos para usar operadores, según resulte apropiado. El uso de operadores de manera similar a los idiomas naturales es deseable; de modo que sería conveniente poder utilizar las siguiente expresiones:

- Naranja mis_naranjas, sus_naranjas;
- mis_naranjas = mis_naranjas + sus_naranjas
- Manzana mis_manzanas
- Fruta mi_fruta
- mi_fruta = mis_naranjas + mis_manzanas
- Ingrediente_de_Pastel Azúcar, Agua, Hornada
- Hornada = Azúcar + Agua

Y también expresiones mas abstractas como

- Combustible Madera;
- Combustión Fuego;
- Fuego += Madera;
- Atributo Suspenso;
- Drama Historia;
- Historia= Historia + Suspenso

De nuevo, las metas son simpleza de la notación y uniformidad de expresión. La tarea de aplicar un nuevo operador en una expresión recae en los objetos sobre los que se actúa y no en el programador. Los operadores sobrecargados son aún más poderosos porque pueden ser heredados con algunas excepciones. Lo que es realmente nuevo en el concepto de sobrecarga de operadores como es la extensión de la sobrecarga de funciones para incluir operadores, que son equivalentes a llamadas a funciones.

SOBRECARGA DE OPERADORES

Es la operación de redefinición de un operador.

Propiedades de la sobre carga de operadores:

- Sobrecargados, no cambiaran su numero de operandos (unitarios o binarios)
- Sobrecargados, no cambiaran su prioridad
- No se pondrán inventar operadores que no existan en el lenguaje.

Para acabar con este apartado de la herencia veremos la tecnología de los lenguajes OO que apoyan lo que se llama **prototipos**, como ya se menciono en *párrafos anteriores a* este concepto también se le ha llamado FrameWorks, Marcos o Patrones nosotros preferimos usar el termino patron ya que expresa mejor su objetivo. **Un patron es parecido a una receta que soluciona tipos de problemas específicos y que por lo menos han sido probados en dos sistemas.** Los orígenes de esta idea son fundados en la arquitectura en donde, por ejemplo para cada tipo de terreno existe documentación sobre que materiales utilizar en la construcción, así como, *que tipo de construcción es recomendable*, que materiales y estructuras habrá que utilizar para una casa de dos pisos, que para un rascacielos, ya que es obvio que tienen diferentes características. Además de que se puede escoger entre una gran variedad de cosas prefabricadas, *es así como surge la idea de aplicar* este tipo de concepto a la ingeniería de software dando como resultado su implementación en algunos lenguajes OO, dicha implementación se hace de dos formas por Clases **Abstractas** o por **Clases polimorficas**. *Los patrones nos dan claramente una idea de la* forma como modelamos nuestro mundo, así de cómo aprendemos. Primero encontramos

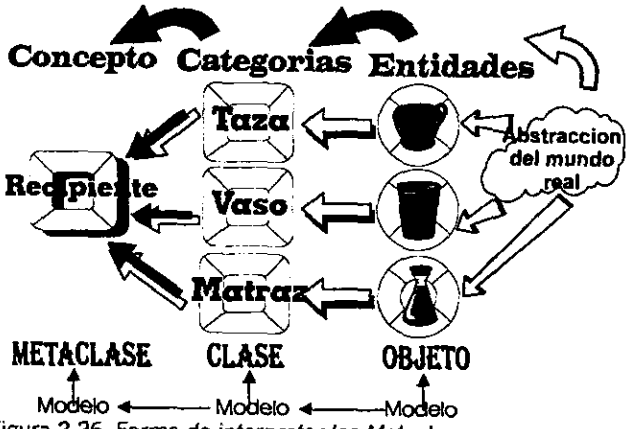


Figura 2-26 Forma de interpretar las Metaclases

la abstracción del mundo real donde lo que buscamos son entidades, estas entidades se modelan como objetos, después estas entidades se suben al *siguiente nivel de abstracción* y lo que obtenemos son categorías que se modelan en forma de clases, después estas categorías se suben al siguiente nivel de abstracción y lo que resulta son conceptos, estos conceptos se modelan como metaclases o clases abstractas, y así se podría ir subiendo de niveles de abstracción por el momento el máximo nivel de abstracción que tenemos son conceptos pero podría tal vez existir metaconceptos y también quizás se podrían modelar. Por ejemplo en la figura 2-26 de la abstracción del mundo real encontramos objetos taza, vaso y matraz que son entidades, después estas entidades les aplicamos nuevamente la abstracción y lo que obtenemos son clases vaso, taza y matraz estas clases representan categorías, ya que de estas clases se pueden instanciar (crear) múltiples objetos, pero prosiguiendo a aplicar nuevamente la abstracción todas esas clases se pueden generalizar

en un concepto llamado recipiente y lo que modela ese concepto en un lenguaje OO es la metaclase.

CLASES ABSTRACTAS

Las clases que se diseñan en una aplicación pueden ser abstractas y concretas. Abstractas son las clases que se suelen incluir el nivel más alto de la jerarquía de clases y se caracteriza porque sirven para definir clases concretas; estas clases no pueden instanciar (no pueden crear instancias, o sea objetos de ella) Las clases concretas por el contrario, se pueden instanciar (se pueden crear instancias). Las Metaclases (plantillas) son generadoras de clases que sirven para generar otras clases.

Propiedades de las clases abstractas

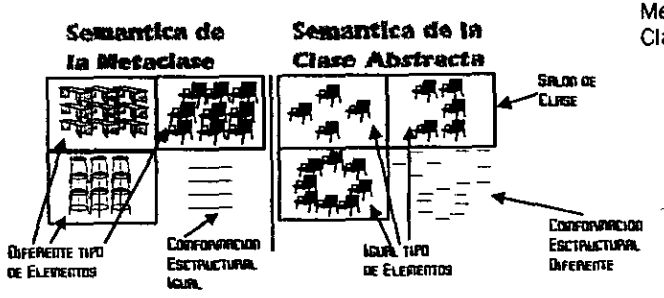
- Una clase abstracta es una clase destinada a generar otras clases derivadas por herencia pero no se puede instanciar, es decir, no se puede utilizar para crear o declarar objetos.
- Las clases abstractas utilizan tipos especiales de funciones denominadas funciones virtuales puras y, al menos, contienen una función de ese tipo.
- Una función virtual pura es una función cuya definición es nula (0), es decir, no tiene cuerpo, y su sintaxis es similar a:

Virtual int poner (int = 0)

METACLASES (PLANTILLAS, CLASES PARAMETRIZADAS, CLASESES POLIMORFICAS)

Las Metaclases también se le conocen como plantillas (templates), marcos, clases

Figura 2-27
Semantica de la Metaclase y de la Clase Abstracta



parametrizadas o clases polimorficas, que permiten definir funciones genéricas

y tipos parametrizados. Las plantillas de funciones (también llamadas funciones plantilla) proporcionan un mecanismo para crear una función genérica; que es una función que puede

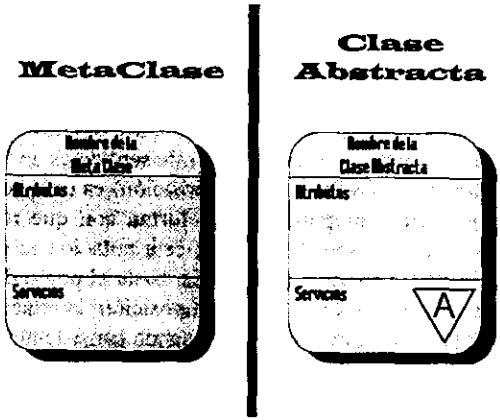
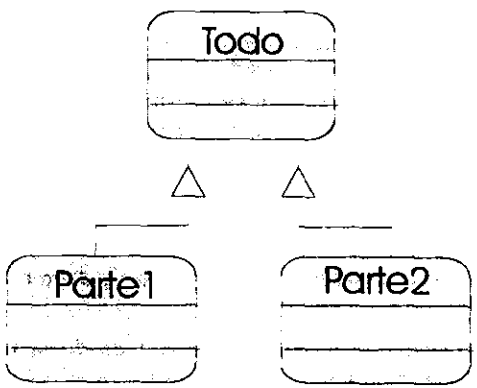


Figura 2-28 Iconografia para las Metaclases y Clases Abstractas

soportar simultáneamente diferentes tipos de datos para sus parámetros por ejemplo se puede crear una pila que acepte enteros, reales, arboles, ventanas, etc. Pero lo más importante de todo esto es entender la semántica de cada una de ellas. **Las metaclases se usaran siempre que la disposición estructural sea la misma pero los elementos sean de diferente tipo.** *Por ejemplo al modelar un salón de clases este puede ser formado por bancas, sillas, bancos, sillones, etc. Pero siempre en la misma disposición es decir ordenado en filas.* **Las clases abstractas se usaran siempre que se construya con los mismos tipos de elementos pero con diferente disposición estructural.** *Por ejemplo volviendo con el ejemplo del salón de clases, siempre se construirá con sillas pero su forma de ordenar las sillas dentro del salón son distintas.* Ver figura 2-27

Figura 2-29 Icono de Agregación



Los iconos que se usaran para representar las Metaclaases y las Clases Abstractas son Ver la figura 2-28

AGREGACION

Formalmente la agregación es una relación entre objetos del tipo todo/parte, es decir el reuso de objetos por objetos; en este sentido es un tipo especializado de asociación. La agregación también llamada (composición) es un mecanismo para formar un todo a partir de las partes componentes. La forma con que representaremos a la agregación es : usando una notación direccional; esta notación usa una línea o líneas, dibujadas hacia fuera desde las partes componentes, hasta el punto del todo pasando por un triángulo hueco que indicara la forma de la composición; la relación todo/parte puede ser dibujada en cualquier ángulo. Sin embargo, ubicando consistentemente los iconos con nuestra forma de pensar en los lugares altos deberá ir el todo y en los lugares bajos las partes componentes, ya que esto producirá un modelo fácil de entender. Ver figura 2-29

Por ejemplo, la agregación puede configurar estructuras ensambladas, como la

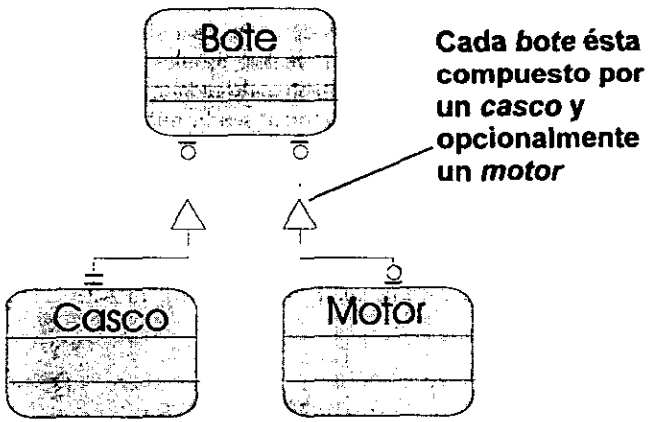


Figura 2-30 La Agregación puede indicar configuraciones de estructuras emsambladas

siguiente: **un bote está formado por casco y motor** véase figura 2-30, o bien cada martillo consta de cabeza y mango. Otros ejemplos de agrupamientos de componentes son: sindicato es una asociación de miembros empleados; cada registro esta compuesto por los valores de sus campos. Además, las partes componentes pueden, a su vez, tener sus propias partes componentes. Por ejemplo, automóvil puede constar de cuatro ruedas, chasis, motor, radiador, etcétera. Su motor puede constar de varios pistones, bielas, válvulas y un monoblock. De esta forma, la composición puede definir jerarquías de configuraciones todo-parte.

La composición es el acto o el resultado de formar un objeto configurado mediante sus partes componentes. La composición reduce la complejidad al tratar muchas

cosas como una sola. Por ejemplo, podemos tratar cada objeto cuerpo humano como una unidad, aunque sea una configuración de otros objetos, como brazos, piernas, cabeza, corazón etcétera. Ciertos objetos complejos, como un título de acciones o un contrato de venta parecen ser estables, aunque otros no lo son. Por ejemplo, cuerpo humano reemplaza de manera constate a sus células. Aun así, cuando observamos a Carlos o a Teresa, todavía los vemos como el mismo objeto, aunque no estén compuestos por los mismos objetos de un momento al siguiente. Del mismo modo, si su automóvil pierde un limpiaparabrisas o

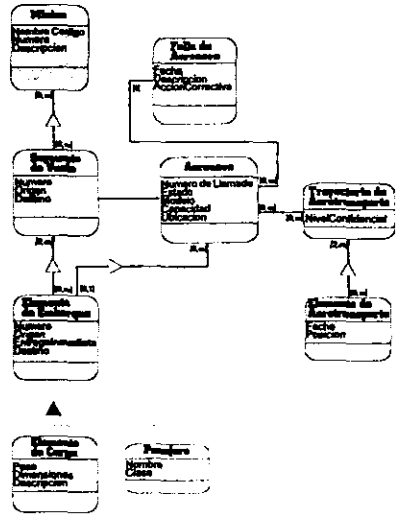


Figura 2-31 Diagrama de Agragación de un Sistema de Tranporte por Via Area en Tiempo Real

se le cambia una llanta, ¿será el mismo objeto automóvil?

La composición trae consigo una serie de preguntas relativas a la forma en que componemos y reconocemos a los objetos complejos. Permitimos que algunos componentes de un objeto cambien con el tiempo, aunque otros componentes se definen como inmutables. Por ejemplo, un matrimonio consta de un esposo y una esposa. Ninguno de estos componentes puede cambiarse sin destruir la pareja. Cada pareja es un objeto cuya abstracción produce el concepto matrimonio.

La agregación y la generalización es un tipo de asociación. Así la composición se puede describir como una asociación "tiene un" (ese bote tiene un casco y un Motor.) La composición forma un objeto, configurado mediante sus partes composición inmutable es el resultado de considerar a uno o más objetos relacionados como los componentes inmutables de un objeto complejo único. Observemos en la figura 2-31 un ejenplo de agregación un poco mas complejo de un Sistema en Tiempo Real de transporte por via aerea.

PROCESO DE AGREGACION

Para saber cuando ocupar la Agregación se tiene que aplicar la siguiente regla de agregación: Sean A y B dos objetos diferentes y 'q' un enunciado referente a una semántica por ejemplo:

- q= "tiene un" (enunciado de semántica estructural)
- q= "parte de" (enunciado de semántica estructural)
- Si B 'q' A entonces A se debe agregar a B.

Además se debe tener en cuenta los siguientes contextos:

- Buscar miembros que formen una colección: Miembros-Colección
- Buscar partes que se puedan ensamblar: Partes-Ensamble
- Buscar contenidos que se puedan ver como recipientes Contenidos-Recipientes.

MODULARIDAD

El Significado de Modularidad. Como Myers observa, es "El acto de dividir un programa en componentes individuales que puede reducir su complejidad en algún grado".

Aunque dividir un programa sea útil por esta razón, una justificación más poderosa para dividir un programa, es que **crea un número bien definido de limites documentados dentro del programa. Estos limites, o interfaces, son inapreciables en la comprensión del programa**. En algunos idiomas, como Smalltalk, no hay concepto de módulo, y la clase forma la unidad física única de descomposición. En muchos otros, incluyendo Object Pascal, C++, CLOS, y Ada, el módulo es una contracción del lenguaje separado, y por lo tanto garantiza un conjunto separado de decisiones de diseño. En estos lenguajes, clases y objetos forman la estructura lógica de un sistema: nosotros ponemos estas abstracciones en módulos para producir la arquitectura física de los sistemas. Especialmente para aplicaciones muy grandes, en que nosotros podemos tener cientos de clases, el uso de módulos es esencial para ayudar administrar la complejidad.

Liskov declara que "modularizar consiste de dividir un programa en módulos que pueden compilarse separadamente, pero que tienen conexiones con otros módulos. También nosotros usaremos la definición de Parnas: Las conexiones entre módulos son las suposiciones que los módulos hacen uno sobre el otro. La mayoría de los idiomas que apoyan el módulo como un concepto separado también distinguen entre la interface de un módulo y su implementación. Así es justo decir que Modularidad y Encapsulación van tomados de la mano. Como con la encapsulación, los lenguajes OO apoyan la Modularidad de diversas maneras. Por ejemplo, los módulos en C++ son nada más archivos que se compilan separadamente. La tradicional practica en la comunidad de C/C++, está en poner las interfaces de módulos en los archivos llamados archivos de cabecera con extensión *.h

y *.hpp; Las implementaciones de los módulos se ponen en archivos con extensión *.c y *.cpp. Las dependencias entre archivos pueden entonces afirmarse usando la macro #include. Este enfoque es enteramente una convención; ya que no es requerido ni impuesto por el lenguaje mismo. Object Pascal es un poco más formal sobre la materia. En este lenguaje, la sintaxis para unidades (su nombre para módulos) distingue entre la implementación e interface de módulo. Las dependencias entre unidades pueden afirmarse únicamente en la interface de un módulo. Ada va un paso adelante. Un paquete (su nombre para módulos) tiene dos de partes: la especificación de paquete y el cuerpo de paquete. Diferente de Object Pascal, Ada permite conexiones entre módulos para ser afirmadas separadamente en la especificación y cuerpo de un paquete. Así es posible que un cuerpo de paquete dependa de módulos que de otra manera no son visibles a la especificación del paquete. Determinar sobre el conjunto de derechos de los módulos para un problema determinado, es casi tan duro un problema como determinar el conjunto de las abstracciones. La descomposición en módulos puede ser bastante difícil; para aplicaciones tales como sistemas de defensa o control espacial.

PROCESO DE MODULARIZAR

Los módulos sirven como recipientes físicos en los que nosotros, declaramos las clases y objetos de nuestro diseño lógico. Esta situación no es diferente a la que es encarada por un ingeniero eléctrico que diseña una placa - madre de computadora, las compuertas NAND, NOR y NOT podrían usarse para construir la lógica necesaria, pero estas compuertas deben físicamente envasarse en circuitos integrados estándares, tales como un 7400 7402 o 7404. Ya que el ingeniero de software carece de partes estándares de software tiene apreciablemente más grados de libertad - como si el ingeniero eléctrico tuviera una fundición de silicio a su disposición.

Para problemas minúsculos, el desarrollador podría decidir declarar cada clase y objetos en el mismo paquete. Para la mayoría del software trivial, una mejor solución es agrupar clases y objetos lógicamente relacionados en el mismo módulo, y exponer únicamente esos elementos que los otros módulos absolutamente deben ver. Este tipo de modularización es una cosa buena, pero puede irse a los extremos. Por ejemplo, considere una aplicación que corre sobre un conjunto de procesadores y usa un mecanismo de pase de mensajes para coordinar las actividades de programas diferentes. En un sistema de gran escala, es común tener varios cientos o igual unos pocos, miles de tipos de mensajes. Una estrategia ingenua podría ser definir para cada clase de mensaje su propio módulo. Como resulta ser, está es una decisión singularmente pobre de diseño. No solamente crea una pesadilla de documentación, sino también hace terriblemente difícil para cualquier usuario encontrar las clases que ellos necesitan. Además, cuando las decisiones cambian, los centenares de módulos deberán ser modificados y recompilados. Este ejemplo muestra como ocultar o modularizar información pueden salir al revés. La modularización arbitraria es a veces peor que ninguna modularización o que modularizar todo.

En el diseño tradicional estructurado, la modularización se preocupa principalmente de la agrupación significativa de subprogramas, usando los criterios de acoplamiento y cohesión. En el diseño orientado a objetos, el problema es sutilmente diferente: la tarea es decidir donde el paquete físicamente de clases y objetos desde la estructura lógica del diseño, son claramente diferentes subprogramas.

Nuestra experiencia indica que hay varias directivas técnicas útiles así como también no-técnicas que nos pueden ayudar, a lograr modularizar clases y objetos inteligentemente. Como Britton y Parnas han observado. *"La meta total de la descomposición en módulos es la reducción del costo del software por permitir a los módulos ser diseñados y revisados independientemente. La estructura de cada módulo debería ser lo suficientemente simple que pueda entenderse totalmente; debería ser posible cambiar la implementación de otros módulos sin el conocimiento de la implementación de otros módulos y sin afectar el comportamiento de otros módulos; la facilidad de hacer un cambio en el diseño debería llevar una relación razonable a la probabilidad de necesitar dicho cambio"*. Hay un borde pragmático a estas directivas. En la práctica, el costo de recompilación del cuerpo de un módulo es relativamente pequeño; si únicamente esa unidad necesita ser recompilada y la aplicación religada. Sin embargo, el costo de recompilar la interface de un módulo es relativamente alto. Especialmente con lenguajes fuertemente tipificados, ya que, hay que recompilar la interface del módulo, su cuerpo, todos los otros módulos que dependen de esta interface, los módulos que dependen de estos módulos, y así sucesivamente. Así, para programas muy grandes (dado que nuestros ambientes de desarrollo no apoyan la compilación incremental), un simple cambio en una interface de módulo podría resultar en muchos minutos si no es que horas de recompilación. Obviamente un administrador de desarrollo no puede permitir que suceda frecuentemente una recompilación masiva. Por esta razón la interface de un módulo debería ser tan estrecha como sea posible. Nuestro estilo es ocultar tanto como se pueda en la implementación de un módulo.

El desarrollador debe por lo tanto balancear dos intereses técnicos; el deseo de encapsular abstracciones, y la necesidad de hacer abstracciones seguras visibles a otros módulos. Parnas, Clements, y Weiss ofrece la guía siguiente: Los detalles de sistema que son probables para cambiar independientemente deben ser los secretos de módulos separados; las suposiciones únicas que deben aparecer entre módulos son esas que se consideran inverosímiles para cambiar. Cada estructura de datos es privada a un módulo; esta estructura puede ser directamente accesada por uno o más programas dentro del módulo pero no por programas fuera del módulo. Cualquier otro programa que requiera información de la estructura datos que almacena un módulo debe obtenerla por llamar programas del módulo. En otras palabras, afanar en construir módulos que son cohesivos (agrupando abstracciones lógicamente conexas) y libremente acoplados (por minimizar las dependencias entre módulos); Desde esta perspectiva, nosotros podemos definir Modularidad como se indica a continuación.

Modularidad es la propiedad de un sistema, que se ha descompuesto en un conjunto de módulos cohesivos y libremente acoplados.

Así, los principios de abstracción, encapsulación, y modularidad son sinérgicos. Un objeto provee un límite claro alrededor de una abstracción, y ambos encapsulación y modularidad provee barreras alrededor de esa abstracción.

Dos puntos técnicos adicionales que pueden afectar las decisiones de modularizar. El primero es que los módulos sirven comúnmente como las unidades elementales e indivisibles de software que puede reusarse a través de aplicaciones, un desarrollador podría escoger empaquetar clases y objetos en módulos de una forma que hace su reuso conveniente. El segundo, muchos compiladores generan código objeto en segmentos, uno para cada modulo. Por lo tanto, puede haber límites prácticos sobre el tamaño de módulos individuales. Con respecto a la dinámica llamadas de subprograma, la colocación de

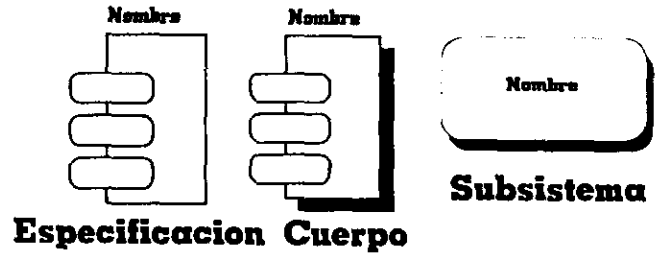


Figura 2-32 Iconografia para Modulos

declaraciones dentro de módulos puede afectar mucho la localidad de referencia y así el comportamiento de comunicación del sistema virtual de memoria.

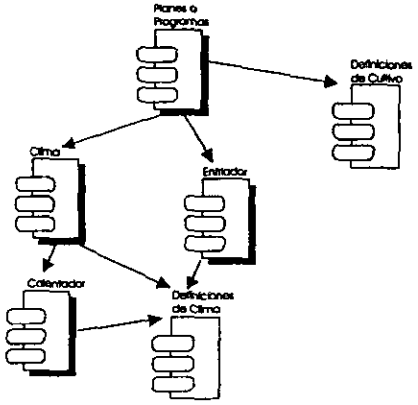


Figura 2-33 Diagrama de Modulos de un Sistema de Jardineria Hydroponico

Los puntos no-técnicos pueden afectar también las decisiones de modularizar. Típicamente las asignaciones de trabajo en un equipo de desarrollo, se dan sobre un módulo, basadas en módulos o por las interfaces entre partes diferentes de la organización de desarrollo. La Seguridad puede también ser un punto: la mayoría del código puede considerarse no clasificado, pero otro código que puede considerarse oculto o clasificado, que es mejor puesto en módulos separados.

El hallazgo de los derechos de clases y los objetos y entonces organizarlos en módulos separados. Son decisiones de diseño en su mayor parte independiente del sistema, La identificación de clases y objetos es parte del diseño lógico del sistema, pero la identificación de módulos es la parte del diseño físico del sistema. Uno no puede hacer todas las decisiones lógicas de diseño antes de hacer todas las físicas, o viceversa; más bien, estas decisiones de diseño suceden iterativamente

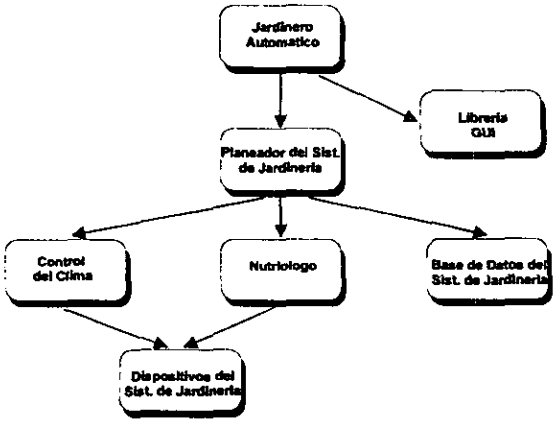


Figura 2-34 Diagrama de Subsistemas de un Sistema de Jardineria Hydroponico

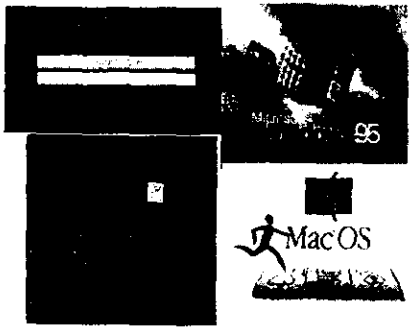
Los diagramas de clases y los diagramas do objetos son utilizados para documentar el diseño lógico de un sistema la notación para un diseño físico de sistemas consiste en módulos que se utilizan concretamente en el diseño los íconos que utilizaremos para representar los módulos y subsistemas serán los siguientes ver figura 2-32.

Un diagrama de módulos es utilizado para mostrar la ubicación de clases y objetos en módulos dentro del diseño físico de un sistema un solo diagrama de módulos puede representar todo o parte de la arquitectura modular de un sistema ver la figura 2-33

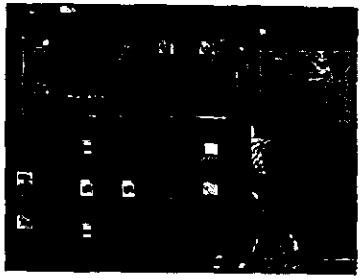
Como se podrá observar el nombre del módulo se pone encima del icono cada nombre del módulo deberá ser único al subsistema que lo engloba un nombre no adornado representa que es privado al sistema que lo engloba, dentro de un rectángulo representa un módulo que es exportado desde un subsistema, uno subrayado representa un módulo que importado desde otro subsistema. La única relación que podremos tener entre módulos es la dependencia de compilación representada por una línea direccionada a la cual se le puede agregar una etiqueta.

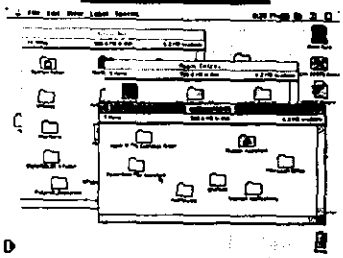
Los diagramas de subsistemas representan sectores de módulos relacionados lógicamente y son muy útiles para programas a gran escala (software industrial-fuerte). Sirven para entender la arquitectura física general de un sistema contenido dentro de un diagrama de módulos, un diagrama de módulos solo puede contener módulos o subsistemas pero no ambos, ver figura 2-34

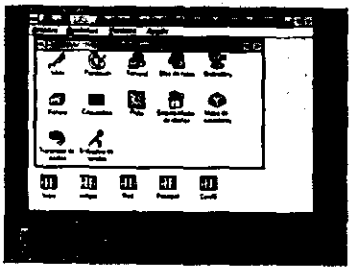
CAPITULO III

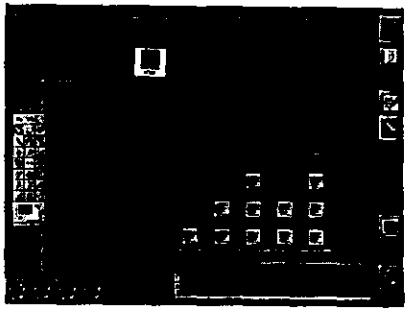


ANALISIS DE UNA INTERFACE USUARIO GRAFICA









Este capítulo comienza con la explicación de que es una interface humano-computadora sus elementos, el porque la importancia actual de las interfaces usuario gráficas, después donde se usan las interfaces; enseguida veremos puntos de diseño de una interface principiando con una revisión de varias teorías, concentrándose en el modelo sintáctico - semántico objeto – acción (SSOA). Para entonces continuar con la frecuencia de uso, perfiles de tarea, y estilos de interacción. Posteriormente se revisara la teoría del color, así como el manejo de fuentes para finalizar con la revisión de algunos puntos legales.

¿QUÉ ES UNA INTERFACE HUMANO-COMPUTADORA? Y SUS ELEMENTOS

La interfaz humano-computadora es el conjunto de elementos que permiten a un usuario interactuar con un sistema computarizado; dicha interfaz humano-computadora se integra de elementos de equipo (hardware), como desplegados de rayos catódicos, teclados, ratones (mouse), etc, y elementos de programación (software), como iconos, menús, formas de entrada de datos, etc. En esta sección nos enfocaremos al aspecto de la programación humano-computadora desde el punto de vista de factores humanos. Nuestro trabajo se enfoca en primera instancia hacia los desarrolladores de interfaz humano-computadora, planteándoles el punto de vista de las necesidades del usuario, con lo que buscamos contribuir a la concepción de sistemas y al diseño centrados en el usuario. Sin embargo, esperamos que el usuario de sistemas computarizados también se beneficie al encontrar algunos fundamentos ergonómicos, psicológicos y cognoscitivos de sus gustos y necesidades.

Figura 3-35 Ejemplo de una Interface Usuario Grafica, una de las primeras y más famosas; La de Mac

LA IMPORTANCIA DE LAS INTERFACES HUMANO COMPUTADORA

Las empresas lideres desarrolladoras de computadoras y de sistemas reconocen que la interfaz humano-computadora, es crítica para determinar el éxito o el fracaso de un producto, y que los sistemas que son más fáciles de usar tendrán una ventaja competitiva en la recuperación de información, la automatización de oficina, y la computación personal. El éxito de un sistema computacional, es decir, su aplicación para la solución de problemas reales, radica en gran medida en la calidad de su interfaz con el usuario, la cual debe ser capaz de proporcionar mecanismos para una explotación fácil y eficiente del sistema.

El software de interface es inherentemente difícil de escribir debido entre otras cosas, a las variaciones de los requerimientos de diversas aplicaciones, a las variaciones en las preferencias de los usuarios y a la necesidad de ofrecer portabilidad entre diferentes plataformas y sistemas operativos.

Por lo anterior, desde hace algunos años se ha venido dedicando cada vez más tiempo y esfuerzo en el análisis, diseño e implementación de interfaces eficaces, lo cual se ha convertido hoy en día en una interesante y relevante área de investigación y desarrollo.

PELEANDO POR EL USUARIO

La frustración y ansiedad son parte de la vida diaria para muchos usuarios de sistemas de información computarizados. *Ellos se esfuerzan por aprender comandos complejos o por entender sistemas de selección de menú, que se supone son para ayudarlos a hacer su trabajo. Algunas gentes encuentran serios casos de shock (rechazo) de computadora, terror a la terminal, o neurosis de red.* Esto hace que ellos eviten usar sistemas computarizados. Estos son los males que se desarrollan más comúnmente en esta era de la electrónica. Mientras que por el contrario los usuarios satisfechos de un sistema computarizado frecuentemente tienen la experiencia de claridad, de sentirse competentes e inclusive de sentirse poderosos al hacer su trabajo con dicho sistema.

Los investigadores han mostrado que un diseño apropiado de la interface humano-computadora puede hacer una diferencia considerable en tiempo de aprendizaje, velocidad de ejecución, porcentaje de error y satisfacción del usuario. Los científicos de computadora e información han probado alternativas de diseño para su impacto sobre estas medidas humanas de desempeño. Los programadores, y los equipos de control de calidad comienzan a ser más cautelosos y poner mayor atención a los puntos de implementación que garantizan interfaces de usuario de alta calidad. Los administradores de centros de computo deben

jugar un papel activo en asegurar que las instalaciones de hardware y software provean un servicio de alta calidad a sus usuarios.

En suma, el uso diverso de computadoras en hogares, oficinas, fábricas, hospitales, centrales de control eléctrico, hoteles, bancos, etc. Estimula el interés generalizado en puntos de la **ingeniería de factores humanos.** La ingeniería de factores humanos es la disciplina que ha retomado los métodos clásicos de la investigación en psicología para aplicarlos al estudio del comportamiento humano en los sistemas computarizados, con el objetivo de determinar los factores psicológicos, cognoscitivos, ergonómicos y como intervienen cuando un humano interactúa con la computadora.

El diseño de interfaces usuario gráficas, se vio como la pintura que se puso al final de un proyecto, ahora parece ser el marco de acero sobre el cual se construye la estructura. Sin embargo, una conciencia de los problemas y un deseo de hacer el bien no son suficientes. **Los diseñadores, administradores y programadores deben estar dispuestos a caminar juntos y pelear por el usuario. Los enemigos a vencer incluyen inconsistentes lenguajes de comandos, secuencias de operación confusas, formatos caóticos de exhibición, terminología inconsistente, instrucciones incompletas, complejos procedimientos de recuperación de error, y mensajes de error extraviados o amenazadores.** La batalla no será ganada por una argumentación fuerte sobre la "Amigabilidad de usuario" de sistemas computacionales o por parciales reclamos de que *"mi diseño es más natural que su diseño."* La victoria será para la gente quien tome un enfoque disciplinado, reiterativo, y empirico del estudio del desempeño humano en el uso de sistemas interactivos.

Más y más, desarrolladores de sistema, y administradores colectan datos del desempeño de los usuarios, distribuyendo encuestas de satisfacción subjetivas, invitando a los usuarios a participar en equipos de diseño, conduciendo estudios repetidos de campo para propuestas novedosas, y usando datos de los estudios de campo para apoyar a las organizaciones en la toma de decisiones.

USOS DE LA INTERFACE USUARIO GRAFICA

El rápido crecimiento del interés en el diseño de Interfaz Usuario Gráfica es de alcance internacional. En los Estados Unidos Asociaciones como, La Asociación para Computo y Maquinaria (ACM) Association for Computing Human Machinery, El Grupo de Interés Especial en la Interacción Humana de Computadora (SIGCHI) Special Interest Group in Computer Human Interaction tuvo más de 5000 miembros en 1991 y era el más rápido crecimiento .

El crecimiento constante en el diseño de interfaz del usuario se aplica notablemente a sistemas diversos como lo son: Editores de textos, procesadores de palabras y formateadores de documentos que se utilizan rutinariamente en muchas oficinas, y algunos negocios ahora usan programas de dibujo, publicación asistida por computadora, y diseño de gráficas, correo electrónico, tableros de anuncios, conferencia por computadora ha proveído

nuevos medios de comunicación. El procesamiento de imágenes, la recuperación, y almacenamiento se utilizan en aplicaciones desde la medicina hasta exploración espacial. Las estaciones de trabajo de simulación y visualización científica permiten entrenamiento barato seguro además de experimentación. Las hojas de cálculo electrónicas son de uso común. Los sistemas de apoyo a la toma de decisiones sirven para analistas de muchas disciplinas. Las computadoras se utilizan frecuentemente para la educación y el entrenamiento de tarea. Cabe mencionar en este punto la explosión que ha tenido la INTERNET debido al WWW gracias a su interface gráfica (Ver la figura 3-2). La automatización

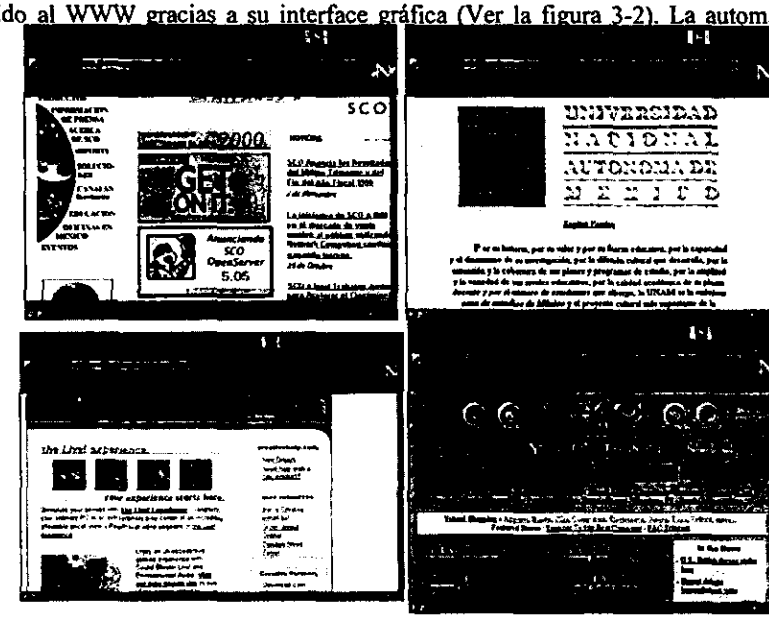


Figura 3-2 Aqui se puede observar que la explocion de la internet ha sido por la WWW que es una GUI

doméstica y el control ambiental son las zonas de investigación activa. El acceso público de información, desde los quioscos de museo a la recuperación bibliográfica, se utiliza ampliamente. Los sistemas comerciales incluyen inventario, personal, reservas, tránsito aéreo, y control eléctrico de utilidad. Las herramientas de Ingeniería de Software Asistido por Computadora ambientes de programación y herramientas técnicas que permiten los rápidos prototipos, como las Areas de CAD-CAM (diseño asistido por computadora, y fabricación asistida por computadora) y puestos de trabajo técnicos. Para la mayoría de los aparatos de consumo electro domésticos, tales como videograbadoras, telefonía, cámaras, y aparatos. El arte, música, deportes, y la diversión todos estan asistidos o mejorados por los sistemas de computadora.

A un nivel individual, los doctores pueden hacer un diagnóstico más exacto, los niños pueden aprender más efectivamente, los artistas gráficos pueden explorar más posibilidades creativas, y los pilotos pueden volar más aviones sin riesgo. Sin embargo, nosotros todavia

debemos arreglar la frustración, temor, y el fallo que resulta cuando un usuario encara complejidad excedente, terminología incomprensible, o esquemas caóticos en un sistema

METAS DE LA INGENIERIA DE SOFTWARE

Antes de empezar a tratar sobre los puntos de análisis y diseño de las interfaces usuario gráficas (GUI), hagamos un breve recordatorio, que servirá como preámbulo a las siguientes secciones. Como se menciono en el capítulo I; todo surgió por construir sistemas computarizados, más grandes y en menor tiempo, como resultado de buscar solución a estos problemas surge la ingeniería de software. La cual entre otras cosas se encarga de los métodos y modelos para el desarrollo y mantenimiento, por ejemplo, técnicas y principios para la especificación, diseño e implementación de sistemas de software incluyendo notaciones y modelos de proceso. Es precisamente aquí donde podemos ubicar el diseño de interfaces usuario gráficas, como parte de las interfaces humano computadora. Así como el diseño de GUI's es parte de la ingeniería de software debemos tomar en cuenta algunos aspectos relevantes de esta ciencia: **(1) Funcionalidad apropiada, (2) Confiabilidad, Disponibilidad, Seguridad e Integridad de datos, (3) Estandarización, Integración, Consistencia y Portabilidad.**

FUNCIONALIDAD APROPIADA

El primer paso es comprobar la funcionalidad necesaria de tareas y subtareas así como las tareas auxiliares que deben efectuarse. Las tareas frecuentes son fáciles de determinar, pero las tareas ocasionales, las tareas excepcionales para las condiciones de emergencia, y las tareas de reparación para arreglar errores en el uso del sistema son más difíciles de descubrir. El análisis de tarea es central, porque los sistemas con la funcionalidad inadecuada frustran al usuario y se rechazan frecuentemente o son subutilizados. Si la funcionalidad es inadecuada, no importa cuan bien la interfaz humana se diseñe. Por otra parte la funcionalidad excesiva es también un peligro, y probablemente la equivocación más común de los diseñadores, porque la confusión y la complejidad hacen que la implementación, el mantenimiento, el aprendizaje, y su uso sea más difícil.

CONFIABILIDAD, DISPONIBILIDAD, SEGURIDAD, E INTEGRIDAD DE LOS DATOS

Un vital segundo paso es asegurar la confiabilidad apropiada del sistema; es decir los comandos deben funcionar como se especifica, la visualización de los datos deben reflejar los contenidos de la base de datos, y las actualizaciones deben aplicarse correctamente. La confianza del usuario de sistemas es frágil; una experiencia con resulta-

dos inesperados, desalentara el uso del sistema por largos periodos de tiempo. La arquitectura del software, y los componentes de hardware, y el soporte de red deberá asegurar alta disponibilidad. Si el sistema no es disponible o introduce errores, entonces no importa cuan bien la interfaz humana se diseñe, también se deberá poner atención para asegurar privacidad, seguridad, e integridad de los datos; La protección debe proveerse para los accesos indeseados, la destrucción inadvertida de datos, o alteraciones maliciosas.

ESTANDARIZACIÓN, INTEGRACIÓN, CONSISTENCIA, Y PORTABILIDAD

Como el número de usuarios y paquetes de software aumenta, las presiones (para y beneficios) de la estandarización también crecen. Las pequeñas diferencias entre sistemas no solamente aumentan el tiempo de aprendizaje, también pueden conducir a errores molestos y peligrosos. Las diferencias grandes entre sistemas requieren de readiestramiento considerable y agobian a los usuarios de muchas maneras. La incompatibilidad de los formatos de almacenamiento, las diversas versiones de hardware y software ocasionan frustración, ineficacia, y retraso. Los diseñadores deben decidir si las mejoras que ellos ofrecen son lo suficientemente útiles para el desplazamiento y la interrupción a los usuarios.

La estandarización se refiere a las características comunes de la interfaz de usuario a través de *múltiples aplicaciones. Las Computadoras de Apple (1987)* exitosamente desarrollaron la primera norma que era aplicada ampliamente por miles de desarrolladores, habilitando así a los usuarios el aprendizaje de múltiples aplicaciones rápidamente. El usuario común de IBM accede a las *especificaciones hasta (1989, 1991); como se puede* observar se tomaron varios años para que surgieran sus beneficios.

En el ambiente UNIX, el lenguaje de comandos fue el estándar desde el principio (con algunas divergencias), pero ahora hay varios compitiendo por normas para la interfaz de usuario gráfica. El UNIX fue un líder e inspiración en la estandarización de formatos de datos; para habilitar la integración a través de numerosos programas de aplicación y herramientas de software, además de permitir portabilidad a través de diferentes plataformas de hardware.

La consistencia principalmente se refiere a sucesiones comunes de acción, términos, unidades, *esquemas, color, tipografía, llamadas, etc., dentro de un programa de aplicación;* se extiende naturalmente para incluir compatibilidad a través del rol del programa de aplicación o sistemas no basados en computadora. Las normas Internacionales emergentes pueden tener una influencia profunda sobre *diseñadores.*

La portabilidad se refiere a la potencialidad para convertir datos y para compartir la interfaz del usuario a través de múltiples ambientes de software y hardware. Organizar la portabilidad es un desafío para diseñadores quienes deben lidiar con diferentes resoluciones y tamaños de dispositivos de despliegue, las capacidades de color, dispositivos *indicadores, formatos de datos, y más.*

Algún UIMSS de ayuda generando código para Macintosh, IBM PC, UNIX, y otros ambientes para que las interfaces sean similares en cada ambiente. Los archivos estándares de texto (en ASCII) pueden trasladarse fácilmente a través de ambientes, pero las imágenes, gráficas, las hojas de cálculo, imágenes de vídeo, son mucho más difíciles convertir. Aunque en la actualidad ya no es así.

MOTIVACIONES PARA EL DISEÑO EN FACTORES HUMANOS

El enorme interés en factores humanos de sistemas interactivos proviene del reconocimiento complementario del diseñó pobre de muchos sistemas actuales y del deseo auténtico de los desarrolladores para crear sistemas elegantes que sirvan a los usuarios efectivamente. Este aumentó de interés emana de cuatro fuentes principales:

- la vida de los sistemas críticos
- los usos industrial y comercial
- las aplicaciones de oficina, hogar, y entretenimiento. y
- los sistemas exploratorios, creativos, y colaboradores.

VIDA DE LOS SISTEMAS CRÍTICOS

La vida de los sistemas críticos incluyen los de control de tránsito aéreo, y reactores nucleares, las utilidades para el control: de la energía eléctrica, de naves espaciales tripuladas, de los despachadores de policías o bomberos, operaciones militares, y monitoreo de instrumentos médicos. En estas aplicaciones, los altos costos son de esperar, sin embargo estos sistemas deberían rendir alta confiabilidad y eficacia. Los períodos largos de entrenamiento son aceptables para obtener rápido, el rendimiento libre de error aun cuando el usuario sufra una tensión nerviosa. La satisfacción subjetiva es un punto menos, porque los usuarios están bien motivados. La retención obtenida es por el uso frecuente de funciones comunes y de práctica para acciones de emergencia.

USOS INDUSTRIAL Y COMERCIAL

Los usos típicos industrial y comercial incluyen operaciones de bancos, aseguradoras, ordenes de entrada, administración de inventario, reservaciones de aerolínea y de hotel, alquiler de coches, utilidades de facturación, administración de tarjetas de crédito y de terminales de punto de ventas. En este caso los costos forman muchos fallos; por ejemplo el costo más bajo puede preferirse aún cuando haya algún sacrificio en la

confiabilidad. El tiempo de entrenamiento del operador es caro, y la facilidad de aprender es importante. La salida del sistema para la velocidad de rendimiento y el porcentaje de error son decididos por el costo total sobre la vida del sistema. La satisfacción subjetiva es de importancia modesta; la retención es obtenida por el uso frecuente. La velocidad de rendimiento llega a ser central para la mayoría de estas aplicaciones a causa del alto volumen de transacciones, pero la fatiga o el agotamiento del operador es un interés legítimo.

APLICACIONES DE OFICINA, HOGAR, Y ENTRETENIMIENTO.

La expansión rápida de aplicaciones oficina, hogar, y las de diversión es la tercera fuente de interés en factores humanos. Las aplicaciones de computo personal incluyen procesadores de palabras, maquinas de transacción automática, juegos de vídeo, paquetes educacionales, recuperación de información, correo electrónico, conferencia por computadora, y administración de pequeños negocios. Para estos sistemas, la facilidad de aprendizaje, el bajo porcentaje de error, y la satisfacción subjetiva son muy importantes porque su uso es frecuentemente discrecional y la competencia es fiera. Si los usuarios no pueden triunfar rápidamente, ellos abandonarán su uso y trataran de competir con otro paquete. En casos donde el uso es intermitente, la retención es probable que sea defectuosa, entonces la ayuda en línea llega a ser importante. Elegir los derechos de funcionalidad es difícil. Los novatos son mejor servidos por un conjunto simple y limitado de acciones; pero, los usuarios con experiencia, su deseo es una funcionalidad más extensiva y rápido rendimiento. El diseño estructurado por capas o niveles es un enfoque elegante de evolución

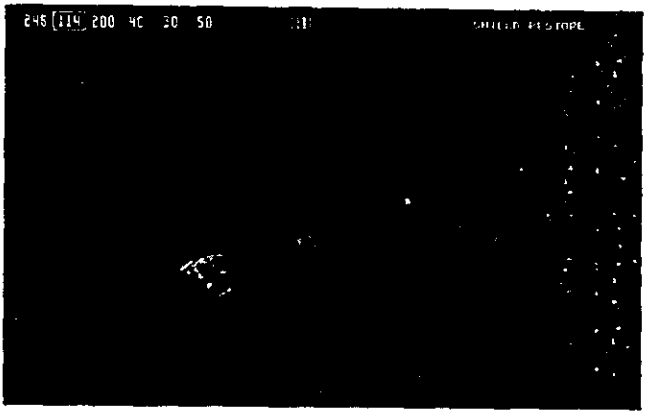


Figura 3.3 Ejemplo de una Interface Usuario Grafica una de un video juego

desde el usuario novato hacia el usuario experto. El bajo costo es importante a causa de que competencia es vivaz, pero el diseño extensivo y la pruebas pueden amortizarse sobre el gran número de usuarios.

SISTEMAS EXPLORATORIOS, CREATIVOS, Y COOPERATIVOS

Una fracción creciente del uso de computadora se dedica al apoyo de empresas humanas intelectuales y creativas. Las enciclopedias electrónicas, los buscadores de base de datos, la escritura colaboradora, la formación de hipótesis estadística, la toma de decisiones del negocio, y la presentación gráfica de los resultados de la simulación científica son ejemplos de ambientes exploratorios. Los ambientes creativos incluyen conjuntos de herramientas de escritura o workbenches (espacios de trabajo), sistemas de diseño de arquitectura o automóviles, estaciones de trabajo de programadores o artistas, y sistemas de composición de música. Los sistemas expertos ayudan inteligentemente en el diagnóstico médico, en hacer decisiones financieras, en las maniobras de órbitas satelitales, y notificaciones militares.

En los sistemas cooperativos se habilitan dos o más personas para trabajar juntas aún cuando los usuarios estén separados por el tiempo y el espacio, mediante el uso de texto electrónico, voz, y de video correo; a través de sistemas electrónicos de reunión que facilitan reuniones cara a cara; o mediante software de grupos (groupware) que activa colaboradores remotos para trabajar sobre un documento, hoja de cálculo, o concurrencia de imagen. En estos sistemas, los usuarios pueden ser inteligentes en el domino de tarea pero novatos en los conceptos subyacentes de computadora. Su motivación es frecuentemente alta, pero son altas sus expectativas. Las tareas del tipo prueba son más difíciles de escribir a causa de la naturaleza exploratoria de estas aplicaciones. El uso puede fluctuar desde el ocasional hasta el frecuente. En suma, es difícil diseñar y evaluar estos sistemas. *A lo mejor, los diseñadores pueden perseguir la meta de desvanecer o hacer transparente la computadora hacia el usuario a través de su interface de usuario para llegar hasta el punto en que el usuario se absorba completamente en el dominio de tarea.* Esta meta parece tener eficacia la mayoría de veces cuando la computadora provee una representación del mundo de acción a través de la manipulación directa. Entonces las tareas son efectuadas por los gestos o selecciones familiares rápidas, con retroalimentación inmediata y un nuevo juego de elecciones.

EL DISEÑO DE INTERFACES HUMANO-COMPUTADORA

La guía para diseñadores comienza a surgir en forma de (1) los modelos o teorías de alto nivel, (2) principios de nivel medio, (3) directivas específicas y prácticas, y (4) estrategias para probar. *Las teorías o los modelos ofrecen una estructura o lenguaje para discutir puntos que son de aplicación independiente, mientras que los principios de nivel medio son usados en alternativas de diseño más específicas. Las directivas prácticas proveen recordatorios útiles de reglas descubiertas por diseñadores previos. La evaluación temprana de prototipos fomenta exploración y habilita pruebas iterativas y el rediseño para corregir decisiones impropias.* El ensayo de aceptación es la prueba de fuego para

determinar si un sistema está listo para la distribución; su presencia puede ser vista como un desafío, pero es también un regalo a diseñadores ya que pueden establecer medidas claras de éxito.

En muchos sistemas contemporáneos, hay una oportunidad grandiosa de mejorar la interfaz humana. Las visualizaciones atropelladas, procedimientos complejos y tediosos, lenguajes de comandos inadecuados, sucesiones inconsecuentes de acciones, y la retroalimentación informativa insuficiente puede generar inquietud y tensión debilitante que conduce al rendimiento pobre, errores frecuentes, errores menores serios y ocasionales, y descontento de tarea.

TEORÍAS DE ALTO NIVEL

Muchas teorías se necesitan para describir los múltiples aspectos de sistemas interactivos. Algunas teorías son explicativas: Estas son útiles en observar el comportamiento, y describir la actividad, concibiendo diseños, comparando conceptos de alto nivel de dos de diseños, y partes relacionadas con el entrenamiento. Otras teorías son predictivas: Estas proveen a los diseñadores formas para comparar diseños propuestos para el tiempo de ejecución o porcentaje de error.

MODELO CONCEPTUAL, SEMÁNTICO, SINTÁCTICO, Y LÉXICO

Un modelo atrayente y fácilmente comprensible es el de cuatro niveles que desarrollaron Foley y Van Dam aproximadamente a finales de 1970.

1.- **El nivel conceptual** es el modelo mental del usuario del sistema interactivo. Por ejemplo hay dos modelos conceptuales para edición de texto que son editores de línea y editores de pantalla completa.

2- **El nivel semántico** describe los significados transmitidos por los comandos introducidos por el usuario y por la visualización de salida de la computadora.

3- ***El nivel sintáctico*** *define como unidades (palabras) que transmiten semánticas* ensambladas en una frase completa que instruye a la computadora para realizar una cierta tarea.

4.-**El nivel léxico** distribuye en los dispositivos dependientes y en los mecanismos precisos para los cuales el usuario especifico en la sintaxis.

Este enfoque es conveniente para diseñadores porque es de naturaleza descendente, es fácil de explicar, y es igual a la arquitectura de software, y permite usar modularidad durante el diseño. Los diseñadores pueden esperar moverse desde el nivel conceptual hasta el nivel léxico, y registrar cuidadosamente las representaciones entre niveles.

GOMS Y EL MODELO DE NIVEL DE TECLAZO

Card, Moran, y Newell entre (1980, 1983) propusieron el modelo (Goals, Operators, Methods and Selection Rules: conocidas como **GOMS** por sus siglas en ingles) metas, operadores, métodos, y reglas de selección y el modelo de nivel de Teclazo. Ellos postularon que los usuarios formulan metas (editar documento) y submetas (inserción de palabra) y que ellos lo logran por utilizar métodos o procedimientos para realizar cada meta (mover el cursor a la ubicación deseada por una sucesión de teclas de flecha). Los operadores son *"elementales actos de percepción, motores, o cognitivos, cuya ejecución es necesaria para cambiar cualquier aspecto del estado mental del usuario o para afectar el ambiente de tarea"* (pulsar tecla de flecha arriba, mover la mano hasta al ratón, recordar el nombre del archivo, verificar que el cursor está en el fin de archivo). Las reglas de selección son las estructuras de control para elegir entre varios métodos disponibles para realizar una meta (borrar por repetición de la tecla retroceso (backspace)) versus borrar por poner marcadores al comienzo y fin de la región y después presionando el botón de borrado).

El modelo de nivel de teclazo es un intento de predecir el tiempo de rendimiento libre de error, el rendimiento experto de tareas por la totalización del tiempo empleado tecleando, indicando, dibujando, pensando, y esperando para que el sistema responda. Estos modelos se concentran en los usuarios expertos y en el rendimiento libre de error, con menor énfasis en el aprendizaje, resolución del problema, tratamiento de errores, satisfacción subjetiva, y retención.

SIETE DE ETAPAS DE ACCIÓN

Norman (1988) ofrece siete de etapas de acción como un modelo de interacción humano-computadora:

1. Formar la meta
2. Formar la intención
3. Especificarla acción
4. Ejecutar la acción
5. Percibir el estado del sistema
6. Interpretar el estado del sistema
7. Evaluar el resultado

Además, las siete etapas del modelo conducen naturalmente a la identificación del *"abismo de ejecución" (la diferencia entre las intenciones del usuario y las acciones permisibles) y el "abismo de evaluación" (la diferencia entre la representación del sistema y las expectativas del usuario).*

Este modelo condujo a Norman a sugerir cuatro buenos principios de diseño. El primero, el estado y las alternativas de acción deberán ser visibles. El segundo, se deberá tener un modelo conceptual bueno con un consistente sistema de imágenes. El tercero, la interfaz deberá incluir representaciones buenas que den a conocer las relaciones entre etapas. El cuarto, el usuario deberá recibir retroalimentación continua. Norman hace gran énfasis en el estudiar errores. El describe como frecuentemente los errores ocurren en movimiento desde metas a intenciones a acciones y a ejecuciones.

CONSISTENCIA MEDIANTE GRAMATICAS

Una meta importante para diseñadores es una interfaz de usuario uniforme(consistente), sin embargo, la definición de consistencia es escurridiza y tiene múltiples niveles que a veces entran en conflicto; también, es algunas veces ventajoso para ser inconsistente. **El argumento para la consistencia es que un lenguaje de comandos o el juego de acciones debería ser ordenado, predecible, describible por unas pocas reglas, y por lo tanto fáciles de aprender y retener. Es muy usado para interfaces de lenguajes de comandos.**

SINTÁCTICO - SEMÁNTICO MODELO DE CONOCIMIENTO DE USUARIO

Las distinciones entre la sintaxis y la semántica han sido hechas largamente por los escritores de compiladores quienes buscaron separar la salida del parsing (análisis sintáctico) de la entrada texto de las operaciones que eran llamadas por el texto. El conocimiento semántico es separado en los conceptos de tarea (objetos y acciones) y los conceptos de computadora (objetos y acciones). Una persona puede ser un experto en los conceptos de computadora, pero un novato en los conceptos de tarea, y viceversa. El **modelo sintáctico - semántico objeto - acción (SSOA)** fue propuesto por Shneiderman, (1980-1983) el cual expondremos a continuación.

CONOCIMIENTO SINTÁCTICO

Cuando se usa un sistema de computadora, los usuarios deben mantener una profusión de dispositivos - detalles dependientes en su memoria humana. Estos detalles sintácticos de bajo nivel incluyen el conocimiento de que acción borra un carácter (delete, backspace, CTRL - H, botón de derecho del ratón, ESC), cual acción inserta una nueva línea después de la tercera línea de un archivo de texto (CTRL-I, tecla INSERT), que icono hace desfilar las líneas de texto hacia delante, que abreviaturas son permisibles, y que teclas de función producen una pantalla previa.

El aprendizaje, uso y la retención de este conocimiento es impedida por dos problemas. El primero, estos detalles varían a través de los sistemas de una manera caprichosa. El segundo, adquirir el conocimiento sintáctico es frecuentemente una pugna porque la arbitrariedad de estas características menores de diseño reduce mucho la eficacia del aprendizaje por asociación de parejas. El conocimiento sintáctico es transmitido comúnmente por el ejemplo y uso repetido. Un problema adicional con el conocimiento sintáctico, en algunos casos, yace en la dificultad de proveer una estructura jerárquica o una estructura modular uniforme para arreglar la complejidad. Por ejemplo, como es que un usuario recuerda los detalles de utilizar un sistema de correo electrónico: presioné RETURN para terminar párrafo, CTRL - D para terminar una carta, Q para abandonar el subsistema de correo electrónico, y logout para terminar la sesión. El usuario inteligente de computadora comprende estas cuatro formas de terminación como comandos en el contexto del sistema completo, pero el novato puede confundirse por cuatro situaciones aparentemente similares que tienen formas sintácticas radicalmente diferentes. Una dificultad final es que el conocimiento sintáctico es dependiente del sistema. Un usuario quien cambia de una de máquina a otra puede encontrarse con teclado, comandos, uso de las teclas de función, y secuencias de acciones diferentes. Seguramente puede haber alguna superposición. Por ejemplo, las expresiones aritméticas podrían ser los mismos en dos de lenguajes; desgraciadamente, sin embargo, las pequeñas diferencias pueden ser las más molestas. Por ejemplo en un sistema se usa K para guardar un archivo y en otro el uso de K para borrar el archivo, o S para grabar versus S para enviar.

CONOCIMIENTO SEMÁNTICO - CONCEPTOS DE COMPUTADORA

El conocimiento semántico en la memoria humana a largo plazo tiene dos componentes: conceptos de tarea y conceptos de computadora. El conocimiento semántico se transmite mostrando ejemplos de uso, ofrece un diseño o la teoría general, relacionando los conceptos al conocimiento previo por analogía, describiendo un modelo concreto o abstracto, e indicando ejemplos de uso incorrecto. Hay una atracción para mostrar el uso incorrecto para indicar claramente los límites de un concepto, pero hay también un peligro, ya que el aprendiz puede confundir el uso correcto e incorrecto. Las plantillass son frecuentemente útiles en mostrar las relaciones entre escribir una carta de negocios que utiliza un programa de computación, **los usuarios tienen que integrar suavemente las tres de formas de conocimiento. Ellos deben tener el concepto alto de nivel de escribir (acción de tarea) una carta (el objeto de tarea), reconocer que la carta se almacenará como un archivo (el objeto de computadora), y saber los detalles del comando salvar (acción de computadora y conocimiento sintáctico).** Los usuarios deben ser fluidos con el nivel medio de concepto de componer una frase y deben reconocer el mecanismo para comenzar a escribir, y terminar una frase.

Finalmente, los usuarios deben saber los detalles apropiados de bajo nivel como deletrear cada palabra (tarea), comprender el movimiento del cursor sobre la pantalla

(concepto de computadora), y saber que teclas pulsar para cada carta (conocimiento sintáctico). La meta es disminuir los conceptos sintácticos y conceptos de computadora mientras se presenta una representación visual de las tareas (acciones y objetos) que es el corazón del enfoque de diseño de la manipulación directa. (ver la sección de Manipulación Directa). Integrar las tres de formas de conocimiento, los objetos y acciones, y los múltiples niveles de conocimiento semántico es un desafío considerable que requiere de gran motivación y concentración. Los materiales educativos que facilitan la adquisición de este conocimiento son de difícil diseño.

RECONOCIENDO LA DIVERSIDAD

La notable diversidad de capacidades humanas, experiencias, estilos cognitivos, y personalidades desafían al diseñador interactivo de sistemas. Este efecto se multiplica por la amplia gama de situaciones, tareas, y frecuencias de uso, el juego de posibilidades llega a ser enorme. El diseñador puede responder, por elegir desde un espectro de estilos de interacción. Un preescolar ejecutando un juego gráfico de computadora es una manera de comparar o hacer referencia a un bibliotecario que hace búsquedas bibliográficas por patrones ansiosos y apresurados. Similarmente, un programador profesional que utiliza un nuevo sistema operativo es una manera de comparar un altamente entrenado y experimentado controlador de trafico aéreo. Finalmente, un estudiante que aprende a través de una lección asistida por computadora es una manera de comparar a un oficinista de reservaciones de hotel que trabaja en servicio a clientes por muchas horas al día.

Estos esbozos realzan las diferencias en las experiencias de conocimiento de los usuarios, entrenamiento en el uso del sistema, frecuencia de uso, y metas, así como también en el impacto del error del usuario. Ningún diseño único podría satisfacer, todos los usuarios y situaciones, así que antes de comenzar un diseño, nosotros debemos hacer la caracterización de los usuarios y situaciones tan precisa y completa como sea posible.

PERFILES DE USO

"*Conocer al usuario*" fue el primer principio de Hansen's (1971 listado de usuarios - principios técnicos). Es una idea simple, pero una meta difícil y desafortunadamente, una meta frecuentemente menospreciada. Nadie argumentaría lo contrario a este principio, pero muchos diseñadores presumen que ellos comprenden a los usuarios y tareas de usuarios. Los diseñadores exitosos son conscientes que la demás gente aprende, piensa, y resuelve problemas de maneras muy diferentes. Para algunos usuarios realmente se les hace más fácil usar tablas que gráficas, con palabras en vez de números, con visualización lenta que con rápida, o con una estructura rígida que con una forma abierta.

¿Idealmente como se debería diseñar?. Todo diseño debería comenzar con una comprensión de los usuarios finales o a quienes esta destinado, incluyendo perfiles de su edad, género, capacidades físicas, educación, antecedentes culturales o étnicos, entrenamiento, motivación, metas, y personalidad. Por ejemplo, una separación genérica seria usuario novato o de primera vez, usuario intermitente inteligente, y el usuario experto - frecuente estos tipos de usuarios podrían conducir a diferentes metas de diseño:

- **Novato o de Primera Vez** La comunidad de usuarios novatos se supone que no tiene ningún conocimiento sintáctico sobre utilizar el sistema y probablemente poco conocimiento semántico del uso de la computadora. Considerando que los usuarios de primera vez saben la semántica de tarea, y los novatos tienen conocimiento somero de la tarea y ambos pueden llegar con ansiedad sobre utilizar computadoras que inhibe su aprendizaje. Superar estas limitaciones es un desafío serio al diseñador. Restringir el vocabulario a un número pequeño familiar, usar términos consistentes es esencial para comenzar a desarrollar el conocimiento del usuario sobre el sistema. El número de posibilidades debería guardarse pequeño, y el usuario novato debería ser capaz de efectuar tareas simples para ganar confianza, para reducir la ansiedad, y para ganar refuerzo positivo del éxito. La retroalimentación informativa sobre la realización de cada tarea es útil, y los mensajes de error deberían ser específicos cuando los errores ocurran.

  Diseñar cuidadosamente manuales de papel y los tutoriales en línea paso a paso puede ser efectivos. Los usuarios intentaran relacionar su conocimiento existente a las tareas, acciones y los objetos en la aplicación, así las distracciones con conceptos de computadora y la sintaxis son una carga extra.

- **Los usuarios intermitentes inteligentes:** Mucha gente conocerá solamente usuarios intermitentes de una variedad de sistemas. Ellos serán capaces de mantener el conocimiento semántico de tarea y los conceptos de computadora, pero ellos tendrán dificultad de mantener el conocimiento sintáctico. La carga de memoria será iluminada por una estructura simple y uniforme en el lenguaje de comandos, menúes, terminología, y así sucesivamente, y por el uso del reconocimiento más que el de recordar. Las sucesiones consistentes de acciones, mensajes significativos, y los frecuentes indicadores (prompts) todos seguramente ayudaran a los usuarios intermitentes inteligentes para que ellos puedan ejecutar sus tareas adecuadamente. Estos usuarios se beneficiarán de pantallas de ayuda en línea para llenar pedazos perdidos de conocimiento sintáctico o semántico de computadora. los manuales de referencia bien organizados también serán útiles.

- **Los usuarios expertos frecuentes:** los usuarios expertos están completamente familiarizados con los aspectos sintácticos y semánticos del sistema y buscan conseguir realizar su trabajo rápidamente. Ellos exigen respuesta rápida, información breve y menos distracción de retroalimentación, y la capacidad para efectuar acciones con simplemente unos golpes de tecla o

selecciones. Cuando una sucesión de tres o cuatro de comandos se ejecuta regularmente, el usuario frecuente es ávido para crear una macro u otra forma abreviada para reducir el número de pasos. Las cadenas de comandos, atajos (Shorcuts) mediante menúes, abreviaturas, y los otros aceleradores son necesarios.

Estas características de estas tres clases de uso deben refinarse para cada ambiente. Diseñar para una clase es fácil; diseñar para varias es mucho más difícil. Otro enfoque para acomodar los diferentes clases de usos está en permitir el control al usuario de la densidad de retroalimentación informativa que el sistema provea. Ya que los novatos requieren más retroalimentación informativa para confirmar sus acciones, considerando los usuarios frecuentes quieren menos distracción de retroalimentación. Similarmente, observamos que las visualizaciones como usuarios frecuentes deben ser más densamente empaquetadas que para los novatos.

PERFILES DE TAREA

Después de cuidadosamente haber sacado el perfil de usuario, los desarrolladores deben identificar las tareas. Cada diseño aceptara el juego de tareas que debe determinarse antes de proceder al diseño, pero muy frecuentemente el análisis de tarea se hace informalmente o implícitamente.

Las acciones de tarea de alto nivel pueden descomponerse en múltiples acciones de tarea de nivel medio que pueden ser refinadas adicionalmente en acciones atómicas que el usuario ejecuta con un único comando, selección de menú, y así sucesivamente. Elegir el juego más apropiado de acciones atómicas es una tarea difícil. Si las acciones atómicas son demasiado pequeñas, los usuarios llegaran a frustrarse por el gran número de acciones necesarias para realizar una tarea de alto nivel. Si las acciones atómicas son demasiado grandes y elaboradas, los usuarios necesitarán muchas acciones con opciones especiales, o ellos no serán capaces conseguir exactamente lo qué desean del sistema.

Las frecuencias relativas de tarea serán importantes en formar, por ejemplo, un juego de comandos o un árbol de menú. Las tareas frecuentemente ejecutadas deberían ser simples y rápidas al efectuarse, uniforme al gasto de alargar alguna tarea infrecuente. La frecuencia relativa de uso es una de las bases para hacer decisiones arquitectónicas de diseño.

Por ejemplo, en un editor de textos,

- las acciones frecuentes podrían ser ejecutadas por teclas especiales, tales como las cuatro flechas de cursor, el INSERT, y DELETE.
- las acciones de frecuencia intermedia podrían ser ejecutadas por una sola tecla plus CTRL, o por una selección de un menú desplegable (pull-down)- ejemplos incluir el carácter subrayado, centro, sangrado, subíndice, o índice sobrescrito.

- las acciones menos frecuentes podrían requerir ir al modo comando y escribir el nombre del comando por ejemplo, MOVER BLOQUE o VERIFICAR EL DELETREO.
- Las acciones infrecuentes o las acciones complejas podrían requerir ir a través de una sucesión de selecciones de menú o llenado de forma por ejemplo, para cambiar el formato de impresión o para revisar los parámetros de protocolo de red.

Una matriz de usuarios y tareas nos puede ayudar para ordenar estos puntos (Ver la tabla 3-1). En cada campo, el diseñador puede poner una marca de verificación para indicar que este usuario efectúa esta tarea. Un análisis más preciso conduciría inclusión de frecuencias, en vez de marcas de verificación simples.

Frecuencia de Tareas por Titulo de Trabajo

Titulo de Trabajo	Consulta por Paciente	Actualización de datos	Consulta atraves de pacientes	Agregar Relaciones	Evaluar el Sistema
Enfermeras	0.14	0.11			
Medicos	0.06	0.04			
Supervisores	0.01	0.01	0.04		
Personal Designado	0.26				
Actualizadores del registro Medico	0.07	0.04	0.04	0.01	
Investigadores Clinicos			0.08		
Programadores de B.D.			0.02	0.02	0.05

Tabla 3-1.-Frecuencia hipotética de uso de datos para un sistema de información medica clínica. Consultas por paciente de personal designado son las tareas mas frecuentes

ESTILOS DE INTERACCION O CONTROL DE DIALOGO

Cuando el análisis de tarea sea completado y la semántica de la tarea objeto y las acciones pueden identificarse, **el diseñador puede escoger entre estos estilos de interacción primarios: selección de menú, llenado de forma, lenguaje de comandos, lenguaje natural y manipulación directa.**

Estilos de interaccón	
Ventajas	*Desventajas*
selección de menú	
rápido aprendizaje reduce teclazos hace estructuras de decisión permite el uso de herramientas de administración de diálogos permite soporte fácil de tratamiento de errores	impone peligro de muchos menúes puede demorar a usuarios frecuentes consume espacio de pantalla requiere porcentajes rápidos de presentación
llenado de forma	
simplifica la entrada de datos requiere entrenamiento modesto hace asistencia conveniente permite el uso de herramientas de administración de formas	consume espacio de pantalla
lenguaje de comandos	
es flexible es apelado poderoso por los usuarios soporta iniciativa del usuario es conveniente para crear macros definidas por el usuario	el tratamiento de errores es pobre requiere de sustancial entrenamiento y memorización
lenguaje natural	
libera de la carga de aprender sintaxis	requiere un dialogo claro puede requerir mas teclazos no puede mostrar el contexto es impredecible
manipulación directa	
presenta conceptos de tarea visualmente es fácil de aprender es fácil de retener permite evitar errores fomenta la exploraciónpermite alta satisfacción subjetiva	puede ser dificil para programar puede requerir dispositivos de despliegue gráfico y dispositivos apuntadores

Tabla 3-2.-Las ventajas y desventajas de los cinco estilos de interacción primarios

Aquí, nosotros damos una descripción, resumen comparativo entre el conjunto de las etapas o de estilos.

- **Selección de menú:** En los sistemas de selección de menú, los usuarios leen un listado de elementos, seleccionan uno, el mas apropiado a su tarea, aplican la sintaxis para indicar su selección, confirman la elección, inician la acción, y observan el efecto. Si la terminología y el significado de los elementos son distintos y comprensibles, entonces los usuarios pueden realizar su tarea con poco aprendizaje o memorización y pocos golpes de tecla. El beneficio más grande puede ser que hay una estructura clara al hacer una decisión, desde solo unas pocas elecciones se ponen de manifiesto a la vez. Este estilo de interacción es apropiado para usuarios novatos e intermitentes y puede ser apelado por los usuarios frecuentes si los mecanismos de selección y la visualización son rápidos.

Para diseñadores, de sistemas de selección de menú requieren de un cuida-

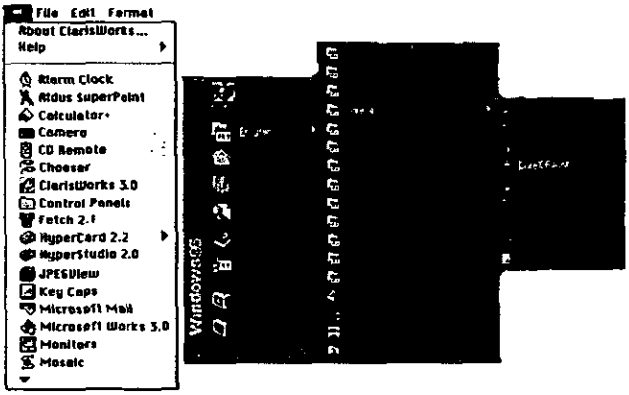


Figura 3-38 Ejemplo de Interacción por Menus, a la Izquierda la de la GUI Mac y a la Derecha de la GUI Windows95

doso análisis de tarea para asegurar que todas las funciones se soportan convenientemente y que la terminología se eligió cuidadosamente y utilizo coherentemente.

- **Llenado de forma:** Cuando se requiere la introducción de datos, la selección de menú comúnmente llega a ser engorrosa, y el llenado de forma (también llamado rellenado de blancos) es apropiado. Los usuarios tienen una visualización de campos relacionados, trasladan el cursor entre los campos, y se introducen datos donde se desea. Con el estilo de interacción de llenado de forma, los usuarios deben comprender las etiquetas de los campos, saber los valores permisibles y el método de introducción de datos, y ser capaces de responder a mensajes de error. Desde el conocimiento del teclado, las etiquetas, y los campos permisibles requeridos, algún entrenamiento puede ser necesarios. Este estilo de interacción es muy apropiado para usuarios intermitentes inteligentes o los usuarios frecuentes.(Ver la Figura 3-5 para un ejemplo de llenado de forma)

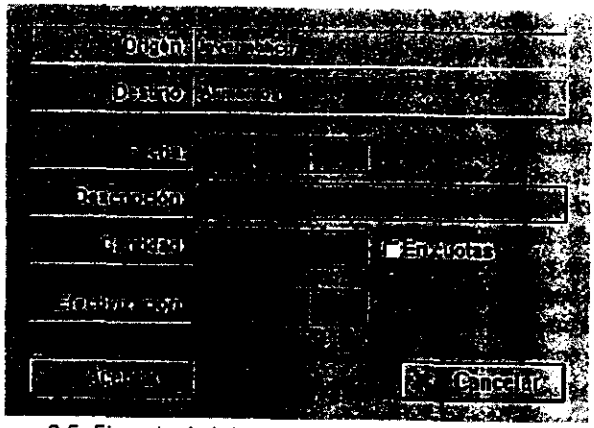


Figura 3-5 Ejemplo de Interacción por Llenado de Forma

- **Lenguaje de comandos:** Es para usuarios frecuentes, los lenguajes de comandos proveen un sentimiento fuerte de control e iniciativa. Los usuarios aprenden la sintaxis y pueden obtener frecuentemente posibles expresiones complejas rápidamente, sin tener que leer indicadores (prompts) distrayentes. Sin embargo, el porcentaje de error es típicamente alto, el entrenamiento es necesario, y la que retención puede ser pobre. Los mensajes de error y la asistencia en línea son difíciles de proveer a causa de la diversidad de posibilidades aunado a la complejidad de representación desde tareas hasta conceptos de computadora y sintaxis. *Los lenguajes de comandos y los lenguajes de programación o consulta son el dominio del usuario experto*

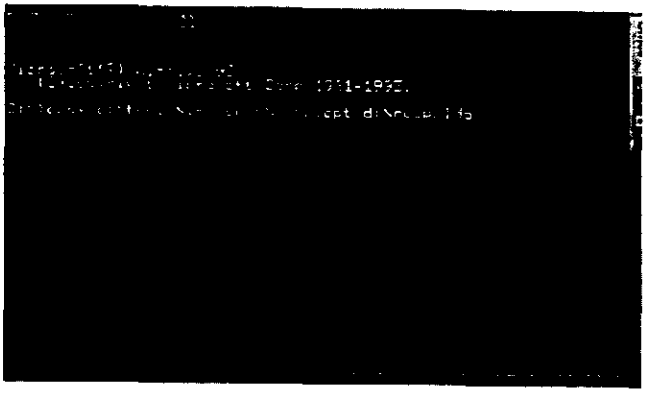


Figura 3-6 Ejemplo Interacción por Lenguaje de Comandos

frecuente, quienes frecuentemente derivan gran satisfacción por dominar un conjunto complejo de semántica y sintaxis.

- **Lenguaje natural:** Es la esperanza de que las computadoras respondan adecuadamente al arbitrario lenguaje natural, sentencias o frases comprometen a muchos desarrolladores de sistemas e investigadores, a pesar del éxito limitado aun esta lejos de que sea totalmente factible. El lenguaje natural de interacción comúnmente provee poco contexto para emitir el siguiente comando, frecuentemente requiere de la aclaración del diálogo, y puede ser más lenta y más engorrosa que las demás alternativas. Todavía, en donde el usuario conoce bien sobre el dominio de tarea, cuyo ámbito es limitado y donde el uso intermitente inhibe el entrenamiento del lenguaje de comandos, existen oportunidades para las interfaces de lenguaje natural
- **Manipulación Directa:** Cuando un diseñador inteligente puede crear una representación Visual del mundo de acción, las tareas de los usuarios pueden simplificarse mucho, porque la manipulación directa de los objetos de interés es posible. Ejemplos de tales sistemas incluyen los visualizadores de editores, el LOTUS 1-2-3, los sistemas de control de trafico aéreo, y los juegos de vídeo. Por indicar en representaciones visuales de objetos y acciones, los usuarios pueden efectuar las tareas rápidamente y observar los resultados inmediatamente. La entrada de comandos por teclado o las selecciones de menú son reemplazadas por dispositivos de movimiento de cursor para seleccionar desde un conjunto visible de objetos y acciones. La manipulación directa es recurrida por los novatos, es fácil de recordar para usuarios intermitentes, y con diseño cuidadoso, puede ser rápido para usuarios

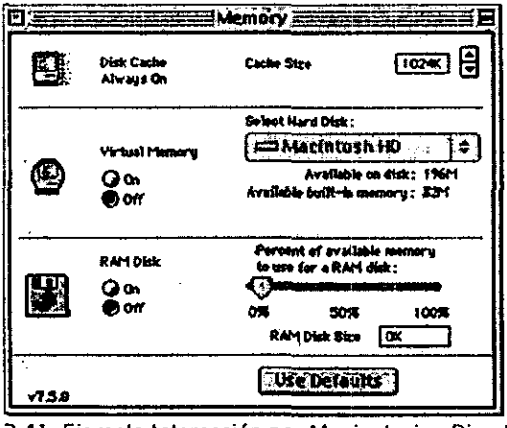


Figura 3-41 Ejemplo Interacción por Manipulacion Directa

frecuentes. En una sección mas adelante se describe manipulación directa y sus aplicaciones con mas profundidad.

Mezclar varios estilos de interacción puede ser apropiado cuando lo requieran las tareas y los usuarios sean diversos. Los comandos pueden conducir al usuario a llenar forma donde la introducción de datos se requiere o los menúes puedan utilizarse para controlar un ambiente de manipulación directa cuando una visualización apropiada de acciones no puede encontrarse.

MANIPULACION DIRECTA

El truco en crear un sistema de manipulación directa es producir una apropiada representación o modelo de la realidad. Algunos diseñadores pueden encontrar en esto dificultad para pensar acerca de los problemas de información en una forma visual, de cualquier modo, con practica, ellos pueden encontrar más natural esta forma de pensar. Si bien detalles acerca de temas como interfaces y tareas no fueron reportados, los resultados mostrados aumentaron la productividad y reducieron la fatiga para usuarios experimentados

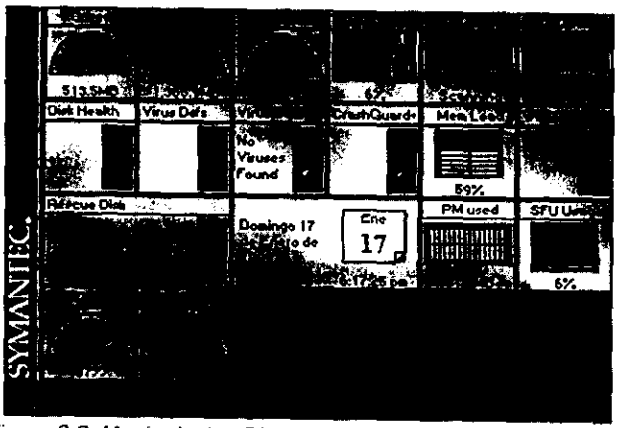


Figura 3-8 Manipulacion Directa

con interfaces de usuario gráficas, comparadas con las interfaces de usuario basadas en caracteres

Muchos autores han intentado describir los componentes principales de la manipulación directa. Un imaginativo observador y diseñador de sistemas interactivos, Ted Nelson (1980), percibe entusiasmo del usuario cuando la interface es construida por lo que el llama *el principio de la virtualidad (una representación de la realidad que puede ser manipulada)*. Rutkowski (1982) da a entender un concepto similar en su principio de *transparencia (El usuario es capaz de aplicar su intelecto directamente hacia la tarea; la herramienta misma parece desaparecer)*. Heckel (1991) lamenta que *"Nuestros instintos y educación como ingenieros nos fomenta a pensar lógicamente en vez de visualmente, y eso es*

contraproducente para el diseño amigable". El sugiere que pensar como un cineasta puede ayudar para el diseño de sistemas interactivos.

La atracción de la manipulación directa es evidente en el entusiasmo de los usuarios. Los diseñadores de ejemplos como procesadores de palabras, hojas de calculo, administración de datos espaciales, vídeo juegos, diseño y manufactura asistida por computadora tienen una innovadora inspiración y una compresión intuitiva de que es lo que los usuarios desean. Cada ejemplo tiene características que pueden ser criticadas, pero veamos dichas características más productivas para la construcción de un retrato integrado de la manipulación directa.

1.- Representación continua de los objetos y acciones de interés

2.- Acciones fisicas o presiones de botones etiquetados en vez de sintaxis complejas.

3.- Operaciones reversibles de forma incremental y rápida cuyo efecto en el objeto de interés es inmediatamente visible.

Usando estos tres principios, es posible diseñar sistemas que tengan estos atributos benéficos:

- Los novatos pueden aprender la funcionalidad básica rápidamente, usualmente a través de demostraciones por un usuario más experimentado.
- Los expertos pueden trabajar rápidamente para realizar un amplio rango de tareas, aun definiendo nuevas funciones y características.
- Los usuarios informados-intermitentes pueden retener conceptos operacionales.
- Los mensajes de error raramente son necesitados.
- Los usuarios pueden inmediatamente ver si sus acciones son lejanas a sus metas, y si las acciones son contraproducentes, ellos pueden simplemente cambiar la dirección de sus actividades.
- Los usuarios experimentan menor ansiedad por que el sistema es comprensible y por que las acciones pueden ser revetirdas muy fácilmente .
- Los usuarios ganan confianza y logran dominio por que ellos son los iniciadores de la acción, se sienten en control, y las respuestas del sistema son predecibles.

El éxito de la manipulación directa es entendible en el contexto del modelo SSOA; El objeto de interés es desplegado así, que las acciones son directamente en el dominio de tarea de alto nivel. Hay poca necesidad de descomposición mental de tareas dentro de múltiples comandos con una forma sintáctica compleja. Por el contrario, cada acción produce un resultado comprensible en el dominio de tarea que es visible inmediatamente. La cercanía de la tarea hacia la acción sintáctica reduce al operador solucionar problemas, carga y nerviosismo.

Dominar las semánticas de tarea de los usuarios, y la distracción de tratar con las semánticas de computadora y la sintaxis es reducida (Ver Figura 3-9). Los sicólogos tienen

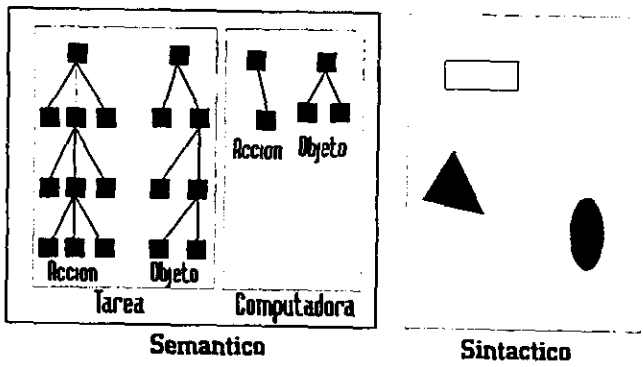


Figura 3-9 Los usuarios de los sistemas de manipulación directa, pueden necesitar tener conocimiento substancial del dominio de tarea. Sin embargo, los usuarios deberan adquirir solo una modesta cantidad de conocimiento semantico y sintactico relativo a la computadora

gran conocimiento que las relaciones espaciales y las acciones son comprendidas mas rápidamente de forma visual mas que con representaciones lingüísticas. Los limites de la manipulación directa serán determinados por la imaginación y las técnicas de diseño.

PENSAMIENTO VISUAL E ICONOS

Los conceptos de *lenguaje visual* y de *pensamiento visual* fueron promovidos por Arnheim (1972) y fueron aprovechados por los diseñadores comerciales de gráficas, académicos orientados en semiótica (semiótica es el estudio de los signos y *símbolos*) y por los *gurus* (*maestros*) de la visualización de datos. Las computadoras proveen un notable ambiente visual para revelar estructuras, mostrando relaciones, y habilitando interactivamente a los usuarios atraídos con diseños artísticos.

La cada vez más, naturaleza visual de las interfaces de computadora puede algunas veces desafiar o igual amenazar la lógica lineal, lo orientado a texto, lo obligatorio, a los programadores racionales quienes fueron el corazón de la primera generación de hackers. Aunque estos estereotipos no hacen frente al análisis científico, ellos dan entender dos caminos que la computación esta siguiendo. Las nuevas direcciones visuales son algunas veces despreciadas por los tradicionalistas como las interfaces WIMP por sus siglas en ingles (Windows, Icons, Mouse y Pull-down menúes) (ventanas, iconos, ratón y menús desplegables), mientras que los devotos de la línea de comandos se han visto tan inflexibles, o igual de testarudos.

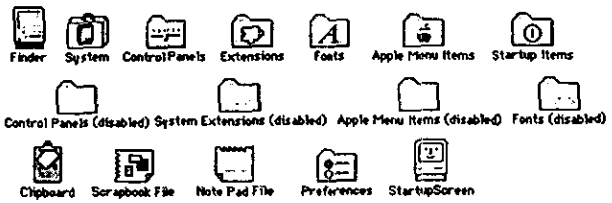


Figura 3-44 Ejemplo de Iconos

No es de sorprenderse que la definición del diccionario de icono usualmente se refiere a imágenes religiosas, pero la noción central es que un icono es una imagen, una pintura, un símbolo representando un concepto (Gittins,1986; Rogers,1989). En el mundo de la computación, los iconos son usualmente pequeños aproximadamente de 64 x 64 pixeles y

Figura 3-45 Iconos usados frecuentemente en programas de dibujo

son representaciones de un archivo o programa (un objeto o una acción ver figura 3-10, los iconos pequeños son a menudo usados para salvar espacio o para ser integrados en otros en otros objetos, tal como en el borde de una ventana. Tampoco es de sorprenderse que los iconos frecuentemente son usados en los programas de pintar, para representar las herramientas o acciones (laso o tijeras para cortar una imagen, una brocha para colorear, un lápiz para dibujar, una goma para destruir o borrar) (Ver Figura 3-11)

Para situaciones donde ambos, tanto un icono visual o un elemento de texto son posibles: por ejemplo, en el listado de un directorio, los diseñadores tienen que entretejer dos puntos: ¿cómo decido entre iconos y texto? y ¿cómo diseñar los iconos?. La bien establecida vía de los signos es una útil fuente de experiencia. Los iconos son inmejorables para mostrar cosas tales como una curva en un camino, pero a veces una frase como solo una via no entre es más comprensible que un icono. Por supuesto, esta el enfoque de tener un poco de cada uno (como por ejemplo, el signo octagonal de STOP) y hay evidencia que los iconos mas las palabras son efectivos en situaciones de computación. Así la respuesta a la primer pregunta (decidir entre iconos y texto) depende no solamente de los usuarios y

de las tareas, sino también de la calidad de los iconos o de las palabras que son propuestas. En adición, estas directivas de especificación de iconos serán consideradas:

- Representar el objeto o acción de una manera reconocible y familiar
- Limitar el numero de diferentes iconos
- Hacer que el icono destaque del fondo
- Considerar iconos tridimensionales, ellos pueden captar al ojo, pero también puede ser distrayentes
- Asegurar que un solo icono se seleccionó y es claramente visible cuando esta rodeado por iconos desselecionados
- Hacer cada icono distintivo de otros iconos.
- Asegurar la armonía de cada icono, como un miembro de una familia de iconos
- Diseñar el movimiento de animación: cuando se arrastra un icono, el usuario podría mover un icono hueco, solo un marco, posiblemente una versión de intensidad de grises, una caja negra.

Agregar detalles de información, tales como sombreado para mostrar el tamaño de un archivo (una gran sombra indicara una archivo grande), espesor para mostrar lo grande del directorio (mas grosor significara mas archivos dentro), color para mostrar la edad de un documento (mas viejo significara más amarillo o más gris), o animación para mostrar cuanto de un archivo ha sido impreso (una carpeta de documento es absorbida progresivamente dentro del icono de la impresora)

Explorar el uso de combinaciones de iconos para crear nuevos objetos o acciones por ejemplo, arrastrar el icono de un documento al icono de una carpeta o un bote de basura, una salida de correo, o un icono de impresora puede ser de gran utilidad. ¿Puede un icono ser agregado a otro documento por pegar dos iconos de documentos adyacentes? Puede los niveles de seguridad ser puestos por arrastrar un documento o un folder hacia un perro guardián, un carro de policía, o un icono de bóveda? Pueden dos bases de datos representadas por iconos ser intersectadas por sobreponer sus iconos?

DISEÑO DE ICONOS

Marcus en 1992 aplico la semiótica como una guía para cuatro niveles de diseño de iconos:

1.- Cualidades Léxicas: Generadas por la maquina forma de las marcas o pixeles, color, brillo, flasheo.

2.- Sintácticas: Apariencia y movimiento de las líneas, patrones, partes modulares, tamaño, forma.

3.- Semánticas: Objetos representados: concretos, abstractos, todo-o parte.

4.- Pragmáticas: agradable, utilidad, identificable, recordable, en general legible.

El recomienda comenzar por crear rápidos bosquejos, promoviendo un estilo consistente, diseñando el trazado sobre una cuadricula, simplificando el aspecto y evaluando los diseños por pruebas con los usuarios. Nosotros podríamos considerar un quinto nivel de diseño de iconos:

5.-Dinamicas: La receptividad hacia los clicks (presionar y soltar el botón del ratón), iluminado, arrastrado y combinaciones.

Las dinámicas de los iconos pueden también incluir un rico conjunto de ademanes con un ratón, pantallas de toque o pluma. Los ademanes podrían indicar por ejemplo: copiar (subir y bajar), borrar (cruzarlo), editar (hacer un circulo), etc. Los iconos también podrían ser asociados a sonidos.

PROBLEMAS CON LA MANIPULACION DIRECTA.

Los diseños con manipulación directa pueden consumir valioso espacio en la pantalla y así forzar valores fuera de la pantalla, requiriendo desplazamiento (scrolling) o múltiples acciones.

- Un segundo problema es que los usuarios deberán aprender el significado de los componentes de la representación visual. Un icono gráfico puede ser significativo para el diseñador, pero puede requerir mucho mas tiempo de aprendizaje que una palabra.
- Un tercer problema es que la representación visual puede ser engañosa; los usuarios pueden captar la representación analógica rápidamente, pero entonces llegar a conclusiones incorrectas acerca de las acciones permisibles.
- Un cuarto problema es que, los mecanógrafos experimentados, moviendo un ratón o levantando un dedo hasta el punto deseado puede algunas veces ser mas lento que teclear.

EL DISEÑO DE LA PROGRAMACIÓN DE INTERFACES HUMANO-COMPUTADORA

El diseño de la programación de interfaces humano-computadora debe determinar, primero, el o los métodos de control de diálogo que serán empleados y,

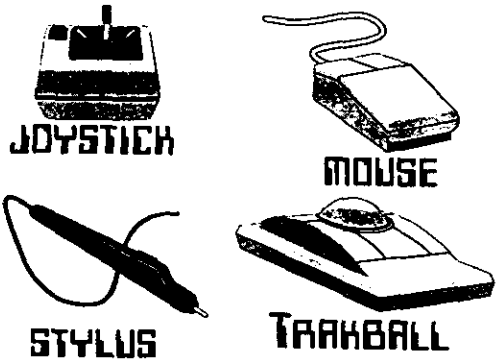


Figura 3-12 Dispositivos Apuntadores

segundo, establecer medios concretos para el o los métodos de control de diálogo seleccionados, la entrada de datos, la salida de datos y la retroalimentación al usuario.

METODOS DE CONTROL DE DIALOGO

Los métodos de control de diálogo entre el humano y la computadora están íntimamente ligados a la tecnología disponible en el equipo. *Así, las primeras interfaces humano-computadora, implementadas en teletipos, se basaban en lenguajes de comandos; posteriormente se emplearon las terminales alfanuméricas de rayos catódicos con lo que se dio paso a implementar los diálogos entre el usuario y la computadora a través de menús de opciones y a través de teclas de función; subsecuentemente, con la disponibilidad de terminales gráficas y de dispositivos apuntadores como el mouse, el track ball, stylus y el joystick* (Ver figura 3-12) surgió el método de manipulación directa actualmente disponemos de equipo que permite al usuario la entrada manuscrita de datos y comandos y ya se encuentran en el mercado algunas interfaces que reconocen la voz humana para recibir comandos. En un futuro no lejano, con los resultados de los trabajos llevados a cabo en el campo de la realidad virtual, podemos esperar que la interacción humano-computadora se efectuará a través del lenguaje natural (castellano, inglés, etc.) y a través del reconocimiento de los movimientos e inclusive del foco de atención indicado por la vista del usuario. Esta evolución tecnológica y el desarrollo asociado de los métodos del control del diálogo entre el humano y la computadora no implica que las nuevas tecnologías hagan obsoletos a los métodos empleados anteriormente. Los diversos métodos siguen siendo vigentes y son más o menos apropiados en función de las características del usuario y de la aplicación. La tabla 3-3 representa que método de control de diálogo es apropiado dependiendo de las características del usuario y de la aplicación:

	Usuario				Interacción
	Experencia		Empleo		
	Experto	Inexperto	Continuo	Esporadico	
Comandos	X		X		Rapida
Teclas de Función	X		X		Muy Rapida
Menús	X	X		X	Muy Lenta
Manipulación Directa	X	X	X	X	Lenta

Tabla 3-3.- La aplicabilidad de algunos métodos de control de diálogo

Desde el punto de vista del usuario debemos considerar dos aspectos: su experiencia y la frecuencia del uso del sistema. La **experiencia** se refiere, por una parte, a la costumbre que posee el usuario para utilizar la computadora como una herramienta y, por otra parte, al conocimiento del problema que la herramienta pretende resolver. No se puede tratar igual a un usuario que nunca ha empleado una computadora y que puede inclusive tener temor o aversión a ella, que a un usuario que tiene años de experiencia empleándola en su trabajo cotidiano. Tampoco se puede tratar igual a un usuario experto, por ejemplo, en tipografía y a uno que no lo es y que emplea un procesador de textos para producir cierto documento. La **frecuencia** de uso se refiere a la periodicidad con la que el usuario interactúa con un sistema determinado. No se puede tratar igual al usuario que cotidianamente emplea cierto sistema que al usuario que esporádicamente hace uso de él.

Desde el punto de vista de la aplicación, debemos considerar la velocidad requerida en los diálogos para llevar a cabo apropiadamente las tareas requeridas. No es lo mismo un sistema a través del cual un operador controla apertura y cierre de válvulas de seguridad en un proceso industrial que un sistema de dibujo por computadora.

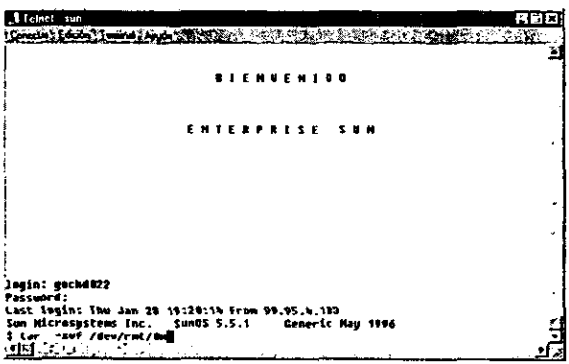


Figura 3-47 Observar la complejidad del Comando Tar en el sistema operativo Unix

En la tabla 3-3 vemos que : Los lenguajes de comandos son apropiados para que usuarios expertos y cotidianos interactúen con la computadora. Estos lenguajes, por una parte, imponen una jerga particular, y por otro, requieren del usuario una memoria excepcional para ejemplificar, consideremos el comando tar –xopvf /dev/rmt/3m, (Ver Figura 3-13) que sirve para manejar un dispositivo de cinta magnética (Tape Archive) bajo el sistema operativo Unix; primero hace falta saber que es Tape Archive, y después hace falta recordar que tar es el comando que se refiere a eso sin comentar sobre las opciones –xopvf... Por otra parte los lenguajes de comandos permiten una interacción rápida entre el usuario y la computadora. El usuario tiene la posibilidad de solicitar directamente lo que desea que se lleva a cabo.

El concepto de la tecla de función es más amplio que el de las teclas etiquetadas *F1. F12* que se encuentran en la mayoría de los teclados estandar actuales. Por una parte, podemos asociar una acción a cualquier tecla del teclado por otra parte podemos construir teclados dedicados, en los que pueden existir teclas etiquetadas con el nombre de una función particular. Sin embargo, las teclas de función suelen ser más crípticas que los comandos. Si el lector ha trabajado con el editor de textos vi (en unix), sabe a lo que nos referimos. Por otra parte, para un usuario cotidiano y un usuario con experiencia, las teclas de función suelen ser un mecanismo más rápido de interacción con la computadora. Los menús de opciones son un medio muy difundido y muy aceptado actualmente para controlar la interfaz humano-computadora. La aplicación de los menús de opciones se basa en el principio cognoscitivo ***de que es más fácil reconocer que recordar.*** Por esto los menús son prácticos para usuarios esporádicos, independientemente de su experiencia en la computadora y en la aplicación.

Cabe señalar que, como se muestra en la tabla 3-3, los menús, aunque muy populares, pueden ser el mecanismo más lento para conducir la interacción entre el usuario y la computadora. Esto se debe naturalmente a la necesidad que se le impone al usuario de *"navegar"* a través de los menús hasta llegar a la opción que desea llevar a cabo. Esta limitante puede ser solventada proporcionando atajos (shortcuts), por ejemplo mediante la combinación de comandos o teclas de función con los menús. De esta forma el usuario puede *"saltar"* hasta el menú que desea sin tener que navegar hasta él. En estas circunstancias, los menús se vuelven apropiados también para los usuarios cotidianos.

La manipulación directa que se refiere a la posibilidad de seleccionar algo (frecuentemente un icono que representa un objeto o colección de objetos en el dominio de la aplicación) y después decir que se desea hacer con él, proporciona una metáfora de interacción humano-computadora que puede hacer la comunicación más intuitiva, a esto se debe la popularidad de las interfaces gráficas como **Macintosh, MS/Windows, X-Window**, etc. que se hacen apropiadas para usuarios expertos o no expertos, cotidianos o esporádicos.

REGLAS DE ORO EN EL DISEÑO

Una vez seleccionado el método o la combinación de métodos de control de diálogo en una interfaz humano-computadora, se deben establecer medios concretos para la implementación de dicho control de diálogo (por ejemplo diseño del lenguaje de comandos, diseño de iconos, etc.) La entrada de datos (por ejemplo diseño de formas), la salida de datos (por ejemplo diseño de despliegues gráficos, diseño de reportes etc.) y la retroalimentación al usuario (por ejemplo diseño mensajes de aviso). Esta sección presenta principios fundamentales de diseño que son aplicables en la mayoría de los sistemas interactivos. Estos principios fundamentales del diseño de interfaz, han sido derivados heurísticamente de la experiencia, deberán validarse, refinarse y extenderse para cada ambiente:

Las investigaciones llevadas a cabo en aspectos de Ingeniería de Factores Humanos en el empleo de software concluyen con un cierto número de recomendaciones que es conveniente considerar par lograr que el diseño de interfaces humano-computadora sea agradable al usuario.

A continuación presentamos una síntesis de trece "reglas de oro", derivadas del trabajo reportado por diversos investigadores.

1.- **Dar al usuario el control:** Esta es la primera y más importante regla que el diseñador de una interfaz humano-computadora debe considerar y aplicar continuamente. No hay nada más frustrante para un usuario que sentirse impuesto por la computadora, obligado a hacer algo, posiblemente sin saber cual será el resultado e imposibilitado para decidir sobre el curso de las acciones. Por supuesto que un diálogo entre el usuario y la computadora puede requerir secuencia y estructura. Lo importante es que el usuario siempre pueda saber en que punto de esa secuencia o estructura se encuentra y pueda regresar o salir de ella si así lo desea

2.- **Adoptar la perspectiva del usuario:** Esta regla se refiere a la necesidad de comunicarse con el usuario en los términos que él entiende, y no imponerle la jerga computacional familiar para él que desarrolla el software. Por ejemplo, es más claro para un arquitecto que dibuja planos por computadora y que desea almacenar el plano con el que está trabajando, recibir el mensaje de la máquina que le informa y le pregunta algo como "ya existe un plano con la clave catastral 1100-10-120-025; ¿desea sustituirlo? [si/no]:" que recibir algo como "el archivo 120.025.PLP ya existe; ¿desea sustituirlo? [si/no]:". Y ni mencionar que si lo que recibe el arquitecto es algo como "file already exists; override? y/n:"

3.- **Dirigir la interfaz al nivel del usuario:** Esta regla se refiere a que el diseñador de la interfaz debe considerar el nivel de experiencia del usuario en cuanto al uso de la computadora, en cuanto al uso del sistema y en cuanto a su conoci-

miento de la aplicación y está entonces relacionada con el método de control de diálogo. Sin embargo, la aplicación de esta regla es más amplia.

Por ejemplo, el diseñador de la interfaz puede considerar dos o tres niveles de usuario, y proporciona mecanismos de ayuda en consecuencia. Para un usuario experto pueden ser suficientes mensajes que especifiquen la sintaxis de los comandos; un usuario novato puede requerir una explicación más amplia y la inclusión de ejemplos.

4.- **Ser consistentes en la interfaz:** La consistencia es muy importante en la interfaz humano-computadora ya que, por una parte, promueve el fácil aprendizaje del manejo del sistema y la extensión de su aplicación y, por otra coadyuva haciendo sentir al usuario que él tiene el control (ver regla 1). Este principio es uno de los más frecuentemente infringidos, y aún así de uno los más fácil para reparar y evitar. Las sucesiones uniformes de acciones deberían requerirse en situaciones similares; terminología idéntica debería utilizarse en indicadores (prompts), menús, y pantallas de ayuda; y los comandos uniformes deberían ser empleados a lo largo de lo anterior. Las excepciones, tales como la no repetición de contraseñas o la confirmación del comando BORRAR, deberían ser comprensibles y limitadas en número.

Uno de los mecanismos de aprendizaje más efectivos es el aprendizaje por analogía; si sabemos algo, y lo nuevo que estamos aprendiendo se asemeja a ese algo que ya conocemos, automáticamente registramos el nuevo conocimiento.

Si el usuario de un sistema computarizado puede intuir cuál será el comportamiento de éste en una situación nueva pero análoga a otra que le es familiar, y si el sistema efectivamente se comporta análogamente adquirirá confianza para trabajar con el sistema e inclusive estará dispuesto a "experimentar" y buscará ampliar la aplicación de dicho sistema.

5.- **Proteger al usuario de trabajar con el hardware y con el *software*:** Esta regla puede ser vista como una extensión o una complementación de la regla número 4 e implica que el diseñador de una interfaz humano-computadora debe estar consciente de que el sistema con el que va a comunicarse el usuario puede tener interfaces con otros programas y con el equipo. Esto debe ser transparente para el usuario, quien debe obtener la imagen de esta trabajando con un ente único y controlado.

6.- **Minimizar los requerimientos de memorización del usuario:** Anteriormente comentamos que los menús de opciones son un medio de control de diálogo muy difundido y aceptado para controlar los diálogos en la interfaz humano-computadora, debido a que permiten aplicar el principio cognoscitivo de que es más fácil reconocer que recordar.

La capacidad de memoria del ser humano no ha podido ser cuantificada, pero sabemos que rebasa por mucho la de la computadora más poderosa hasta ahora construida. Sin embargo, la memoria humana se divide en memoria de largo plazo y memoria de corto plazo. La memoria de largo plazo almacena todos los

conocimientos que posee un ser humano y es la que tiene una capacidad muy grande; la memoria de corto plazo almacena los datos necesarios para la acción intelectual inmediata, y sólo es capaz de almacenar *"siete más o menos dos pedazos"* de información.

Es necesario que el diseñador de una interfaz humano-computadora tome en consideración las restricciones de memoria de corto plazo del ser humano, de esta forma el usuario tendrá la experiencia de una interacción más agradable.

7.- **Seguir principio de buen diseño gráfico:** La estética es un elemento importante para crear un ambiente de facilidad de empleo en una interfaz humano-computadora.

Los principios de diseño gráfico no se aplican únicamente a las interfaces gráficas. Por ejemplo, se debe considerar el balance alrededor de los ejes centrales en reportes y en forma de captura, así como la alineación del punto decimal es importante en las tablas numéricas. En las terminales modernas a color y en los desplegado gráficos, el diseñador debe considerar además el empleo y la mezcla de colores y formas para crear un ambiente apropiado. El color y la forma son información. En secciones, mas adelante se vera con mayor detalle los puntos de teoría del color y el empleo de las fuentes.

8.- **Ofrecer retroalimentación informativa al usuario:** La investigación en Factores Humanos en computación ha demostrado el incremento en el grado de ansiedad de los usuarios de un sistema cuando no saben lo que el sistema está haciendo y cuando los tiempos de respuesta en la misma acción o en acciones semejantes –desde la perspectiva del usuario- cambian dramáticamente de un caso a otro.

Es conveniente que un sistema siempre retroalimente al usuario cuando ha recibido una orden; de la misma manera, es conveniente que le informe lo que está haciendo (sin olvidar la recomendación propuesta en la regla 2), y que en la medida de lo posible, cuando el tiempo de respuesta va a exceder unos cuantos segundos, se le informe ya sea el tiempo aproximado que dilatará o progresivamente el porcentaje de avance logrado.

Para acciones frecuentes y menores, la respuesta puede ser modesta, mientras para las acciones infrecuentes y mayores, la respuesta debería ser más substancial. La presentación visual de los objetos de interés, provee un ambiente conveniente para mostrar cambios explícitos.

9.- **Activar para los usuarios frecuentes, el utilizar atajos (shotcuts):** Como la frecuencia de uso aumenta, también aumenta los deseos del usuario de reducir el número de interacciones y aumentar el ritmo *"rendimiento"* de dicha interacción. Las abreviaturas como: teclas especiales, comandos ocultos, y las instalaciones de macro son apreciadas por los usuarios inteligentes-frecuentes. Los tiempos de respuesta más corta y los porcentajes más rápidos de visualización son otras atracciones para los usuarios frecuentes.

10.- **Diseñar diálogos para producir cierres:** Las secuencias de acciones deberán organizarse en grupos con un principio, un medio, y un fin. La retroalimentación informativa en la terminación de un grupo de acciones dará a los operadores la satisfacción de realización, una sensación de alivio, la señal para soltar opciones y planes de eventualidad desde sus mentes, y una indicación que el camino esta limpio para preparar el siguiente grupo de acciones.

11.- **Ofrecer tratamiento de errores simple:** Tanto como sea posible, diseñar el sistema de tal manera que el usuario no pueda cometer un error serio. Si un error es hecho, el sistema debería detectar el error y ofrecer mecanismos simples y comprensibles para manejar el error. El usuario no debería tener que volver a escribir el comando entero, sino que debería necesitar reparar solo la parte defectuosa. Los comandos erróneos deberían dejar el estado del sistema sin cambiar, o el sistema debería dar instrucciones sobre como restaurar el estado.

12.- **Permitir la fácil reversión de acciones:** Tanto como sea posible, las acciones deberían ser reversibles. Esta característica releva la ansiedad, de que el usuario sabe que los errores pueden deshacerse; así fomenta la exploración de opciones no familiarizadas. Las unidades de reversibilidad pueden ser una acción única, una introducción de datos, o un grupo completo de acciones.

13.- **Proporcionar documentación en línea:** La documentación en línea se refiere a que el usuario obtenga a través de la interfaz humano-computadora todo lo que necesita para operar el sistema y para recuperarse de errores

PREVENCIÓN DE ERRORES

No hay medicina contra la muerte, y contra el error ninguna regla se ha encontrado, Sigmund Freud, (Inscripción que él escribió sobre su retrato). Los Usuarios de procesadores de textos, hojas de cálculo, base de datos, sistemas de control de trafico aéreo, y otros sistemas interactivos tienen equivocaciones mucho más frecuentemente de lo que se podría esperar. **Una de dirección para reducir la pérdida en la productividad debido a errores está mejorar los mensajes de error provistos por el sistema de computadora.** Lamentablemente la practica común de muchos programadores de *temperamento flemático,* que es conocida como ***retorno silencioso***. Es decir, ante la incapacidad de llevar acabo el servicio la función opta por no hacer nada y retornar. Esto no es mas que poner una bomba de tiempo en el programa. La función retorna dejando al cliente en un estado ilegal quien continuara su trabajo tranquilamente hasta que más tarde o temprano la bomba explote (quiera la suerte que sea más temprano que tarde para que el daño sea menor). Otra filosofía adoptada es la ***histérica***, tal vez practicada por programadores más *sanguíneos o coléricos*, es la de poner un mensaje de error y terminar. La debilidad de las prácticas anteriores se debe a que, quien detecte el problema no necesariamente puede resolverlo ni tiene porque decidir como tratarlo. **Una solución puede ser combinar los temperamentos sanguíneo y flemático: Alarma pero organizadamente.**

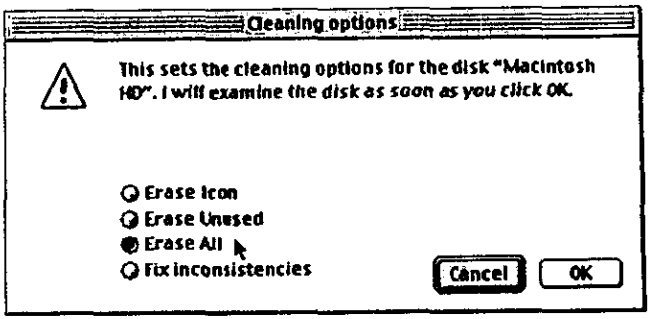


Figura 3.48 Ejemplo de un mensaje de Advertencia

Shneiderman en 1982 informo sobre cinco experimentaciones, en que los cambios a los mensajes de error condujeron al éxito mejorando para reparar los errores, reducir el porcentaje de error, y aumentar la satisfacción subjetiva. Los mensajes superiores de error eran más específicos, positivos en el tono, y constructivos (diciendo al usuario qué hacer, más que meramente reportarle el problema). Más que usar mensajes vagos y hostiles, tal como **ERROR DE SINTAXIS o DATOS PROHIBIDOS**, los diseñadores fueron animados a utilizar mensajes informativos, tal como **FALTA PARENTESIS IZQUIERDO o LAS OPCIONES DEL MENU ESTAN EN LA RANGO DE 1 A 6.** Los mensajes mejorados de error, sin embargo, no son la única medicina útil. **Un enfoque más efectivo está en impedir que los errores ocurran.** Esta meta es más realizable que puede ser utilizada en muchos sistemas.

El primer paso está comprender la naturaleza de errores. Una de perspectiva es que la gente comete equivocaciones o *"tropieza"*, que los diseñadores pueden evitar organizando funcionalmente los menúes y pantallas, diseñando las elecciones de menú o comandos para ser distintivas, y hacer difícil a los usuarios que sus acciones sean irrevocables. Se puede ofrecer otras directivas tales como retroalimentación sobre el estado del sistema, y diseño para la consistencia de comandos.

TECNICAS PARA ASEGURAR ACCIONES CORRECTAS.

En un breve resumen se describen tres técnicas específicas para reducir errores por asegurar acciones completas y correctas: Corregir apareamiento de pares, secuencias completas, y comandos correctos.

1.-**Los pares correctos de apareamiento:** Es un problema común por la carencia de pares correctos de apareamiento. Tiene muchas manifestaciones, Un ejem-

plo es el fallo por abrir un paréntesis izquierdo y no cerrar el paréntesis derecho. Si un sistema bibliográfico de búsqueda permite expresiones Booleanas tales como COMPUTADORAS Y (SOCIOLOGIA O PSICOLOGIA) y el usuario fallara sino pone el paréntesis derecho al final, entonces el sistema produciría un mensaje ERROR DE SINTAXIS. Otro error es el fallo para incluir el símbolo marca de cierre (")para cerrar una cadena en EL LENGUAJE BASIC. El comando 10 IMPRIME "HOLA" está en error si la marca derecha falta. Similarmente, un @B o el otro marcador se requiere que indique el fin de negrita, itálica, o resalte del texto en procesadores de textos. Un ejemplo final, si el archivo de texto contiene estas @B Esta es letra Negrita @ entonces las tres palabras entre la marca @B aparecerán en la negrita sobre la impresora. Si la marca derecha @B falta, entonces el resto del archivo se imprimirá en negrita. En cada de uno de estos casos, un par de apareamiento de marcadores es necesario para que la operación este completa y correcta. La omisión de la marca de cierre puede ser prevenida por el uso de un editor, preferentemente orientado a pantalla, el cual ponga el par de componentes comenzar y terminar sobre la pantalla en una sola acción. Por ejemplo, escribiendo un paréntesis izquierdo genere paréntesis derecho y ponga el cursor entre ellos para permitir la creación de los contenidos. Un intento para borrar uno de los paréntesis causa el apareamiento de paréntesis (y posiblemente los contenidos también) sean borrados. Así, el texto nunca puede estar sintácticamente en forma incorrecta. Alguna gente encuentra este enfoque rígido por ser demasiado restrictivo y puede preferir una forma más leve de protección. Cuando el usuario escriba un paréntesis izquierdo, en la esquina inferior izquierda de la pantalla se visualice un mensaje que indique la necesidad de un paréntesis derecho, hasta que ese carácter se escriba.

2.- **Secuencias completas:** Algunas veces, una acción requiere de varios pasos o comandos para alcanzar la terminación. Ya que la gente puede olvidar completar cada paso de una acción, los diseñadores intentan ofrecer una sucesión de pasos como una acción única. En un automóvil, el conductor no tiene que colocar dos switches de la señal vuelta a la izquierda para cada luz. Un solo switch ocasiona que ambas señales de vuelta se iluminen (frente y retaguardia) sobre el lado izquierdo del automóvil para destellar. Cuando un piloto baja el tren de aterrizaje, centenares de pasos y verificaciones se llaman automáticamente. Este mismo concepto puede aplicarse a los usos interactivos de computadoras. Por ejemplo, a la secuencia de marcar, parámetros de configuración de comunicación, entrar en comunicación, y los archivos de carga frecuentemente ejecutados por muchos usuarios. Afortunadamente, la mayoría de los paquetes de software de computación de comunicaciones activan usuarios para especificar estos procesos una vez, y entonces ejecutarlos simplemente seleccionando el nombre apropiado.

La construcción de bucles de los lenguajes de programación requiere que un MIENTRAS-HAZ-COMIENZO-FIN (WHILE-DO-BEGIN-END) o PARA-SIGUIENTE (FOR-NEXT)estas estructuras deben estar completas, pero a

veces los usuarios olvidan poner la estructura completa, o ellos borran un componente pero no los otros componentes. Una de solución sería, indicar a los usuarios que ellos quieren que un bucle, y que el sistema suministre la sintaxis completa y correcta, que llenaría el usuario. Este enfoque reduce escribir y la posibilidad de cometer algún error tipográfico o un resbalón, tal como la omisión de un de componente.

La noción de sucesiones completas de acciones puede ser difícil de implementar, porque los usuarios pueden necesitar emitir acciones atómicas así como también secuencias completas. En este caso, a los usuarios se les debería permitir que defina sucesiones que posean el concepto de subrutina o macro debería ser disponible a cada nivel de uso.

Los diseñadores pueden reunir la información sobre las potenciales, secuencias completas y estudiar las sucesiones de comandos que actualmente son usadas y los patrones de errores que la gente actualmente hace.

3- **Comandos Correctos:** Los diseñadores Industriales de comandos reconocen que los productos exitosos deben ser seguros y deben impedir que el usuario haga uso incorrecto del producto. Los motores de avión no pueden ponerse en reversa hasta que el tren de aterrizaje haya tocado tierra, y los automóviles no pueden ponerse en reversa mientras van viajando hacia adelante a más de 10 kilómetros por hora. Muchas cámaras simples impiden exposiciones dobles (aunque el fotógrafo puede querer exponer un recuadro dos veces).

Los mismos principios pueden aplicarse a sistemas interactivos. Considere estos errores típicos hechos por los usuarios de sistemas de computadora: Ellos llaman comandos que no son disponibles, elecciones de menú de que no son permitidas, piden archivos que no existen, o introducen valores que no son aceptables. Estos errores son ocasionados frecuentemente por molestos errores tipográficos (de Tecleo), tales como usar una abreviatura incorrecta de comando; presionar un par de teclas, en vez de una sola tecla deseada; deletrear mal un nombre de archivo; o hacer un error menor tal como omitir, insertar, o transponer caracteres. La gama de mensajes de error van desde un breve recordatorio o QUE?, hasta los vagos COMANDO IRRECONOCIBLE o ERROR DE SINTAXIS, al condenatorio NOMBRE INCORRECTO DE ARCHIVO o COMANDO ILEGAL. El breve es apropiado para usuarios expertos quien han hecho un error trivial y puede reconocerlo cuando ellos ven la línea de comando sobre la pantalla. Pero si un experto se ha aventurado a utilizar un nuevo comando y ha entendido mal su operación, entonces el mensaje breve no es útil.

Algunos sistemas ofrecen terminación automática de comando que permite al usuario escribir simplemente unos cuantas letras de un comando grande. El usuario puede requerir que la computadora complete el comando al presionar la tecla barra espaciadora, o la computadora puede completarlo tan pronto como la entrada sea suficiente para distinguir el comando de otros. La terminación automática de comando puede grabar teclasos y es apreciado por muchos

usuarios, pero puede también ser perturbador porque el usuario deberá considerar cuántos caracteres a de escribir para cada comando, y deberá comprobar que la computadora ha hecho la terminación que se destinó.

Otro enfoque está en tener la computadora ofreciendo los comandos permisibles, las elecciones de menú, o los nombres de archivo sobre la pantalla, y dejar que el usuario seleccione con un dispositivo apuntador. Este enfoque es efectivo si la pantalla tiene espacio amplio, el porcentaje de visualización es rápido, y el dispositivo indicadores rápido y exacto. Cuando el listado crece demasiado se deberá adaptar sobre el espacio disponible en pantalla, algunos usan la descomposición jerárquica. Imagine que los 20 comandos de un sistema operativo se visualizan constantemente sobre la pantalla. Después los usuarios seleccionan el comando IMPRIMIR (o el icono), el sistema automáticamente ofrece el listado de 30 archivos para la selección. Los usuarios pueden hacer la selección por lápiz óptico (lightpen), pantallas de toque (touchscreen), o ratón (mouse) en menos tiempo y con mayor exactitud de lo que ellos podrían escribir el comando IMPRIMIR GASTOS ENERO -JUNIO. No es siempre fácil de convertir un comando complejo en un número pequeño de selecciones y así reducir errores. Indicar listados largos puede ser demandante visualmente y molesto si usuarios son los meanógrafos competentes.

TEORÍA DEL COLOR

CONCEPTOS DEL COLOR

No hay duda de que todos sabemos lo que significa color y hemos entendido y reconocido los colores desde muy pequeños. ¿Pero sabemos realmente lo que es color? ¿Es acaso el factor diferencial entre objetos que de otra manera serian idénticos? ¿O es quizá la experiencia de percepción de diferentes longitudes de las ondas luminosas de las figuras? ¿O el matiz, luminosidad y saturación de la luz o de los objetos?.

Estas tres definiciones de color son técnicamente correctas. El color puede ser descrito y tratado desde una variedad de perspectivas. Quizás sea esta la razón por la cual la definición en el diccionario ocupa tanto espacio. En este apartado, haciendo a un lado las definiciones académicas, al utilizar el termino color nos referiremos a:

La combinación de substancias fosforescentes de color en un monitor de computadora, o la combinación de tintas sobre el papel, que se perciben como color.

El aspecto más importante del color que se debe tener presente es que cada individuo percibe y reacciona al color de manera diferente. En esta sección se usan algunos términos referentes al color que se describen brevemente a continuación:

- **Colores Complementarios** . - están ubicados uno frente al otro en la rueda de color. Por ejemplo el rojo y el verde son colores complementarios.
- **Matiz.**- es el color, o el grado de color.
- **Colores Relacionados.**- Se encuentran muy cerca uno del otro en la rueda de color. Por ejemplo el amarillo y el dorado son colores relacionados.
- **Saturación.**- Es la intensidad o cantidad de color. Los colores saturados tienen una apariencia mas viva o profunda.
- **Valor.**- es la claridad u obscuridad relativa de un color con respecto al negro.

LA LUZ EN CONTRAPOSICIÓN A LA TINTA

En esta sección se explicara como se genera el color a través de la computadora y luego como breviario cultural como se imprime sobre el papel. Para los propósitos de este apartado se definió en párrafos anteriores el concepto del color como: La combinación de substancias fosforescentes de color en un monitor de computadora, o la combinación de tintas sobre papel, que se perciben como color. ¿Que se da a entender cuando decimos la combinación de substancias fosforescente en un monitor de computadora, o de tintas sobre papel? ¿Cómo se crea el color en la luz? ¿Cómo se crea en la impresión?

El color es un fenómeno de percepción no un componente o una característica de una sustancia. El color es un aspecto de la visión, es una respuesta consistente en una reacción física del ojo y de la interpretación automática del cerebro.

Newton realizó varios experimentos y descubrió que el color está en la luz no en los objetos y que la unión de todos los colores del espectro producen la luz blanca. Existen dos clases de colores primarios, los colores primarios luz y los colores primarios pigmento:

Los colores primarios luz se refiere a los colores que se forman al descomponer la luz, mientras que los colores primarios pigmento son los colores con los cuales se conforman los colores para impresión.

COLORES PRIMARIOS LUZ

La luz blanca está compuesta por tres colores primarios: rojo, verde y azul. Existen también colores complementarios luz, también conocidos como secundarios; estos son amarillo, magenta y cyan; dichos colores son producidos por la combinación de los colores primarios luz en proporciones iguales.

- rojo+verde=amarillo
- rojo+azul=magenta
- verde+azul=cyan

COLORES PRIMARIS PIGMENTOS

Los colores complementarios luz (secundarios) son los colores primarios pigmento, es decir amarillo, magenta y cyan. El colorante que absorbe una de las luces tiene la combinación de las otras dos.

RGB

En la computadora, el color se especifica usando los valores RGB. RGB es la abreviatura de rojo, verde y azul, siglas en ingles de red (rojo), green (verde) y blue (azul), y se refiere a la forma en que estos tres colores de luz se combinan con diferentes intensidades para producir la aparición de muchos colores diferentes en el monitor a color de una computadora. Cada pixel del monitor a color consiste de tres colores de substancias fosforescente: rojo, verde y azul. Cada fósforo emite solo un color de luz, la intensidad de la luz emitida puede variar indivídualmente para los tres fósforos que forman cada pixel. Las combinaciones de estos tres colores fósforos, donde cada color de substancia fosforescente esta apagada o encendida, producen ocho colores básicos: negro, rojo, verde, azul, cian, magenta, amarillo y blanco. Al activarse todos los fósforos se produce el color blanco; al desactivarse todos los fósforos, se produce el color negro. Cada fósforo rojo, verde o azul del monitor a color de una computadora es capaz de producir un máximo de 256 intensidades del color del fósforo, dependiendo del tipo de monitor y de la tarjeta de vídeo. Al variar las intensidades de los fósforos en cada pixel, los monitores a color pueden mostrar aproximadamente 16 millones de colores diferentes. Como se forman los colores básicos RGB Ver figura.3-15

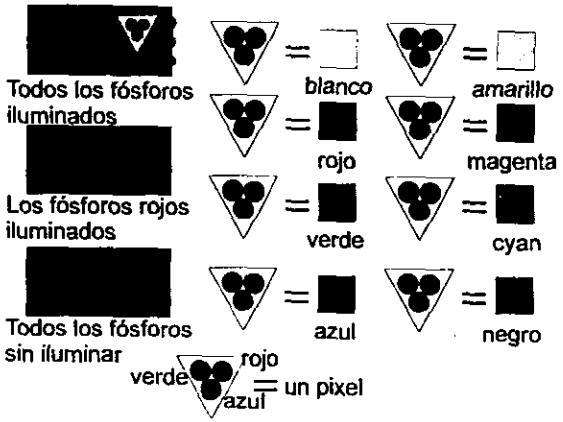


Figura 3.15 El proceso de creación del color metodo RGB

En la teoría del color existen dos principios que son: la síntesis aditiva del color y la síntesis substractiva del color de las cuales hablaremos a continuación. La suma de los colores complementarios en proporciones iguales nos producen un color primario. El combinar colores complementarios luz en proporciones diferentes producen colores intermedios de la luz blanca y la combinación de todos los colores luz producen la luz blanca y la ausencia de colores luz producen el negro, que es lo inverso en colores pigmento.

SINTESIS ADITIVA DEL COLOR

La síntesis aditiva es utilizada principalmente en las pantallas de televisión y los monitores de computadora donde existe un mosaico de rojo verde y azul, creados por una gran cantidad de puntos, el ojo no puede distinguir a simple vista este mosaico pero si ve la composición de colores que se forman. Las principales síntesis aditivas del color son las siguientes

a) dos colores primarios en proporciones iguales crean un color secundario

b) la unión de todos los colores primarios luz en proporciones iguales crean el blanco

c) la unión de dos o más colores primarios luz en proporciones diferentes crean otros colores.

El RGB es un proceso aditivo que emite y combina diferentes intensidades de luz roja, verde y azul para producir la apariencia de diversos colores.

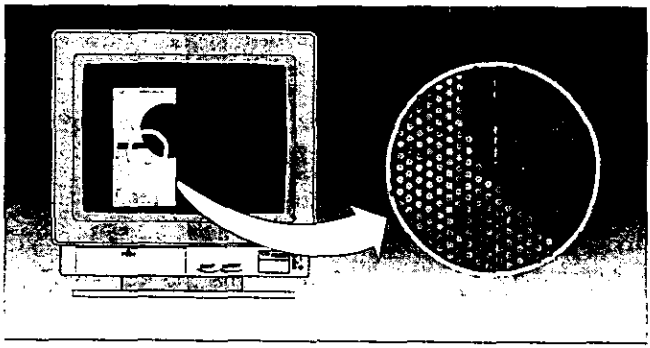


Figura 3-50 Resultado del proceso de la Síntesis Aditiva del Color

La combinación de colores complementarios en proporciones diferentes nos producen una gran variedad de colores intermedios de la luz y la combinación de todos estos colores complementarios nos crean el color negro.

Como ya se menciono, el color pigmento negro se forma a partir de la combinación de todos los colores primarios pigmento, pero en las impresoras de color; el color negro se utiliza para la saturación del color es decir por medio del color negro se le da la intensidad o tonalidad a los diversos colores.

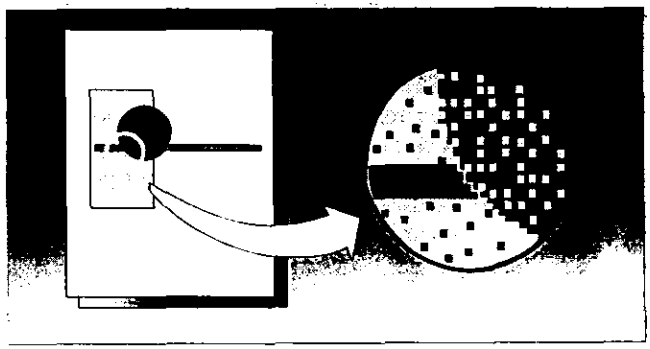


Figura 3-52 Resultado de la Síntesis Substractiva del Color

Después de explicar cómo el sistema de computo describe la información del color ¿Cómo se imprime color sobre la pagina? Según se ha descrito anteriormente, la impresora puede producir ocho colores básicos. Usando estos ocho colores básicos, el controlador de impresión emplea diversas técnicas para mezclar colores, combinando sistemáticamente pixeles de los ocho colores básicos para producir mas tonos de colores. Uno de los métodos para combinar colores que usa el controlador de impresión es el llamado Diseño (también llamado dithering); en la modalidad diseño se producen mas tonos de color o escala de grises definiendo un tamaño nuevo y mas grande de pixel llamado celda combinada (de parpadeo), esta celda combinada (de parpadeo) consiste de un pixel de 8 por 8 puntos de color que forman un diseño especifico. El diseño impreso aparece como un solo color, aun cuando sea una combinación de los ocho colores básicos. (Ver fiugra 3-19)

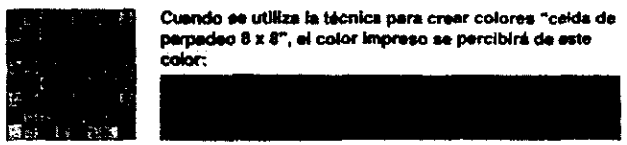


Figura 3-53 El proceso de Diseño oDither

CYMK

El color se especifica en la impresora usando los valores de los colores CYMK (abreviatura de las siglas en ingles de cyan, yellow (amarillo), magenta y black (negro); (Se utiliza K en vez de B para evitar confusión con la palabra inglesa blue que significa azul. CYMK se refiere a la forma en que estos tres colores de tinta se combinan para producir diferentes colores en la impresora. Cuando se imprimen combinaciones de puntos de las tintas cyan, amarillo y magenta se producen pixeles de los ocho colores básicos: negro, rojo, verde, azul,cyan, magenta, amarillo y blanco. El blanco se produce cuando no se usa ninguna de las tintas (quedara el color del papel que sé este usando) y el negro se produce mezclando los tres colores de tinta.

La impresora puede producir realmente cualquier color que se desee variando la cantidad de cada tinta que se aplique a la pagina o combinando los ocho colores básicos en una variedad de diseños de puntos. Los colores CYMK se obtienen por medio de un proceso substractivo, absorbiendo (o substrayendo) algunas ondas luminosas y reflejando los colores luz que queden para producir la apariencia de diversos colores sobre la pagina.

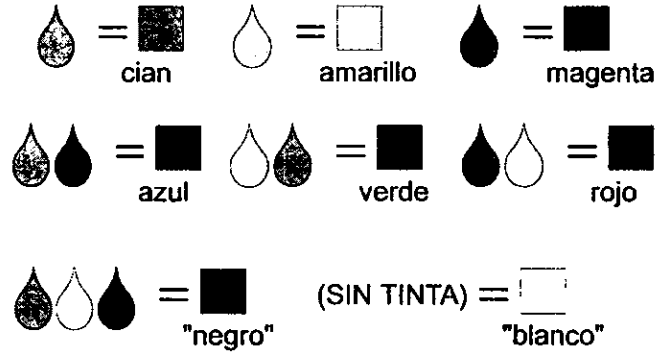


Figura 3-17 El proceso de creación del color metodo CYMK

SINTESIS SUBSTRACTIVA DEL COLOR

El colorante que absorbe una de las luces tiene la combinación de las otras dos y este es el complemento del color que ha sido substraído de la luz blanca. Por síntesis substractiva la suma de dos colores en proporciones iguales nos producen un color primario por ejemplo:

magenta y amarillo absorbe el verde y el azul dejando ver únicamente el rojo.

Para llenar un área con un tono especifico, las celdas de combinadas se ordenan en filas y columnas, como baldosas. Si los bordes del área no son ángulos rectos, las celdas combinadas son recortadas, como se muestra en la figura 3-20. El controlador de impresión

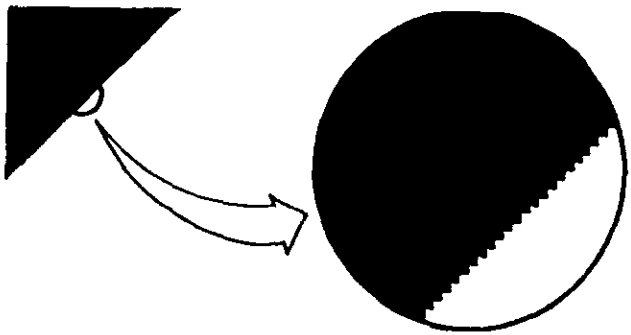


Figura 3-20 Celda combinada de parpadeo

puede describir millones de tonos de colores a la impresora usando las diferentes técnicas para combinar los ocho colores básicos. Otro método para combinar colores que se utiliza es el conocido como Disperso (scatter), el cual funciona de manera similar al método diseño pero con la diferencia de que coloca los puntos de tinta al azar, reduciendo así los diseños de puntos geométricos que aparecen algunas veces cuando se utiliza el método de diseño.

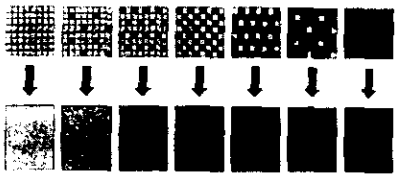


Figura 3-21 Metodo para combinar colores llamado Disperso o Scatter

COLOR

El color es una herramienta de comunicación muy efectiva cuando se usa adecuadamente. Esta sección ofrece sugerencias sobre como seleccionar y diseñar con color así como los peligros del mal uso. A fin de mejorar las interfaces y los materiales de comunicación.

Las exhibiciones de color son atractivas a los usuarios y pueden mejorar frecuentemente el desempeño de tareas, pero el peligro de emplear mal el color es alto. El color puede:

- Apaciguar o golpear al ojo
- Agregar acentos a una exhibición sin interés
- Facilitar discriminaciones sutiles en exhibiciones complejas
- Enfatizar la organización lógica de la información
- Atraer la atención a advertencias
- Provocar reacciones emocionales fuertes como regocijo, excitación, temor, o coraje

Los principios desarrollados por artistas gráficos para usar color en libros, revistas, signos de carretera, y televisión ahora son adaptados para las exhibiciones de computadora (Thorell y Smith, 1990; Marcus, 1992). Los programadores y los diseñadores de sistemas interactivos rápidamente aprenden como crear las exhibiciones efectivas de computadora y evitar el peligro.

No hay duda que el color hace a los video juegos más atractivos para los usuarios, transmite más información sobre una planta de poder o los diagramas de control de procesos, y es necesario para imágenes realistas de gente, escenarios, u objetos tridimensionales. Estas aplicaciones requieren color. La mayor controversia existe sobre los beneficios del color para exhibiciones alfanuméricas.

Ninguno conjunto simple de reglas rige el uso del color, pero un número de directivas puede llegar a ser el punto de partida para diseñadores:

COMO DISEÑAR CON COLOR.

LA RUEDA DE COLOR Y LA RELACIÓN ENTRE COLORES (USO)

La selección de los colores apropiados para el trabajo actual o tarea, requiere de un nivel de comprensión de las relaciones entre los colores. La rueda de color es muy útil para identificar las combinaciones de colores que armonizan y aquellas que deben evitarse.

Seleccione un matiz en la rueda de color para unificar los documentos. Variando la claridad u obscuridad del color hará que se distingan los elementos y se imparta variedad.

- La selección de dos matices complementarios en la rueda de color distrae la vista debido a que los colores parecen vibrar cuando están juntos. Si bien el uso de colores complementarios no se recomienda, se puede cambiar la saturación o el valor de uno de los colores complementarios para lograr que armonicen.
- La selección de dos matices que se encuentran adyacentes en la rueda de color crea armonía debido a que los colores adyacentes están relacionados.
- La selección de matices con tres colores de por medio en la rueda de color crea contraste. Para obtener combinaciones de colores que contrasten, use colores más obscuros de trasfondo y colores más brillantes en primer plano.
- Al seleccionar los colores, tenga en cuenta que el blanco y el negro también

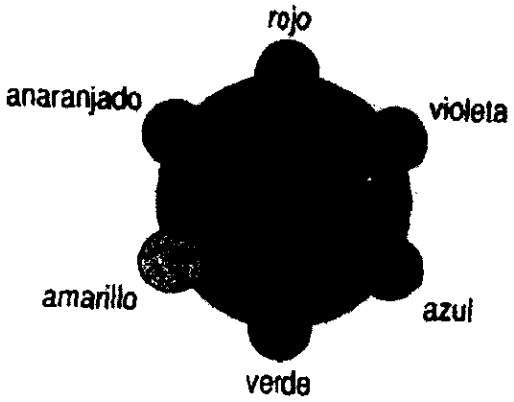


Figura 3-22 La Rueda del Color sirve para diseñar con color

cuentan como alternativas de color. Por lo tanto, al diseñar con color, planee el uso tanto de negro, blanco (o espacio en blanco) y color según sea necesario.

DIRECTIVAS O SUGERENCIAS

Uso del color conservadoramente: La capacidad de crear virtualmente cualquier combinación de colores aumenta la tentación de usar todas las combinaciones posibles. *!Resista la tentación!*, reserve el uso del color para producir impacto en el lector y para que sirva de ayuda a la compresión del tema. Muchos programadores y diseñadores novatos son ávidos para usar color para iluminar sus exhibiciones, pero los resultados son frecuentemente contraproducentes. Un sistema de información domestico tuvo siete letras en su nombre con grandes letras, cada letra con un color diferente. A una distancia, la exhibición pareció acogedora y llamativa; al acercarse, sin embargo, era difícil de leer.

En vez de mostrar relaciones significativas, colorear inapropiadamente los campos extravían a los usuarios en buscar relaciones que no existen. En un diseñó de exhibición pobre, el letrero blanco se usó para los campos de entrada y para explicaciones de las teclas PF (teclas de ayuda), conduciendo a los usuarios a pensar que ellos tenían que oprimir las teclas PF3 o PF9. Usar un color diferente para cada selección en un menú de 12 opciones produce un efecto abrumador. Usar cuatro colores (tales como rojos, azules, verdes, y amarillos) para las 12 opciones, extraviarán todavía a los usuarios en pensar que todos las opciones similarmente coloreados se relacionan. Una estrategia apropiada sería mostrar todos las selecciones de menú en un solo color, el título en un segundo color, las instrucciones en un tercero color, y mensajes de error en un cuarto color. Igualmente esta estrategia puede abrumar si los colores golpean demasiado visualmente.

- **Limite del número de colores:** Muchas guías de diseño sugieren limitar el número de colores en una exhibición alfanumérica únicamente a cuatro, con unos siete colores en la sucesión entera de exhibiciones. El usuario experimentado, puede ser capaz de beneficiarse de un número más grande de códigos de color.
- **Reconocer el poder del color como una técnica para codificar:** Use el color para explicar, no como decoración. El color acelera el reconocimiento para muchas tareas, pero puede inhibir el desempeño de otras tareas que van contra el esquema de codificar por color. Por ejemplo, en una aplicación de contabilidad, si las líneas de datos de cuentas morosas con más de 30 días se codifican en rojo, estas serán fácilmente visibles entre las cuentas no morosas codificadas en verdes. En el control de trafico aéreo, los aviones que vuelan alto podrían codificarse de manera diferente de los aviones que vuelan bajo para facilitar su reconocimiento. Al programar podrían codificarse, declaraciones de manera diferente desde las declaraciones viejas hasta las recientes con el fin de mostrar el progreso de escritura o mantenimiento de programas.

  Si, en la aplicación de contabilidad con el color que codifica de día moroso, la tarea es ahora para ubicar cuentas con balances de más de $55, el color codificante de día moroso puede inhibir el desempeño sobre la segunda tarea. En la aplicación de programar, el código de color de adiciones recientes puede hacer más difícil de leer el programa entero. Los diseñadores deberán intentar hacer un enlace cercano entre tareas de los usuarios y el codificar color.
- **Codificar el color con el esfuerzo mínimo del usuario:** En general, el color codificado no debería tener que ser asignado por los usuarios cada vez que ellos desempeñan una tarea, pero debiera aparecer para ellos, por ejemplo, inician el programa para verificar para cuentas morosas por más de 30 días. Cuando los usuarios desempeñan la tarea de ubicar cuentas con balances de más de $55, el nuevo color codificado debería aparecer automáticamente.
- **Dejar el código de color bajo el control de usuario:** Cuando fuera apropiado, los usuarios deberían ser capaces de apagar el color codificante.

Por ejemplo, si en un verificador de ortografía colorea las posibles palabras mal escritas en rojo, entonces el usuario debería ser capaz de aceptar la verificación y apagar el codificando. La presencia del rojo altamente visible codificando, es una distracción para leer el texto así como para la comprensión.

- **Diseñar primero en monocromo:** La primera meta de un diseñador de exhibición debería estar al colocar los contenidos en un modelo lógico. Los campos relacionados pueden ser mostrados por la contigüidad o por modelos estructurales similares; por ejemplo, los registros consecutivos de empleado pueden tener el mismo modelo de indentación. Los campos relacionados pueden también ser agrupados por una caja dibujada alrededor del grupo. Los campos no relacionados pueden ponerse separados por espacio en blanco por lo menos una línea vacía verticalmente o tres caracteres vacíos horizontalmente. Las exhibiciones en monocromo deberían, considerarse seriamente como el formato primario porque aproximadamente el 8 por ciento de varones en comunidades Europeas y norteamericanas tiene alguna forma de daltonismo. Puede ser ventajoso diseñar para monocromo porque exhibiciones de color no pueden ser universalmente disponible. Los diseños monocromos pueden aumentar el auditorio de consumo de un producto o permitir la operación cuando un monitor a color falle.
- **Usar el color para ayudar en formateo:** En exhibiciones en donde hay gran densidad de objetos donde el espacio es un premio, los colores similares pueden usarse para agrupar artículos relacionados. Por ejemplo, en una exhibición de un despachador tabular de asignaciones de la policía, los automóviles de policías sobre los llamados de emergencia podrían codificarse en rojo y los automóviles de policías sobre los llamados de rutina podrían codificarse en verde. Entonces, cuando una nueva emergencia,

Figura 3-23 Ejemplo de como se puede usar Color en una GUIpara un lenguaje de programación

surgiera, sería relativamente fácil de identificar los automóviles sobre llamadas de rutina para asignar uno a la emergencia. Diferentes colores pueden usarse para distinguir físicamente campos lógicamente distintos. En un bloque estructurado de un lenguaje de programación, los diseñadores podrían mostrar los niveles de anidamiento al codificar las declaraciones en una progresión de colores por ejemplo, verdes obscuros, verdes claros, amarillos, naranjas claros, naranjas obscuros, rojos, etc.

- **Ser consistente en el código de color:** Use las mismas reglas para la codificación del color a lo largo del sistema. Si los mensajes de error están en rojo, entonces habrá que asegurarse que cada mensaje de error aparezca en rojo; un cambio al color amarillo puede interpretarse como un cambio en la importancia del mensaje. Si los colores se usan de manera diferente por varios diseñadores del mismo sistema, entonces los usuarios dudarán como ellos intentan asignar el significado a los cambios de color. Un conjunto reglas sobre la codificación de color deberán anotarse para el beneficio de cada diseñador.
- **Estar alerta a las expectativas sobre los códigos de color comunes:** El diseñador necesita hablar con el operador del sistema con el fin de determinar qué códigos de color se aplican en cada tarea. Por ejemplo un conductor de automóvil con experiencia, el rojo lo considera usualmente para indicar parada o peligro, amarillo es una advertencia, y verde seguir. En el circulo financiero, el rojo es una la pérdida financiera y negro es una ganancia. Para los ingenieros químicos, el rojo es caliente y azul es frío. Para los que hacen mapas, azules significa agua, verdes significa bosques, y amarillos significa desiertos.

  Estas múltiples convenciones pueden ocasionar problemas a los diseñadores. Un diseñador podría considerar usar rojo para señalizar que un motor se sobrecalienta y listo, pero un usuario podría comprender el rojo codificando, como un indicio de peligro. Una luz roja se usa frecuentemente para indicar que esta encendido un equipo eléctrico, pero algunos usuarios se han hecho ansiosos por esta decisión porque el rojo tiene una fuerte asociación con el peligro o parar. Cuando sea apropiado, hay que indicar el código de color y sus interpretaciones sobre la exhibición o en un panel de ayuda.
- **Estar alerta a problemas con parejas de color:** Si se satura la exhibición de rojo y azul (puros) al mismo tiempo, puede ser difícil para usuarios absorber la información. El rojo y azul están sobre los extremos opuestos del espectro, y los músculos que circundan el ojo humano se contraerán por intentos de producir un foco brusco para ambos colores simultáneamente. El color azul parecerá retroceder y el rojo parecerá venir hacia adelante. El texto azul sobre fondo rojo presentaría un especial, desafío difícil a los usuarios para leer. Similarmente, otras combinaciones parecerán ser llamativas pero difíciles de leer por ejemplo, amarillo sobre morado, magenta sobre verde. Muy poco contraste también es un problema: Imagine cartas

amarillas sobre fondo blanco o cartas marrones sobre fondo negro. En cada monitor a color, el color aparece de manera diferente, y pruebas cuidadosas con los diversos colores de fondos y texto son necesarios. La marcha (1984) probó 24 combinaciones de color sobre un monitor de Amdek que conectó al IBM PC, usando 36 estudiantes universitarios. El encontró la tarifa de errores oscilada desde aproximadamente uno a cuatro de errores por 1000 caracteres leídos. El negro sobre azul y azul sobre el blanco fueron dos colores con porcentaje de error bajo en ambas tareas, y el magenta sobre verde y verde sobre el blanco fueron dos colores con un alto porcentaje de error. Las pruebas con otros monitores y analizar el impacto del color sobre las tareas son necesarias, para alcanzar una conclusión general sobre los pares de color más efectivos.

- **Usar los cambios de color para indicar cambios de condición (estado):** Si un velocímetro de automóvil tiene una lectura digital de la velocidad, podría ser útil cambiar desde números verdes para límites de velocidad bajos al rojo para límite superiores de velocidad y actuar como una advertencia. Similarmente, en una refinería de petróleo, los indicadores de presión podrían cambiar de color conforme el valor estuviera arriba o abajo de los límites aceptables. De esta manera, color actúa como un método atención - prosigue. Esta técnica es potencialmente importante cuando hay centenares de valores continuamente mostrándose.
- **Usar color en exhibiciones gráficas para mayor densidad de información:** En gráficas con múltiples porciones, el color puede ser útil en mostrar que segmentos de recta forman el diagrama lleno. Las estrategias usuales para diferenciar líneas en gráficas en blanco y negro son las líneas punteadas, las líneas más gruesas, y las líneas tachadas pero estas estrategias no son tan efectivas como usar colores separados para cada línea. Por ejemplo los planos arquitectónicos se benefician del color que codifica las líneas de instalaciones eléctricas, telefónicas, agua-caliente, agua-fría , y gas-natural. De manera similar, los creadores de mapas pueden tener la mayor densidad de información cuando se usan códigos de color.
- **Cuidar la pérdida de resolución con exhibiciones de color:** Muchas exhibiciones de color tienen resolución más pobre que las exhibiciones hechas en monocroma. Los beneficios de codificar color deben sopesarse contra la pérdida de resolución. Las exhibiciones de color pueden también ser más costosas, más pesadas, menos confiables, más calientes, y más grandes que exhibiciones monocroma. Christ (1975) informa sobre 41 estudios de los beneficios de color, pero poco de estos estudios se hicieron sobre CRTs. Tullis (1981) describe un experimento en el que usó color en una de cuatro de versiones de un display CRT. Las gráficas de color no mostraron una mejora en el rendimiento significativamente, pero las reacciones del usuario fueron positivas: *" El color destaca partes importantes,"* *" Da gran velocidad y es muy claro " y" el Color agrega fuerza a una tarea que de otra manera seria aburrida."* La complejidad de usar color se

demostró en los estudios de decisión al hacer las tareas como, simple ubicación de información o recordar información, con sistemas de manejo de información. Aunque haya códigos de color se encontró que para ser beneficioso y preferido, había una interacción con factores de personalidad.

Relaciones adicionales intrincadas se encontraron en una comparación con gráficas con códigos de color versus gráficas monocroma en gráficas de pastel, gráficas de barra, diagramas de línea, y tablas de datos en el que color mejoro el desempeño en todos. El color es un variable sutil que puede mejorar significativamente la capacidad de tomar una decisión para extraer información

En resumen, con exhibiciones a color se aumenta la calidad en los diseños de sistemas.Hay indudablemente beneficios: por ejemplo más satisfacción del usuario y aumento frecuentemente en el desempeño; sin embargo, hay peligros verdaderos en el mal empleo del color. Los diseñadores deben tener cuidado para hacer diseños apropiados y para conducir evaluaciones completas

Las directivas realzan las potencialidades complejas para beneficios y los peligros con el color que codifica de visualizaciones alfanuméricas.

DIRECTIVAS PARA EL USO DEL COLOR

Beneficios

- Apacigua o golpea el ojo
- Acentúa una visualización sin interés
- Facilita discriminaciones sutiles en visualizaciones complejas
- Enfatiza la organización lógica de información
- Atrae la atención a advertencias
- Provoca reacciones emocionales como regocijo, excitación, temor, o coraje

Directivas

- Utilice el color conservadoramente: limite el número y cantidad de colores.
- Reconozca la potencia del color para acelerar o alentar tareas.
- Asegúrese que el color que codifica apoya a la tarea
- Codifique el color con el esfuerzo mínimo del usuario.
- Deje el código de color bajo el control de usuario
- Diseñe primero en monocroma:
- Utilice color para ayudar en el formateo.
- Sea uniforme en el código de color.
- Este alerta a las expectativas sobre los códigos de color comunes.
- Use los cambios de color para indicar cambios de condición (estado)

- Utilice color en visualizaciones gráficas para mayor densidad de información.

Peligros

- La resolución puede degradar a las visualizaciones de color.
- Las parejas de color pueden ocasionar problemas.
- La fidelidad de color puede degradar sobre otro Hardware.
- La impresión o conversión a otros medios puede ser un problema.
- Ajuste de la brillantez y el contraste

Generalmente un mayor contraste con relación a la brillantez es lo mejor, pero para algunas personas es mejor el alto contraste con una brillantez media.

El color de la pantalla *"El blanco y el negro usualmente son los mejores"* ya que estos dos colores crean el mayor contraste, pero la exposición prolongada de la vista en una pantalla de este tipo puede provocar problemas visuales por lo que se recomienda usar fondo gris claro con letras negras ya que cansa menos la vista y se puede leer por prolongados periodos de tiempo otra combinación que se recomienda es fondos amarillos claros y cafés claros con letras negras o azules. En la actualidad se utilizan una gran variedad de fondos (Backgrounds) para la pantalla, desde fondos muy obscuros, hasta fondos muy claros, al igual que el color de la tipografía que se desea utilizar; es muy importante recordar que no en todos los fondos se pueden crear un buen contraste con las letras e imágenes que se van a mostrar, esta decisión de contrastes se deja en manos del diseñador de la GUI, ya que este decidirá si lo que desea, es crear un impacto visual, una buena lectura de la pantalla (información), o ambas cosas.

DIRECTIVAS DE USO DE LA BRILLANTES Y EL CONTRASTE

- Usar el contraste para el realce
- Fondos Oscuros con Letras Claras
- Fondos Claros con Letras Oscuras
- Cuidar el tipo y puntaje de letra
- El fondo gris semi-oscuro con letra negra a la larga cansa menos a la vista y es muy útil para lecturas prolongadas sobre la pantalla.

TEORIA DE LAS FUENTES

El objetivo de este punto es el de tomar en consideración y poner en práctica todos los factores necesarios para que un grupo humano asimile de una manera más

fácil y efectiva, una idea presentada en forma de palabras e imágenes. El diseñador debe conocer el perfil del grupo al que se dirige para corresponder a sus necesidades y lograr la satisfacción de las mismas.

Palabras e imágenes deben interrelacionarse de tal manera que no se contrapongan ni sean apreciadas como partes independientes, sino como elementos que dan sentido a la unidad total, cumpliendo un objetivo común, que es el de hacer más accesible un mensaje.

CONCEPTOS BÁSICOS:

El hombre se comunica a través de mensajes impresos que contienen símbolos de dos tipos:

- **Fonogramas.-** Símbolos visuales con sonido propio.
- **Logogramas.-** Símbolos puramente visuales.

MEDIDAS:

El sistema de medidas más antigua es el de puntos. Antes de su introducción en Francia (s. XVIII) cada tipógrafo podía escoger el tamaño que gustase. Para acabar con esta confusión, se ideó el sistema de puntos, el cual se basa en la división de la pulgada en 72 subdivisiones. En la actualidad funcionan dos sistemas: el Angloamericano y el Europeo.

La tipografía se mide en puntos, picas o cuadratines.

- 12puntos = 1 pica
- 1pulgada = 72 puntos (aprox.)
- 1pica = 0.42 cm.
- 1 pulgada= 6 picas

El alto de la tipografía, el interlineado, los espacios extras entre letras y palabras y plecas, se miden en puntos.

FAMILIAS TIPOGRÁFICAS:

Una familia tipográfica es un conjunto de tipos cuyos rasgos de diseño están ligados. Las variaciones dentro de una misma familia pueden ser de diversos tipos, anchos, peso, postura, pero a través de todos ellos los caracteres básicos y familiares persisten.

JUSTIFICACIÓN DE TEXTO:

Existen cinco alternativas en la composición de un texto:

- Líneas justificadas
- Líneas justificadas a la izquierda
- Líneas justificadas a la derecha
- Líneas justificadas con el eje central.
- Líneas desiguales sin patrón de repetición.

prueba para ver las diversas formas de justificacion

Justificación a la izquierda

prueba para ver las diversas formas de justificacion

Justificación al centro

prueba para ver las diversas formas de justificacion

Justificación a la derecha

prueba para ver las diversas formas de justificacion

Justificación completa

prueba para ver las diversas formas de justificacion

Sin Justificación

Figura 3-24 Las diversas justificaciones

¿QUE ES UN TIPO?:

La forma que los dibujantes dan a la letra y a los signos de los alfabetos creados por ellos, reciben el nombre de TIPO DE LETRA. Por lo que el Tipo de letra individual, figura o marca de puntuación llamado también Carácter.

FUENTE DE TIPOS:

Una fuente de tipos consiste en un surtido de caracteres que incluye: mayúsculas o caracteres de caja alta, minúsculas o caracteres de caja baja, figuras, fracciones, números, signos de puntuación etc... de un tamaño y cara particular. Una fuente puede variar en el número de caracteres que contenga, además del alfabeto y marcas de puntuación, algunas fuentes incluye caracteres especiales como símbolos matemáticos y acentos extranjeros.

CARACTERÍSTICAS DE LOS TIPOS:

Aparte de las características que corresponden a la forma: mayúsculas, minúsculas, etc..., un significado de crear variedad en lo que se refiere a las caras tipográficas es variando la proporción (anchura) y el peso de la cara. El tipo tiene tres aspectos básicos que son: Forma, Proporción y Peso de la cara.

- **Forma:** Está dada por sus características formales, tamaño e inclinación.
  - **Capitular o Mayúscula:** Se usa frecuentemente en los textos más sobresalientes de un escrito, como encabezados e inicios de textos.
  - **Minúsculas o Bajas:** Se utilizan estas letras más comúnmente que ninguna otra por su claridad. Esto se debe a la similitud que tienen con la letra normal caligráfica.
  - **Itálica, Cursiva o Inclinada:** Por su forma los textos escritos con este tipo de letra, permiten dar ritmo y agilidad a la lectura. En algunos casos, sirve para dar contraste cuando se combina con letras normales o redondas.

Capitular o Mayúsculas
ABCDEFGHIJKLMNÑ
OPQRSTUVWXYZ

Minúsculas o Bajas
abcdefghijklmnñ
opqrstuvwxyz

Itálica, Cursiva o Inclinada
abcdefghijklmnñ
opqrstuvwxyz

Figura 3-59 Ejemplo de las formas

- **Proporción:** Está dada a partir de la proporción, norma de una letra y su comparación de ésta; con su forma condensada y extendida.
  - **Condensada:** Esta tipografía se usa para economizar espacio cuando se tiene demasiado texto. En los encabezados, este tipo economiza espacio en forma horizontal y lo extiende verticalmente.
  - **Normal o Redonda:** Es una proporción estándar, que se adecua a cualquier escrito legible y simple.

- **Extendida o Abierta:** A diferencia del tipo condensado, este carácter ocupa el mayor espacio disponible de poco texto. En los encabezados funciona a la inversa del tipo condensado.

Fuente Century Condensada

Fuente Century Normal

Fuente Century Extendida

Figura 3.26 Ejemplo de la proporción en las fuentes

- **Peso de Cara:** Esta dado según el espesor de los trazos:
- **Light:** Este tipo es sinónimo de suavidad, se usa como recurso cuando en un original se encuentran columnas demasiado extensas, para que en esta forma se convierta una página pesada en un plano luminoso y legible.
  - **Médium:** Este peso de la cara es el más flexible, y puede ocuparse en cualquier texto.
  - **Bold:** Los caracteres de este tipo se imprimen para destacar la fuerza o contrastar notablemente con el resto del texto.
  - **Outline:** Las características de este tipo permiten introducirlo con gran legibilidad sobre fondos con variantes de tono o color. Esto se debe a que el perfilado de esta letra, separa al fondo del cuerpo de la misma.
  - **Inline:** Estos caracteres destacan más aún las cualidades del tipo outline.

Fuente CityD Light

Fuente CityD Medium

Fuente CityD Bold

Fuente Cooper Outline

FUENTE ATLANTIC INLINE

Figura 3.27 Los diferentes pesos de cara

¿QUÉ ES UNA FAMILIA TIPOGRÁFICA?

Es un conjunto de tipos, cuyos rasgos de diseño, coinciden o son similares. Las variaciones de los tipos dentro de una misma familia pueden ser diversas: formas, ancho y peso, pero a través de todas ellas, las características básicas y familiares existen.

CLASIFICACIÓN DE LAS FAMILIAS: Se dividen en cinco familias básicas:

- **Romana o Latinas:** son todos aquellos tipos básicos en el alfabeto clásico romano y fue con el diseño del alfabeto moderno de Nicolás Jenson cuando aparecieron las diferentes variaciones. La primera de ella fue la BEMBO, pero la más famosa fue la diseñada por el francés GARAMOND, a las que siguieron las del inglés Casino, el inglés Baskerville, el francés Didot y el italiano Bodoni. La belleza, clasicismo y legibilidad de estos caracteres, ha hecho que sean utilizadas y sigan utilizándose hasta nuestros días. Como continuadores del diseño romano, pueden citarse el tipo Times y la antigua Sabon.
- **Egipcias:** Son todas aquellas cuya característica básica es la forma rectangular de los patines en los vértices de sus bastones, estos tipos son generalmente gruesos de su carácter. Pertenecen también a esta familia los tipos Egyptian, Clarendon, Volta, etc.
- **Grotescas:** Forman parte de esta familia los tipos de palo seco, es decir los que no tienen ningún adorno ni patines (sanserif).

  Su estructura responde originalmente al diseño del alfabeto clásico Griego. El tipo más representativo de las grotescas es la futura. A este famoso diseño han surgido entre otros los tipos: Venus, Helvética, Haas, Akzidente y Univers.
- **Inglesas o Manuscritas:** Esta familia agrupa todos aquellos tipos basados en la escritura caligráfica y pueden ser tan variados como individuales.
- **Ornamentadas o Decorativas:** Esta última familia esta formada por todos aquellos tipos de letras cuyo diseño se basa en el estilo de la época a que pertenecen. Forman también parte de esta familia, todos aquellos tipos que siendo parecidos a los anteriores, no encajan en ninguna de las categorías mencionadas. Ver ejmplo de las direfentes familias en la figura 3-28

Romanas o Latinas
Garamond Bodoni
Times Clasica Garamond
Egipcias
Egyptian
Clarendon
Grotesca
Futura Lucida Sans
Comic Sans
Inglesas o Manuscritas
Embassy BrushScipt
Lucia *Shelley Andante*
Ornamentadas o Decorativas
DEXGOTHIC
DECORATED SCOPD
ALGERIAND

Figura 3-28 La clasificación de las familias de fuentes

ELEMENTOS DE LA LETRA:

- **Fuste:** Línea vertical. Determina la altura y el peso de la cara. En caso de más peso es el principal.
- **Barra de cruce:** Línea horizontal Subida en la parte de en medio poco mas abajo. Va de fuste a fuste o de fuste a muesca.
- **Muesca o Bucle:** Línea circular Determina la proporción, altura y peso de la cara.
- **Brazos:** Pueden marcar la proporción de la letra.

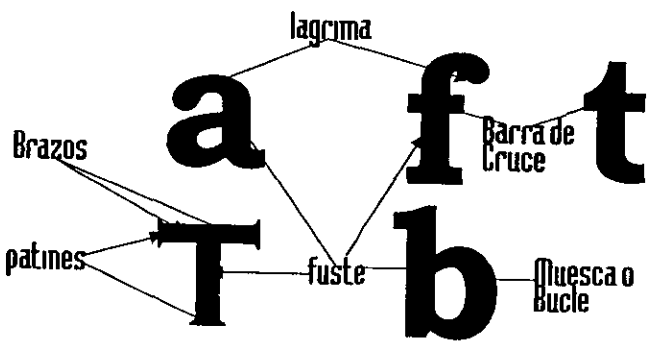


Figura 3-29 Elementos de la letra

- **Emplastamientos, Patines, Serifs o Pies:** *Determinan un estilo;* se refieren a los remates de las líneas principales: fuste, muesca y espina.
- **Lágrima:** Determina el estilo de terminación de los brazos.

APLICACIONES DE LA LETRA

Para aplicar una letra es necesario analizar su forma, peso, intencionalidad, etc. No por el hecho de ser una letra bella, puede cumplir con el propósito de nuestro mensaje, y debemos recordar que no existe una sola letra para cada mensaje, ya que con distintos tipos podemos dar el mismo mensaje.

Ciertas cualidades de la letra como puede ser la belleza, pueden ser aplicada no sólo a la forma, sino también al desplegado y a la ejecución. A simple vista resulta agradable a nuestros ojos, pero al leerlas su mensaje no se relaciona con su presentación. No hay mensaje, pero si estilo y pueden ser bellas y adaptarse a muchas cosas.

Las características generales que distinguen a una letra de otra para un uso adecuado son:

- **Tamaño:** Sirve para significar la relación entre la letra y la superficie donde aparecerá la letra. Tamaño también es el espacio que rodea la superficie en donde se aplica la letra y la distancia entre una y otra.
- **Forma:** El término forma, quiere decir el estilo o tipo de letra utilizado. Una vez que el tamaño ya esta definido, debe existir un análisis cuidadoso para definir la forma. Se tiene que considerar a la estructura de cada letra por *individual, no tanto* como una unidad, sino como parte de cada palabra y relacionar una letra con las demás.
- **Peso:** Las letras negras sobre fondo blanco serán de más peso, las letras delegadas sobre el mismo fondo se considerarán livianas. La letra pesada sirve para dar al mensaje importancia o atraer la atención; en cambio la letra liviana da velocidad, ritmo y legibilidad.
- **Layout**: Es la disposición que se le da a la pieza de letra, que se va a hacer sobre la superficie: tamaño, forma, peso. Esto se debe considerar para el inicio del trabajo. Es la composición general arreglada para obtener el más placentero resultado y toma en cuenta las posibilidades de contraste de letras negras sobre superficies blancas.

  El ritmo y el balance, se deben tomar en consideración sobre la forma y el peso: pero el layout debe de ser el responsable para la inmediata impresión del trabajo sobre el observador.
- **Espaciamiento:** Hay tres diferentes tipos de espaciamientos:
  - 1.- Entre letras
  - 2.- Entre palabras

– 3.- Entre líneas.

Cada uno de estos tipos de espaciamiento, pude agregar o disminuir la legibilidad, la cual pule el trabajo final. La forma de una letra, como el propósito de un trabajo describe la cantidad de trabajo que se requiere. La legibilidad es el punto más importante para lograr y asegurar un buen resultado en el mensaje.

- **Ejecución:** La ejecución de las letras debe de ser hecha en grandes dimensiones para que en el momento de la reducción, las pequeñas fallas que presenten desaparezcan en la diminución de la letra. El retocar con blanco, debe ser usado sólo para borrar pequeñas irregularidades, no para dar a la letra su forma normal.

LENGUAJE DE LA LETRA

La letra por sí misma, por su forma, contraste, tamaño: puede expresar ideas o puede reforzar el significado de las palabras:

- **Letra de Palos Seco:** Es indicada para expresar actualidad, mecanismo y fuerza.
- **Letra de estilo Romano:** Es indicada para expresar clasicismo, tradicionalismo, religión, arte y sensibilidad.
- **Letra Gruesa:** Es símbolo de fuerza, poder y energía.
- **Letra Delgada:** Simboliza debilidad, suavidad, elegancia y lujo.
- **Cursiva Mayúscula:** Símbolo de dinamismo.
- **Letra Mayúscula:** Indica título, encabezado, anuncio.
- **Letra Minúscula de estilo Romano:** Indica conversación, frase, charla.

La tipografía con características de imagen puede ser aplicada en tres áreas básicas para representar la historia.

1.- Alusiones del área geográfica de la historia.

2.- Alusiones de la época de que trata la historia.

3.- Alusiones del tema de la historia: esta se subdivide en dos categorías;

– a) La idea material que esta en discusión.

– b) La idea abstracta, inherente en la historia.

Gracias a la gran variedad de tipografía en seco, el poder representar una idea o una palabra se ha simplificado. Con solo hojear cualquier catálogo de tipos, encontraremos en ellos la tipografía que podrá interpretar el mensaje que queremos dar.

CÁLCULO TIPOGRÁFICO

Aunque no es muy comun usar el calculo tipográfico en el diseño de una GUI si puede ser util para realizar ayudas en linea o paginas Web.

Si se siguen correctamente las instrucciones, cualquier persona puede realizar un cálculo tipográfico. En el procedimiento, sólo hay un factor que puede hechar a perder el cálculo, y es el amplio espaciado que a menudo utilizan en el cajista manual u operario que maneja la máquina para composición. En el mejor de los casos, los métodos para determinar la longitud que tendrá un manuscrito son sólo aproximados. Por lo tanto, al hacer un cálculo del espacio es siempre recomendable redondear los cálculos y dejar un margen de seguridad del 5%. Cuando las líneas son cortas, pueden presentarse problemas, de suceder así habrá que aumentar el espacio entre palabras o exagerar los espacios blancos para poder pasar palabras a la línea siguiente. Se deben de contar los caracteres por línea, o hacer una aproximación de estos, contando cada espacio en blanco entre las palabras como un carácter. Después se multiplica para tener un total de caracteres. La cifra obtenida servirá para calcular los gastos de composición y el espacio que habrá de ocupara el texto una vez compuesto.

Desarrollo:

1.- Se multiplican las pulgadas de la cuartilla por el número de caracteres según el tipo de máquina utilizado:
4"x 10 caracteres = 40 caracteres en cada línea.

2.- Hallando un número de caracteres en cada línea, se multiplican estos por el número de lineas en cada cuartilla:
25 líneas x 40 caracteres = a 1000 caracteres en cada cuartilla

3.- Conociendo el número de caracteres que existe en cada cuartilla, se multiplican estos por el número total de cuartillas:
1000 caracteres x 300 cuartillas = 300 000 caracteres.

4.- Se determina el ancho de línea de composición en cuadratines, luego se busca el número de caracteres por cuadratín de la medida en puntos del tipo deseado, el cual se multiplica por el número de cuadratines de la línea.

5.- Se divide el resultado del tercer paso entre el cuarto y se obtienen las líneas de la justificación requerida.

6.- Sabiendo la altura de la página, dividir el número de puntos (72 ptos. = 1") entre el número de puntos en que está parado para saber cuantas líneas del mismo tipo caen en una pulgada.

7.- Se multiplica el resultado anterior por la altura, para saber cuantas líneas hay en la página.

8.- Se divide el resultado del paso número cinco entre el resultado del número siete para obtener el resultado de las páginas totales.

9.- Si hay columnas, se divide el resultado entre estas. Un texto debe leerse con facilidad y esto va a depender del tamaño de os tipos de letras, de la longitud de las líneas y del interlineado entre estas. El material impreso en formato normal habitualmente se lee a una distancia de 30 a 35 cm. El tamaño de los tipos debe calcularse para esa distancia. Es importante recordar, que cualquier dificultad en la lectura, significa perdida de comunicación, y que las líneas demasiado largas o muy cortas se pueden convertir en molestas, creando una pesadez en la lectura. El ancho adecuado de la columna, crea las condiciones para un ritmo regular y agradable, que posibilita la lectura distendida y por completo pendiente de su contenido. Las anchuras de columna dependen del tamaño de los tipos y de la cantidad de texto. El arreglo tradicional ha sido el de dos y tres colinas y es el que combina con los espacios del anunció.

Como la longitud de líneas, también debe dedicarse mucha atención al espacio de las mismas, conocido con el nombre de interlineado interlínea, porque al igual que la excesiva longitud o cortedad de las líneas, también la interlínea influye en la composición y con ello en la legibilidad del texto. Las líneas demasiado próximas entre si, perjudican la velocidad de la lectura puesto que entran al mismo tiempo en el campo óptico el renglón superior e inferior y con un interlineado excesivo al lector le cuesta encontrar la unión con la línea siguiente, la inseguridad crece y el cansancio llega con mayor rapidez. Un buen interlineado puede conducir ópticamente al ojo de línea en línea, le presta apoyo y seguridad el ritmo de la lectura se puede estabilizar rápidamente, lo leído se conserva mejor en la memoria.

DIRECTIVAS PARA USO DE TIPOS:

1.- Usar el menos número posible de tamaños y tipos, sin que esto limite la flexibilidad en expresar un tema en forma apropiada.

2.- Siempre hay que usar la misma familia para el texto y para los encabezados, hay que pensar en una familia que nos permita hacer el suficiente número de variantes... condensada, condensada itálica, médium, extendida, light, bold, etc.

3.- Restringir la tipografía en columnas estandarizadas en ancho y colocación en la página; crea un patrón que le da ritmo al flujo de la lectura sobre la pantalla y es otro elemento que ayuda en la unificación.

4.- Para el uso de cada tipo, hay que seguir especificaciones constantes en cuanto al espacio de interlínea y como debe de manejarse según el ancho de columna. Al estandarizar especificaciones logramos dos cosas:

- a.- Ofrecer al máximo de legibilidad en el espacio disponible;
- b.- Dar una mancha tipográfica uniforme a todo el largo de la interface.

5.- Hay que ser consistente en la colocación, espaciamiento y uso de la tipografía a través de la interface o ayuda. Hay que estandarizar todo lo posible como:

espaciamiento entre los diversos elementos que son encabezados, textos, margen y encabezado, espacio en torno a subtítulos, espacio entre fotografía y pié de foto. **TIP** para

el uso del tamaño de la fuente. El mejor puntaje de letra que hay para leer un documento largo en una pantalla de computadora es de 12 puntos

PUNTOS LEGALES

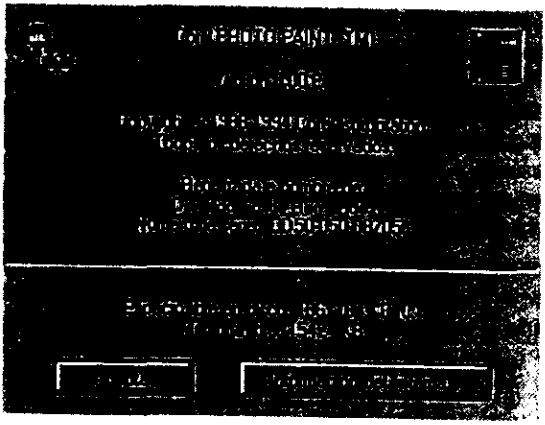
Como la interfaz de usuario ha llegado a destacar, los puntos legales serios han surgido. Los puntos de privacidad son siempre un interés cuando las computadoras se utilizan para almacenar datos o para controlar actividad: Médica, legal, financiera, militar, u otros datos que frecuentemente tienen que estar protegidos para impedir acceso no aprobado, alteración prohibida, pérdida inadvertida, o travesura maléfica. La seguridad física para negar el acceso es fundamental; además, la protección de la privacidad puede involucrar la interfaz de usuario en el cifrado y descifrado, acceso de contraseña (password), control de acceso de archivos, comprobación de identidad, y verificación de datos. La protección efectiva debe proveer un grado alto de privacidad con un mínimo de interferencia.

Un segundo punto es la seguridad y la fiabilidad. La interfaz del usuario para: una aeronave, automóviles, equipo médico, sistemas militares, o control de salas de reactores

Figura 3-64 Ejemplo de protección de propiedad o derechos de autor (copyright) para el software e información

nucleares pueden afectar decisiones de vida o muerte. Si un controlador de tráfico aéreo es temporalmente confuso por los contenidos de la visualización, puede conducir a un desastre. Si la interfaz de usuario de tal sistema se demuestra que es difícil para comprender, podría dejar al diseñador, desarrollador, y el operador desatar un pleito legal alegando diseño inadecuado. Los diseñadores deberían esforzarse en hacer interfaces de alta calidad bien probadas que adhieran un toque de arte en el diseño. El diseño de interfaz de usuario no es aún, una profesión establecida con estándares claros.

Un tercer punto es la protección de propiedad o derechos de autor (copyright) para el software e información (Ver Fig.3-30).Los desarrolladores de software quienes han gastado tiempo y dinero para desarrollar un paquete se frustran al intentar recuperar sus costos y tener una ganancia, si los usuarios potenciales piratean (hacer copias ilegales de) el paquete, en vez de comprarlo. Diversos esquemas técnicos han tratado de impedir el copiado, pero los inteligentes hackers pueden comúnmente evitar las barreras. Es inusual para una compañía entablar una acción judicial por la copia individual de un programa, pero los casos se han llevado contra corporaciones y universidades. Los acuerdos de licencia de sitio son una solución, porque estas permiten copiar dentro de un sitio una vez los costos



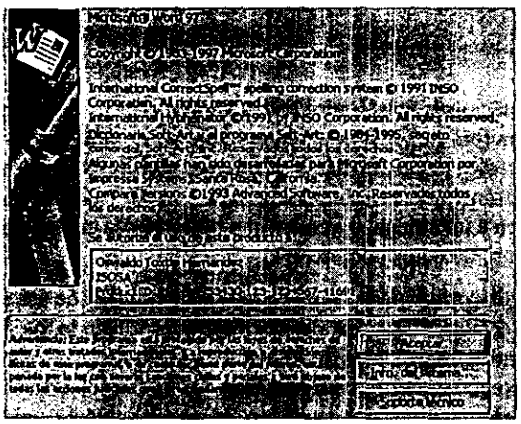
se han pagado. Situaciones más complicadas surgen dentro del contexto de acceso a la información en línea (online). ¿Si un cliente de un servicio de información en línea paga por el tiempo de acceso a la base de datos, el cliente tiene derecho para extraer y almacenar la información electrónicamente recuperada para posterior uso? ¿Puede el cliente enviar una copia electrónica a un colega, o vender una bibliografía cuidadosamente seleccionada desde una grande base de datos comercial?

El punto más discutible para los diseñadores de interfaz de usuario es: Los derechos propiedad y la protección de patente de sus interfaces. Cuando las interfaces de usuario comprendían comandos codificados, todos en letras mayúsculas sobre un Teletipo, habia poco que poder protegerse. Pero el surgimiento del diseño de interfaces usuario gráficas artísticamente, con animaciones y la ayuda en línea extensiva ha conducido a los desarrolladores a un archivo para la protección de propiedad (copyright). Esta actividad ha conducido a muchas controversias: ¿Qué material es elegible para la protección de propiedad? Desde fuentes, líneas, cajas, sombreado, y los colores no pueden usualmente acordarse propiedades, alguna gente reclama que la mayoría de las interfaces no son protegibles. Los férreos defensores de la protección sustentan que el conjunto de componentes es un trabajo creativo, simplemente como una canción que es compuesta es incopiable sus notas, arreglos y letra o un poema es incopiable sus palabras. Aunque arreglos estándares, tal como la L rotada del formato de las hojas de cálculo, no son protegibles, colecciones de palabras, tal como el árbol de menú del Lotus 1-2-3, se han

aceptada como protegibles. Quizá el concepto más confuso es la separación entre ideas (no protegibles) y expresiones (protegibles). Las generaciones de jueces y los abogados han luchado con el punto; ellos acuerdan únicamente que no hay *"ninguna línea de dirección nítida"* entre idea y expresión, y que la distinción deberá decidirse en cada caso. La mayoría de los tratadistas de información acordaron que la idea de trabajar sobre múltiples documentos a la vez desplegados en múltiples ventanas simultáneamente no es protegible, pero que las expresiones específicas de ventanas (las decoraciones de bordes, las animaciones para el movimiento, y más) podrán ser protegibles.

¿Es la protección de propiedad o las patentes más apropiadas para la interfaz de usuario? Tradicionalmente, la protección de propiedad se utiliza para las expresiones artísticas, literarias, y musicales, considerando que la patente se utiliza para dispositivos funcionales. Hay interesante crossovers, tales como propiedades para topografías, dibujo ingenieril, y decoraciones sobre tasas de té, y patentes para algoritmos de programa de computación. Las propiedades son fáciles de obtener (solo ponga un aviso de propiedad en la interfaz de usuario y un archivo de propiedad a la aplicación), son rápidas, y no son comprobadas. Las patentes son complejas, lentas, y costosas para obtener, ya que estas deben ser comprobadas por la Oficina de Marca Registrada y Patentes.

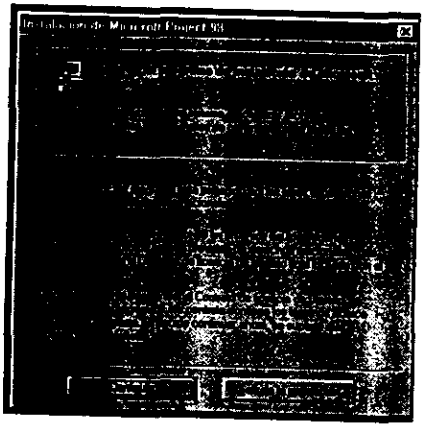
¿Qué constituye transgresión de la protección de propiedad? Si otro desarrollador copia exactamente su propiedad registrada de interfaz de usuario, esto es claramente un caso de transgresión. Puntos más sutiles surgen cuando un competidor hace una interfaz



de usuario que tiene elementos notablemente similares, a tu juicio que son propios. Para ganar un caso de transgresión de propiedad, usted deberá convencer a un jurado de "observadores ordinarios" que el competidor realmente vio su interfaz y que la otra interfaz es "considerablemente similar" a la suya.

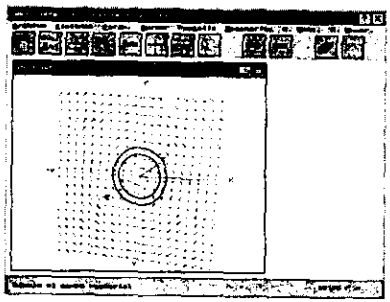
¿Debe la interfaz de usuario ser protegida de propiedad? Hay varios locutores respetados quienes creen que la interfaz de usuario no debería ser protegida de propiedad. Ellos pelean que esa interfaz de usuario debería compartirse y que el ser protegida de propiedad impide el progreso, si los desarrolladores tuvieran que pagar para el permiso para

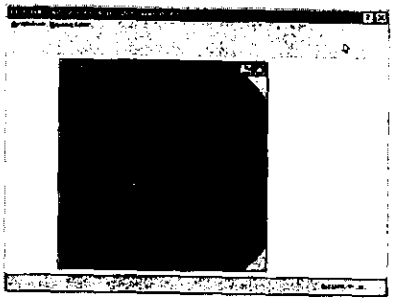
cada una de las características de interfaz del usuario que ellos vieron y quisieran incluir en su interfaz. Ellos sostienen también que la protección de propiedad interfiere con los beneficios de la estandarización y que las variaciones artísticas innecesarias crearían confusión e inconsistencia. Por otra parte los defensores de la protección de propiedad para la interfaz de usuario desean reconocer realizaciones creativas y permitir la protección, para fomentar la innovación mientras por otra parte asegurar que los diseñadores son premiados por sus trabajos. Aunque las que ideas no sean protegibles, las expresiones específicas tendrían que ser licenciadas desde el creador, presumiblemente por un costo, del mismo modo que cada foto en un libro de arte debe ser licenciado y reconocido, o cada uso de una canción, tocada, o cada cita se debe otorgar permiso. Preocuparse sobre la complejidad y costo de este proceso y la renuencia de los dueños de la propiedad protegida para compartir es legítimo, pero la alternativa de no proveer protección podría demorar la innovación.

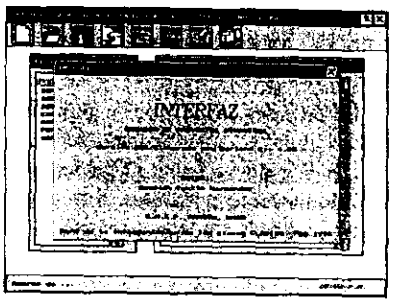


CAPITULO IV

DISEÑO DE UNA INTERFACE USUARIO GRAFICA







ASPECTOS DEL DISEÑO

RESUMEN El desarrollo de interfaces gráficas humano-computadora con el empleo de funciones gráficas de bajo nivel como BGI (Borland Grafic Interface) implica el desarrollo en el lenguaje C de un volumen importante de código, relativamente complejo, debido al manejo necesario de estructuras de datos, memoria dinámica, apuntadores y apuntadores a apuntadores, etc. Con el objetivo de facilitar y agilizar el desarrollo de este tipo de interfaces bajo MS-DOS, encapsulamos las BGI en el lenguaje orientado a objetos C++. Creamos diversos Widgets (objetos Gráficos) que están dentro de un conjunto de clases para interfaces gráficas, proporcionando así un nivel de abstracción superior para el programador. El resultado es un conjunto de clases denominado **CLADIUG (Clases para el Diseño de Interfaces Usuario Gráficas)** flexible y fácilmente reutilizable. En este Capitulo presentamos los aspectos más relevantes del diseño de la librería CLADIUG.

ANTECEDENTES O PREAMBULO

HERRAMIENTAS DE PROGRAMACIÓN

Programadores experimentados tienen que construir interfaces de usuarios con lenguajes ensambladores de bajo nivel, programando en lenguajes de alto nivel (C, Pascal, BASIC, Ada, Etc), librerías de programa para interfaces de usuario, o mas avanzadas herramientas de programación llamadas (toolkits).

Los lenguajes de programación acompañados con librerías o funciones son familiares para programadores experimentados y proporcionan gran flexibilidad. Sin embargo, el esfuerzo para construir interfaces de usuario y sobre todo las interfaces gráficas con este tipo de herramientas es grande, se queda expuesto a potenciales errores y la falta de estandarización es alta, las revisiones consumen bastante tiempo y el mantenimiento es difícil. Las herramientas mas avanzadas de programación o sea los tolkits son diseñadas para proveer a los programadores con rutinas especialmente diseñadas que manejen widgets estándares tales como: ventanas, barras de desplazamiento, menús pull-down o pop-up, campos de entrada de datos y cajas de diálogos.

Los Toolkits pueden llegar a ser complejos, y los ambientes de programación para esos toolkits tales como el toolkit para desarrolladores Microsoft Windows 3.0, Apple Macintosh MacApp, y Unix X-Windows Tolkit (Xtk) requieren de meses de aprendizaje para que los programadores adquieran competencia. Aun así, la carga en crear aplicaciones es grande, y el mantenimiento es difícil. La ventaja es que el programador tiene un control extenso y gran flexibilidad en la creación de interfaces. Los toolkits se han vuelto populares entre los programadores, pero estos solo proveen soporte parcial para la consistencia, y los

administradores de proyecto y diseñadores todavía deberán depender fuertemente de la experiencia de los programadores.

MANEJADORES DE INTERFACES DE USUARIO (UIMSS)

En esta seccion tocaremos brevemente el enfoque generalizado de lo que debe cumplir el diseño de GUI's, asi como una vision hacia al futuro de que es, lo que se espera de las GUI's. **El Termino Sistema de Administración (o Manejador) de Interfaces de Usuario por sus siglas en ingles (user-interface management system (UIMS))** es usado para describir herramientas de software que permiten a los diseñadores crear una completa y funcional interface de usuario sin tener que programar en los tradicionales lenguajes. La idea principal en UIMS es la independencia de la interface de usuario (o independencia del dialogo), paralelamente a la idea de la independencia de los datos en un manejador de base datos (DBMS). En ambos casos, la meta es separar la interface de usuario o lógica de diseño de los aspectos fundamentales de la implementación. En los DBMSs, los diseñadores de información no necesitan conocer las operaciones fisicas de la manipulación de archivos (manejo de índices, algoritmos de ordenamiento y resolución de colisiones, manipulación de punteros, marcado para borrar, etc.) para crear sistemas de información. Los cambios fisicos a la organización de los archivos, optimización de algoritmos, seguridad, respaldos y otros aspectos computacionales no deberán afectar el diseño de la información. Los diseñadores de información pueden hacer ciertos cambios en la presentación de la información sin dañar el almacenaje interno, y las transformaciones son manejadas automáticamente. Similar independencia de acción es lograda con UIMSs

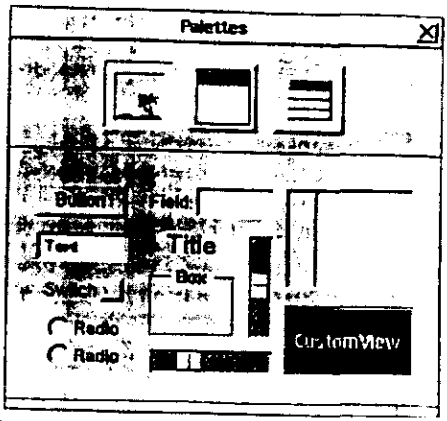


Figura 4-1 Paleta de Controles (Widgets) del constructor de Interfaces deUsuarioNeXT que pueden ser arrastrados para construir interfaces

La mayor ventaja de los UIMSs es que el proceso de desarrollo puede ser mas abierto, mas evidente y comprensible por un amplio rango de personas, haciendo cambios fácil-

mente como un resultado de hacer pruebas de uso. Al mismo tiempo el enfoque de los UIMSs ayuda en asegurar mayor consistencia en el diseño por usar *widgets estándares*, la distribución puede ser hecha uniformemente y la terminología puede ser controlada. La actual dirección en aplicaciones y en las herramientas UIMSs es ciertamente hacia las interfaces usuario gráficas con manipulación directa (Myers,1988). Un ejemplo de los modernos UIMSs para estaciones de trabajo incluyen el constructor de interfaces usuario gráficas NeXT's el cual tiene paletas de controles (widgets) que incluyen barras de desplazamiento, radio botones, cajas de verificación, campos de entrada de texto, etiquetas, listas desplazables y botones como en la figura 4-1. Una ves que los controles has sido seleccionados y arrastrados a su ubicación, el diseñador puede escoger atributos o propiedades para cada control desde un panel inspector ver Figura 4-2

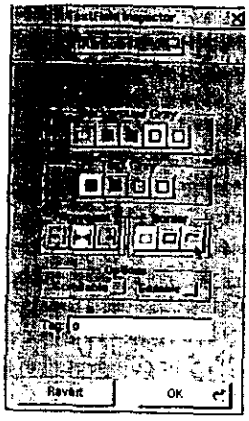


Figura 4-2 Panel inspector para poner atributos a los elementos de Interface de Usuario del constructor de interfaces NeXT

DESCRIPCIÓN DEL PROBLEMA.

Como ya se menciono en el capitulo 3 el éxito de un sistema computacional, es decir, su aplicación para la solución de problemas reales, radica en gran medida en la calidad de su interfaz con el usuario, la cual debe ser capaz de proporcionar mecanismos para una explotación fácil y eficiente del sistema.

El software de interfaz es inherentemente difícil de escribir debido entre otras cosas a las variaciones en los requerimientos de diversas aplicaciones, a las variaciones en las preferencias de los usuarios y a la necesidad de ofrecer portabilidad entre diferentes plataformas y sistemas operativos. Por lo anterior, desde hace algunos años se ha venido dedicando cada vez más tiempo y esfuerzo en el análisis, diseño e implementación de interfaces eficaces, lo cual se ha convertido hoy en día en una interesante y relevante área de investigación y desarrollo.

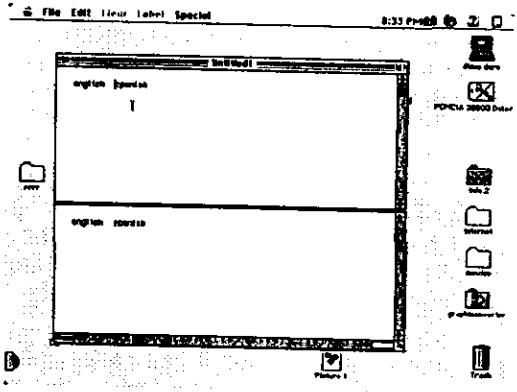


Figura 4-3 La GUI de Mac del S.O. System7

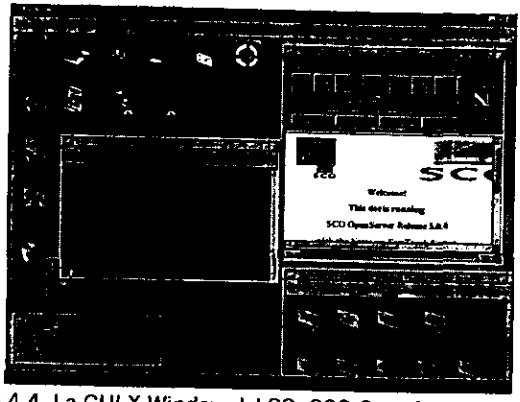


Figura 4-4 La GUI X-Window del SO. SCO OpenServer

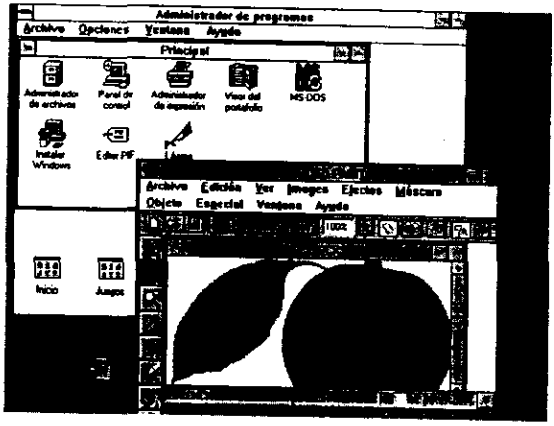


Figura 4-5 La GUI Windows3.11

En la actualidad se cuenta con varias interfaces gráficas e innumerables guias y herramientas que facilitan el desarrollo de interfaces humano-computadora en especifico nos referimos a las interfaces humano-computadora, llamadas Interfaces Usuario Gráficas (GUI). Dentro de las interfaces gráficas más destacadas tenemos a las siguientes: la de Windows, a la de Mac y a la de X-Window o también denominada X11. Ver los ejemplos de GUI's de windows, mac, X-Window Figuras 4-3 a 4-7

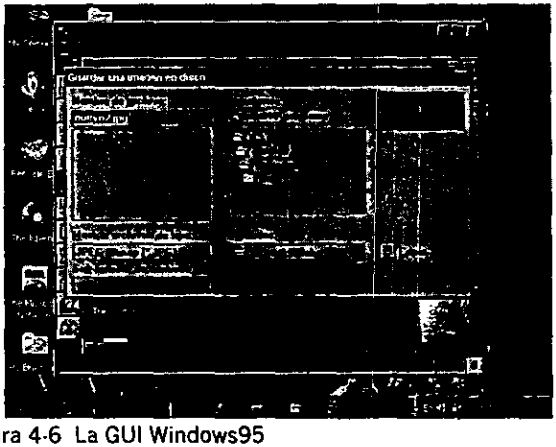


Figura 4-6 La GUI Windows95

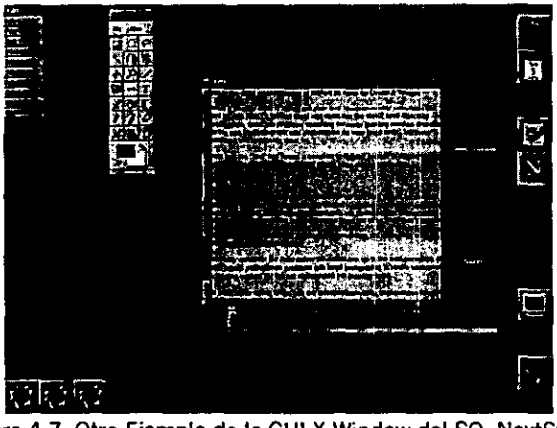


Figura 4-7 Otro Ejemplo de la GUI X-Window del SO. NextStep

El *surgimiento* de dichas interfaces gráficas trajo consigo la aparición de diversos paquetes de herramientas ("tolkits"), tales como OWL, MFC, Xview, DecWindows y *Motif*, por mencionar algunos. Estos toolkits proporcionan una capa de alto nivel en la construcción de bloques de interfaces con el usuario y pretende facilitar la programación

con respecto al nivel de *"bibliotecas de funciones de bajo nivel para gráficos"*, que resulta sumamente difícil y propenso a errores.

Sin embargo, a pesar de las facilidades ofrecidas por estos toolkits, no existe ningún toolkit para el desarrollo de GUI's bajo el sistema operativo MS-DOS, en cambio nuestra experiencia ha demostrado que la programación, al nivel de funciones gráficas básicas (específicamente en este caso las llamadas BGI), resulta aún tediosa y los volúmenes de código siguen siendo considerables. Por otra parte debido a que el lenguaje huésped en los diversos toolkits es C, es necesario invertir gran cantidad de tiempo en la programación de código confiable, reutilizable y mantenible.

OBJETIVOS

Tomando en cuenta lo anterior, y con propósito de facilitar el desarrollo de interfaces sin perder de vista la funcionalidad y apariencia que deben proporcionar y tener los widgets, se plantea como un objetivo mas, desarrollar una biblioteca en programación orientada a objetos, integrada por clases para crear objetos abstractos que encapsularan los widgets de interfaz, permitiendo con ello la creación de código de interfaz-humano computadora con las siguientes características:

- Mas Simple
- De Menor Volumen
- Fácilmente reutilizable
- Fácilmente Mantenible

Todas las anteriores características se deben comparar con las ofrecidas por los toolkits como OWL (Object Window Libraries) o con las MFC (Microsoft Fundation Class) que te permiten el manejo de la interfaz gráfica pero solo para el ambiente windows; sin embargo todavía hay muchas maquinas con sistema operativo DOS.

En la siguiente sección se describe la metodología que se siguió para la consecución de los objetivos propuestos.

Esta Metodología puede resumirse en los cuatro pasos siguientes:

1.- Elección de los objetos de interfaz (widgets).

2.- Definición de las propiedades de los objetos.

3.- Clasificación de los objetos

4.- Implementación.

Se determinaron los objetos de interés para interfaces orientadas básicamente a sistemas de información y Base de Datos. De manera general para este tipo de aplicaciones, se puede mencionar que se requieren objetos tales como contenedores, menús, botones de funciones, formas, campos de entrada y/o salida de información, etiquetas, áreas de

mensaje, cajas de advertencia y dialogo, encabezados, botones de selección múltiple, lista de selección, ventanas de texto y gráficos entre otros.

DEFINICIÓN DE LAS PROPIEDADES DE LOS OBJETOS.

Para cada objeto seleccionado fue necesario determinar su comportamiento y sus caracteristicas básicas (propiedades). La apariencia de un objeto se define por un conjunto de acciones que es posible realizar sobre él. El comportamiento de un objeto esta definido por los eventos a los cuales es sensible y por la forma en que responde a dichos eventos. Estos objetos proporcionan un conjunto de propiedades definidas para cada objeto gráfico (widget), las cuales son heredadas a través de una jerarquía definida por un árbol de tipos de Widgets.

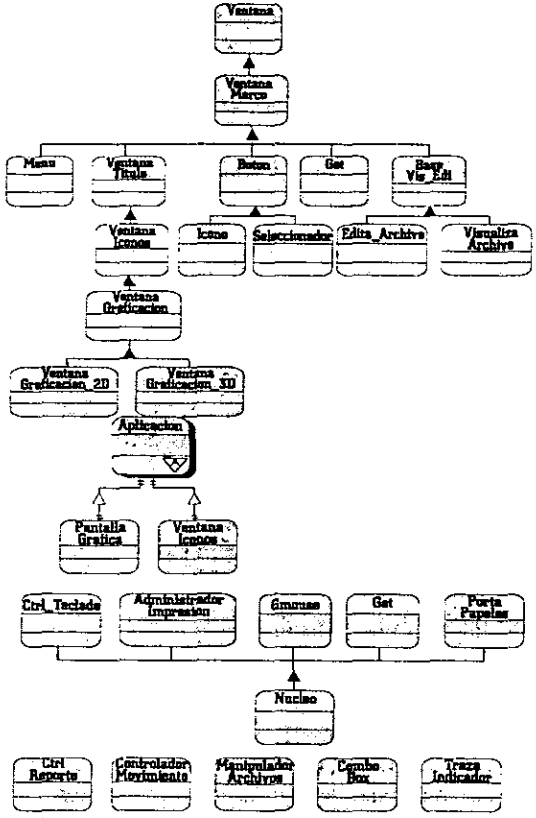


Figura 4-8 Jerarquia de clases CLADIUG

Las características de los objetos se determinaron partiendo de lo general a lo particular, esto es, primero se definieron aquellas características que son comunes a todos los objetos y posteriormente se determinaron aquellas especificas al tipo de objeto. Las acciones que se pueden tomar de manera común para todos los objetos permiten definir por ejemplo:

- Tamaño
- Coordenadas iniciales
- Tipo de posicionamiento (absoluto, con respecto al dispositivo de despliegue gráfico, o relativo con respectivo a otro widget)
- Visibilidad (visible u oculto).

Como ya se menciono, existen propiedades especificas para cada tipo de objeto. Aquí se destacaran únicamente aquellas definidas para los objetos de tipo campo de entrada/salida de información y de tipo botón función. A los objetos tipo campo, además de las definidas intrisicamente, se agrego el siguiente conjunto de propiedades: longitud de campo, tipo de campo, editable. El tipo de campo se implemento haciendo uso del mecanismo de "Callbacks". Se implementaron callbacks para validar entradas de tipo texto, entero y flotante.

Los botones son objetos especiales que además de sus atributos asociados tienen la capacidad de llevar acabo alguna acción cuando ocurre un evento sobre ellos. Estas acciones asociadas definen su comportamiento a nivel aplicación, de tal forma que se pueden programar diferentes tipos de botones, de acuerdo a la acción que ejecuten. La implementación de tales acciones se lleva a cabo mediante la asociación de funciones o "Callbacks" que el diseñador de una interfaz humano-computadora puede programar de este modo, se pueden definir tantos tipos de botones como requiere una aplicación.

CLASIFICACIÓN DE LOS OBJETOS.

Una vez elegidos los objetos gráficos y sus propiedades se procedió a clasificarlos mediante una jerarquía de clases que los agrupara de acuerdo a su funcionalidad

IMPLEMENTACIÓN

La implementación de las clases se realizo mediante el encapsulamiento de widgets utilizando el lenguaje orientado a objetos C+ + y como resultado se obtuvo, una biblioteca de clases que proporcionan mecanismos para la creación y manipulación de objetos de interfaz, esta biblioteca fue diseñada pensando en poder llegar a convertirse en una biblioteca de componentes reusables. Escogimos el lenguaje y medio ambiente de desarrollo de C++ por razones cuya explicación detallada rebasa el alcance de

este capitulo, sin embargo podemos mencionar que entre dichas razones destacan que C + + mantiene compatibilidad con el lenguaje C estructurado convencional, es un lenguaje con fuerte control de tipos de datos y de ligas dinámicas polimorficas, además de su gran portabilidad para varias plataformas, entre otras características, lo que da una robustez superior que facilita la construcción de sistemas eficaces, correctos y robustos.

La figura 4-8 representa el modelo conceptual de la jerarquía de clases correspondientes a abstracciones con las que se puede implementar una interfaz gráfica humano-computadora.

Nuestro núcleo es un tipo general de Widget que usa la herencia multiple, el cual administra los recursos como: teclado, mouse, portapapeles, ventana de mensajes e impresión, el control del núcleo lo tiene la clase aplicacion que es una clase abstracta por lo tanto el usario no la contruye, ni la destruye, ni la contola solo pude hacer uso de sus recursos, que es la ventana principal en una interfaz o sea que es el Widget raíz en la jerarquía interna de Widget . Para ejecutar alguna acción en una aplicación, el usuario oprime un botón que es (un objeto de tipo botón)

Actualmente estamos usando la biblioteca para el desarrollo de una interfaz de una aplicación tipo científica.

Para la implementación de aplicaciones bajo una arquitectura cliente/servidor , en la que el cliente implementa la funcionalidad de interacción con el usuario y se comunica con uno o más servidores, posiblemente localizados en otras maquinas interconectadas en una red, se necesita desarrollar una abstracción para la comunicación entre procesos, esta abstracción debería ser implementada en una o más clases escritas en C++ que encapsulen un socket similar a los usados en Unix empleando el protocolo de comunicaciones TCP/IP.

CONCLUSION.

Su utilización queda abierta a cualquier interfaz Humano-Computadora (GUI) que necesite objetos gráficos como base para su implementación . Ya que la implementación mediante la tecnología orientada a objetos permite la reutilización del código para realizar mejoras al (GUI) actual, ya sea extendiendo las propiedades de las clases existentes o ampliando la clasificación con otros tipos de objetos requeridos en otro tipo de aplicaciones. Esto se realiza con relativa facilidad en el marco de la Programación Orientada a Objetos a través de los mecanismos de herencia, agregación, redefinición y polimorfismo inherentes al paradigma y lenguaje empleado en el presente trabajo.

DISEÑO DE LA LIBRERIA CLADIUG

A continuación se presenta las partes más relevantes de diseño de la libreria CLA-DIUG así como la aplicación de la metodología del capitulo II en diseño de la libreria

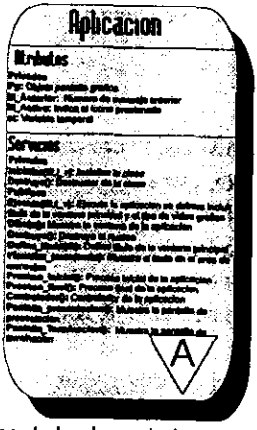


Figura 4-9 Diagrama de la clase abstracta Aplicación

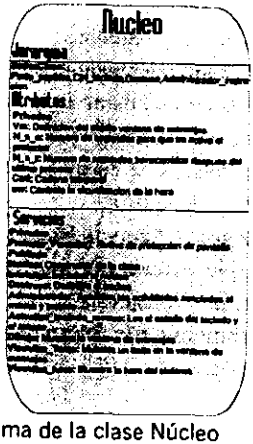


Figura 4-10 Diagrama de la clase Núcleo

Clase: Ventana
Visibilidad: Exportada
Jerarquia:
Documentacion: La clase Ventana permite definir una ventana de trabajo, visualizar texto dentro de ella, cambiar el tamaño y tipo de la letra a visualizar, mover esta, cambiar el tamaño, grabar la imagen existenteantes de abrir la ventana y activar la ventana.

Atributos:
Privados:
C_F_A: Almacena el color del fondo que se uso anteriormente
C_T_A: Almacena el color del texto que se uso anteriormente
C_REF: Indica el color con el que se limpiara la ventana al salir
ARCH_TMP: Guarda elnombre del archivo que se usara para grabar la pantalla
ARCH_TMP1: Nombre del archivo temporal
xcad: Cadena temporal para la visualización de texto
Protegidos:
A_CAR,L_CAR: Ancho y largo del carácter a visualizar
N_car_visualizar: Nùmero de caracteres a visualizar
N_ren_visualiza: Número de renglones a visualizar
X1,Y1,X2,Y2: Limites de la ventana
C_VENT: Almacena el color de la ventana
C_TEXT: Almacena el color del texto a visualizar
C_FOND: Almacena el color del fondo del texto a vizualizar
TLt_tipo: Almacena el tipo de letra actual dentr de la ventana
TLt_mulx,TLt_muly: Almacena la escala en X y Y
Inicia_X,Inicia_Y: Indica donde iniciara la visualización dentro de la ventana
TLt_hori: Almacena la orientación de la letra
SALVAR_PANTALLA: Indica si se grabara la pantalla
VENTANA_VISIBLE: Indica si la ventana esta visible o no
VENTANA_ACTIVA: Indica si la ventana esta activa
MOUSE_DENTRO: Indica si el mouse esta dentro de la venta o no
St_mouse: Indica el estado del mouse

Servicios
Privados:
Inicializar_var(): Inicializa las variables de la ventana

Figura 4-11 Parte de la platilla para la Clase Ventana

Public:

Ventana(),~Ventana: Rutinas constructura y destructora de la clase ventana
Define_ventana(x1,y1,x2,y2,cl): Define la ventana y su color
Ventana_centrada(lg_x,lg_y,cl,inc_x,inc_y): Define una ventana centrada de de longitud y color dados
Centra_texto(y,texto): Centra texto dentro de la ventana
Visualiza_texto(x,y,texto,t_v): Visualiza un texto dentro de la ventana
Visualiza_caracter(x,y,cl,t_v): Visualiza un caracter dentro de la ventana
Activa_graba_ventana(tp): Activa grabar la ventana
Activa_color_refresco(col): Indica el color de la ventana al cerrar
Actual_pos_ventana(x1,y1,x2,y2): Regresa la actual posición de la ventana
Mover_ventana(x,y): Mueve la ventana
Mover_ventana_n_pixeles(x,y): Mueve la ventana actual una determinada cantidad de pixeles
Cambiar_tamano(x1,y1,x2,y2): Cambia de tamaño de la ventana
Largo_X(): Retorna el largo en X de la ventana
Largo_Y(): Retorna el largo en Y de la ventana
Largo_caracter(): Retorna el largo del carácter actual
Ancho_caracter(): Retorna el ancho del carácter actual
Mouse_dentro(): Indica si el mouse esta dentro de la ventana
Define_color_fondo(cl): Define el color de fondo al visualizar texto
Define_color de texto(cl): Define el color de texto a visializar
Define_color(cfd,ctx): Define el color del fondo y del texto a visualizar
Define_color_ventana(cl): Define el color de la ventana
Actual_color_texto(): Indica el actual color del texto
Actual_color_fondo(): Indica el actual color de fondo de texto
Actual_color_ventana(): Indica el color actual de la ventana
Tamano_letra(lg_x,lg_y): Define el tamaño de letra a visualizar en la ventana
Define_tipo_letra(tipo,horient,multx,divx,multy,divy): Define el tipo de letra, orientación y tamaño
Limpia_Recuadro(x1,y1,x2,y2,cl): Limpia la porcion de la ventana con un color dado
Oculta_ventana(tp): Oculta la ventana o la revisualiza
Ventana_activa(tp): Activa o Desactiva la ventana
Estado() Retorna el estado de la ventana (Activo o no)
Retorna_inicio_X(): Retorna el inicio del área de trabajo en X
Retorna_inicio_Y(): Retorna el inicio del área de trabjo en Y
Graba_ventana(arch,tp): Graba la actual ventana, si tp es (0) imagen con formato (1)imagen sin formato

Figura 4.12 Continuación de la plantilla de la clase ventana

Lee_ventana(arch,tp): Lee una imagen a la *ventan actual si tp* es (0) Imagen con formato (1) imagen sin formato
Posicion_mouse(x,y): Retorna la posición de l mouse dentro de la *ventana*
Presionado(boton): Revisa si el boton es oprimido por el mouse dentro de la ventana

Figura 4-13 Fin de la plantilla de la Clase Ventana

Clase: Ventana_Marco
Visibilidad: Importada
Jerarquia: superclase Ventana
Documentacion: La clase Ventana permite *definir una ventana de trabajo* con marco el cual se activa si el mouse esta sobre la ventana, .paermite hacer todas las actividades de la clase ventana y cambiar el *color y forma del marco*

Atributos:
Privados:
C_rec_inf: Color del recuadro inferior
C_rec_sup: Color del recuadro superior
C_base_a: *Color* de la base de la ventana activada
C_base_n: Color de la base de la ventana no activa
Mouse_dentro_Ventana: Indica si el mouse esta *dentro de la ventana*

Servicios
Publicos:
Ventana_marco(): Constructor de la ventana con marco
Marco_ventana(tb): Dibuja el marco de la ventana según el tb
***Define_colores*(c_r_i,c_r_s,c_b_a,c_b_n):** Define los colores de la ventana con marco
Mouse_dentro() : Revisa si el *mouse esta* dentro de la ventana con marco
Dibuja(): Dibuja la ventan con marco

Figura 4-14 Segundo ejemplo del uso dela plntilla para la Clase Ventana_marco

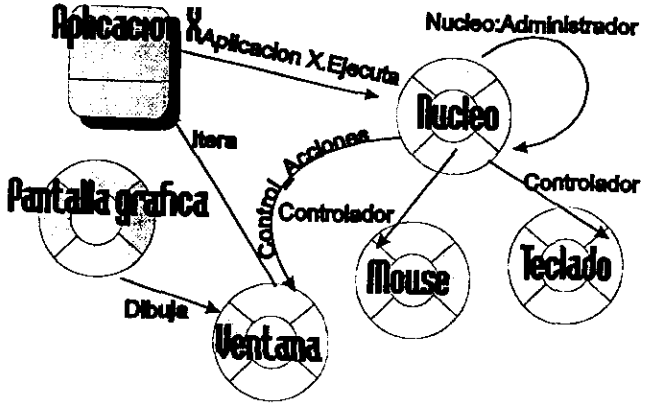


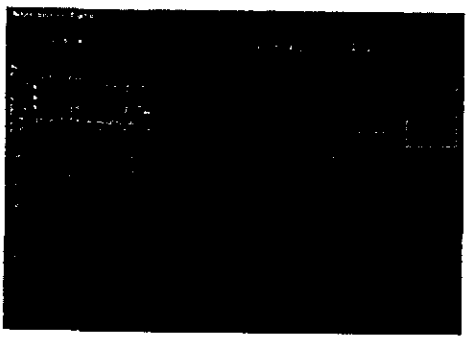
Figura 4-15 Porción de un Diagrama de Objetos de una Aplicación

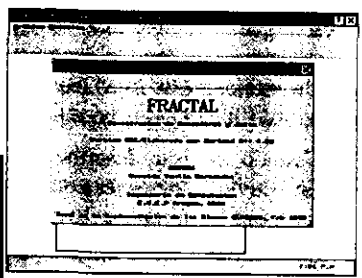
Proceso Inicial
Pantalla de presentacion
Controlador
Ejecucion de Proceso
Pantalla de terminacion
Proceso Final

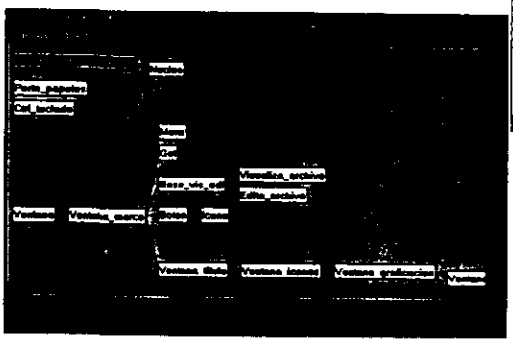
Figura 4-16 Escenario para cualquier Aplicación usando la libreria CLADIUG

CAPITULO V

IMPLEMENTACION DE UNA INTERFACE USUARIO GRAFICA







COMPROMISOS ENTRE CONTRATADO Y CONTRATISTA

Podemos considerar que programar bajo el modelo de objetos consiste en

- Programar clases
- Crear objetos a partir de las clases
- Enviar mensajes a los objetos (llamar a funciones de las clases a través de los objetos)

El programador puede situarse en dos puntos de vista con respecto a una clase:

Como cliente o heredero de la clase. En este caso humildemente y de buen corazón confía en que la clase que va utilizar, o de la cual va a heredar, sea robusta y este bien instrumentada. Es decir, supone que esta no le generara errores si él cumple los requisitos que la clase exige para su uso.

Como instrumentador de la clase. En este caso autosuficientemente considera que desarrollara una clase correcta, que no le traerá errores a sus usuarios si estos cumplen disciplinadamente con los requisitos de utilización de la misma.

Una clase por lo general juega estos dos roles: cliente de unas clase y servidor de otras clases, estableciéndose durante la ejecución de un sistema un proceso en espiral que tiene dos extremos:

Cuando tenemos clases (y también funciones en una concepción híbrida como la de C++) ofrecidas en una biblioteca construida por el propio constructor del compilador (incluso algunas de ellas inmersas dentro del propio lenguaje como las de los tipos básicos y sus operaciones). Para el programador estas clases son servidores totales (aunque por supuesto internamente estas clases pueden utilizarse entre sí). En esta caso la confianza debe ser máxima (aunque no absoluta porque ningún compilador esta exento de fallas).

Cuando la clase describe al objeto raíz (objeto cuya creación desencadenara la creación y envío de mensajes a otros objetos) como es el caso de Eiffel; o la función principal *main (quien es la que crea e invoca los servicios de otros objetos)* como es el caso de C++. Este seria el cliente total (también denominado actor)

El éxito de un sistema (desde el punto de vista de programación) se basa, en que durante la programación cada parte cumpla su compromiso en este proceso según el rol que este desempeñando. El programador de una clase debe lograr que la clase cumpla lo que promete y a su vez cumplir con lo que le piden las clases que la utilizan.

Es tipo de programación se podría ver como la metáfora del contrato. La metáfora de la programación por contratos consiste en contratos que deben cumplir los objetos en tiempo de ejecución. Entre un objeto contratista que es el cliente de otro objeto contratado por que le envía un mensaje (le pide que aplique una función miembro de este) se establece un contrato que determina:

Una precondición que es el requisito que debe cumplir el contratista para poder aplicar una función (solicitarle un servicio) del contrato.

Una postcondición que es la garantía que el contratado le da al contratista si este cumple con la precondición

El invariante que expresa la integridad del contratista y lo que debe cumplirse en todo momento de su existencia.

CONFIGURACION DEL COMPILADOR.

Para poder compilar cualquier proyecto que use la librería CLADIUG se deberá hacer las siguientes modificaciones en el menu *Options Proyect:*

- Directories:
  - **Include :** agregar la ruta en donde están todos los fuentes de CLADIUG por ejemplo: c:\bc4;c\milib
  - **Library:** Poner la ruta donde se encuentra la librería CLADIUG precompilada si se usa esta. Por ejemplo: c:\bc4;c:\librería
  - **Intermediate:** Esto es opcional. Poner la ruta donde se crearan todos los objetos. Por ejemplo: c:\objs
- En el topico Compiler:
  - **Defines:** Poner _ IDIOMA_ESPANOL_
  - **Code Generation:** Encender o palomear las opciones
  - *Unsgined characters*
  - Duplicate strings merged
  - Debugging: Encender o palomear las opciones
  - Test stack overfolw
  - Precompiled Headers:
  - Do not generate or use
- *En el topico* 16-bit Compiler:
  - ***Processor:*** Verificar que se este usando el instruction set: 80386
  - **Memory Model Override:** usar *Large y en Options* Encender o *palomear la opción* Put constant strings in code segments
- En el topico *Optimizations:*
  - **Specific:** *Encender o palomear las opciónes:*
  - Executable size

- Optimize globally
- Speed: Encender o palomear todas las opciones.

• En el topico Linker:

- General: Apagar la opción Case sensitive exports and import (16-bit only).

CONFIGURACION DEL PROYECTO.

Existen dos formas de poder configurar el proyecto.

1. Usando la librería precompilada CLADIUG, la cual solo se tendrá que incluirse en el proyecto (como se muestra en la figura 5-1) para poder usar todas las

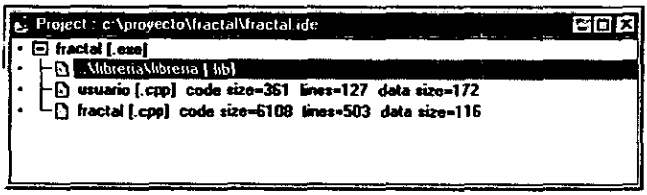


Figura 5-1 Ejemplo de como configurar un proyecto usando la libreria CLADIUG precompilada

caracterirsticas.

2. Usando los fuentes de CLADIUG, los cuales se tendran que incluir todos en el proyecto.

CODIFICACION

A continuación se mostrara algunas partes relevantes de la implementación de la librería CLADIUG:

```
#ifndef __NUCLEO_HPP__
#define __NUCLEO_HPP__
#include "mouse.hpp"
#include "v_marco.hpp"
#include "porta_pa.hpp"
#include "ctrl_tec.hpp"
#include "admin_ip.hpp"
#include "teclado.hpp"
extern int Programa_activo; // Indica si el programa esta o no activo
extern int Tecla; // Tecla actual del bufer
extern char Caracter; // Caracter actual del bufer
 Clase que funge como núcleo del sistema, administra los siguientes recursos:
Teclado
Mouse
Porta papeles
Ventana de mensajes Impresión

 De una manera organizada y completa para poder ser
 compartidos por toda clase que lo requiera

class Nucleo: public Porta_papeles, public Ctrl_teclado, public GMouse, public Administrador_impresion{
private:
   Ventana_marco   *Vm; // Definición del objeto ventana de mensajes
   long  N_s_e;  // Numero de segundos para que se active el protector de pantalla
   long  N_s_t;  // Numero de segundos transcurridos después del ultimo proceso
   char *cad;  // Cadena temporal
   long  sw;  // Variable temporal para el control de la visualización de la hora
   void Protector_pantalla(void); // Rutina que hace de protector de pantalla

public:
////////////////////////////////////////////
// Rutinas de control del núcleo  //
////////////////////////////////////////////
   Nucleo(void); // Constructor de la clasevoid Inicializa(void); // Inicializa el núcleo
   void Destruye(void); // destruye el núcleo
   void Administrador(void); // Actualiza el teclado, mouse y las actividades asociadas con
   estos
   void Actualiza_teclado_mouse(void) // Lee el estado de teclado y el mouse
   {Retorna_bufer_teclado(Caracter,Tecla); Mpos();if(Tecla == ALT_Q) Programa_acti-
   vo = 0;}
   void Dibuja(void);  // Dibuja la ventana de mensajes
////////////////////////////////////////////
// Rutinas de control de la ventana de mensajes  //
////////////////////////////////////////////
   void Visualiza_texto(const char *texto); // Rutina que visualiza el texto en la ventana
   de mensajes
   void Visualiza_hora(void); // Rutina que visualiza la hora del sistema
};
```

```
#endif

////////////////////////////////////////////////////////////
 El control de esta clase lo tiene la clase APLICACION el usuario no la
  construye, ni destruye, ni la controla, solo puede hacer uso de sus recursos

 Son visibles todos los comportamientos públicos de las clases
   A) Teclado
   B) Mouse
   C) Porta papeles
   D) Administrador de impresión

+ Para Imprimir un archivo por medio del administrador de impresión
  requiere estar grabado en disco, entonces especifique:
     NCO-Parametros_Administrador_impresion(const char *arch);
  indicando el nombre del archivo a imprimir el núcleo se encargara
  de imprimirlo.
+ Para hacer uso del porta papeles haga lo siguiente:
    a) Indique la cantidad de líneas a mandar al porta papeles
         NCO-Crea_porta_papeles(unsigned int num_elem);
    b) Inserte cada una de las cadenas al portapapeles
         NCO-Inserta_porta_papeles(const char *cad);
    c) Para saber si esta activo el porta papeles
         NCO-Retorna_Estado_porta_papeles(void);
    d) Para saber el numero de elementos en el porta papeles
         NCO-Retorna_total_elementos(void);
    e) Para recuperar una cadena especifica del porta papeles
        const char   NCO-Retorna_cadena_porta_papeles(unsigned int num_elem);
+ Para visualizar un mensaje en la ventana de visualización de mensajes
  escriba este en el siguiente comportamiento
     NCO-Visualiza_texto(const char *texto);
*/


ifndef __APLICAC_HPP__
#define __APLICAC_HPP__

#include "pantgraf.hpp"
#include "v_iconos.hpp"
#include "teclado.hpp"
#include "vis_ayud.hpp"
#include "s_arch.hpp"
#include "r_esp.hpp"
// Clase prototipo de aplicaciones
class Aplicacion{
private:
    Pantalla_Grafica *Pg;    // Declaración del objeto Pantalla Gráfica
    unsigned int     M_Anterior; // Numero de mensaje anterior
```

```
    unsigned int     M_Activo; // Numero de mensaje actual
     int  st;    // Variables temporales
    char  *cad;
    void Inicializa(const char *tit, const int t_v);  // Inicializador de la clase
    void Destruye(void); // Destructor de la clase
protected:
    Ventana_iconos  *Vt;    // Declaración del objeto ventana de trabajo
    char  **Texto_explicativo;   // Texto explicativo
public:
    void Ejecuta(const char *tit, const int t_v); // Ejecuta la Aplicacion se deberá de incluir
    // el titulo de la ventana principal y el tipode vídeo gráfico:
    // (4) SVGA 1024x768 256 colores
    // (3) SVGA  800x600 256 colores
    // (2) SVGA  640x480 256 colores
    // (0) VGA   640x480  16 colores
    void Dibuja(void); // Dibuja la ventana de la Aplicación
    void Define_titulo(const char *tit) {Vt-Define_titulo(tit);} // Define el titulo de la
    ventana principal
    void Visualiza_texto(const char *texto); // Visualizar la cadena en el arrea de mensajes
    virtual void Proceso_inicial(void) = 0; // Proceso inicial de la Aplicacion
    virtual void Proceso_final(void) = 0; // Proceso final de la aplicación
    virtual void Controlador(void) = 0; // Controlador de la aplicación
    virtual void Pantalla_presentacion(void) = 0; // Pantalla de presentación
    virtual void Pantalla_terminacion(void) = 0; // Pantalla de terminación
};
#endif
/*
//////////////////////////////////////////////
// Modo de uso de la clase Aplicacion  //
//////////////////////////////////////////////
#include aplicac.hpp"
// Clase de la Instala
class Instala: public Aplicacion {
private:
    Icono *Iconos;    // Puntero a los iconos
    int      Niconos;     // Indica el numero de iconos activos
    int      Ipresionado;  // Indica el icono presionado
public:
    void Proceso_inicial(void);  // Proceso inicial del programa
    void Proceso_final(void); // Proceso final del programa
    void Controlador(void); // Controlador de la Aplicacion
    void Pantalla_presentacion(void); // Pantalla de presentación
    void Pantalla_terminacion(void); // Pantalla de terminación
    void Dibuja(void); // Dibuja la ventana de la Aplicacion
};
#include "nucleo.hpp"
extern Nucleo *NCO; // Definición externa del objeto Núcleo
```

```
extern int Programa_activo; // Indica si el programa esta o no activo
extern int Tecla; // Tecla actual del bufer
extern char Caracter; // Caracter actual del bufer
extern int X_MAX; // Dimensiones de la pantalla
extern int Y_MAX;
extern char *ARCHIVO_AYUDA; // Indica el archivo que se mostrara de ayuda
extern unsigned int N_texto_explicativo; // Indica cual texto explicativo a visualizar
void Instala::Proceso_inicial(void)
{
ARCHIVO_AYUDA = "C:\\INTEGRA\\SYS\\HELP\\INTEGRA.HLP"; // Indica el ar-
chivo de ayuda
Texto_explicativo = new char *[3];
Texto_explicativo[0] = "";
Texto_explicativo[1] = "Terminar el programa y retornar al DOS";
Texto_explicativo[2] = "Visualiza la ayuda sobre el sistema";
Dibuja(); // Dibuja la pantalla de trabajo
}
void Instala::Proceso_final(void)
{
}
void Instala::Controlador()
{
}
void Instala::Dibuja(void) // Dibuja la pantalla de trabajo
{
Aplicacion::Dibuja(); // Dibuja la ventana con iconos
}
void Instala::Pantalla_presentacion(void) // Presentación inicial
{
}
void Instala::Pantalla_terminacion(void) // Pantalla de terminación
{
}
void main(int argc, char *argv[])// Programa principal de integra
{
  int svga = 0, st = 0;
// Revisa a que modo gráfico iniciara la pantalla Gráfica
if(argc == 2) {
 svga = argv[1][0] - '0';
if(svga 0 || svga  4 || svga == 1) st = 1;
} else st = 1;
if(st)printf("\n\n%s\n\n%s\n\n%sN\n\n%s\n%s\n%s\n%s\n%s\n",TXT1,TXT2,argv[0],
TXT3,TXT4,TXT5,TXT6,TXT7); else {
Instala Is;
Is.Ejecuta(TXT1,svga);
  }
}
*/
```

```
#ifndef __MOUSE_HPP__
#define __MOUSE_HPP__
#include "definic.hpp"
extern "C" {
#include h
}
#define SI  1
#define NO  0
#define MOUSE_REQUERIDO  1
#define MOUSE_NO_ACTIVO  2
#define MOUSE_OPCIONAL  3
#define BR  1
#define BM  3
#define BL  2
typedef struct {
  int present, buttons;
} Mresult;
typedef struct {
  int button, flag, x, y;
} Mstatus;
typedef struct {
  int x_count, y_count;
} Mmovement;
// Definición del mouse en el modo gráfico
typedef struct {
  unsigned int xkey, ykey, ScreenMask[16], CursorMask[16];
} Gcursor;
// Clase  MOUSE en modo gráfico
class GMouse {
private:
    unsigned  int  Driver_exists: 1;  // Esta el driver del mouse presente o no
    unsigned  int  Mtrabajando:  1; // Indica si en mouse esta trabajando
    unsigned  int  Mview:        1; // Esta el MOUSE visible o no
    C_dbl        Limites; // Indica los limites del mouse
    int  Mtype; // Indica el tipo de mouse a mostrarse
    union    REGS inreg, outreg;        // Estructuras para el manejo del MOUSE
    Mstatus      V_Mstatus;
    Mresult      V_Mresult;
    Mmovement    V_Mmovement;
    void  Mxlimits(int minx,int maxx); // Establece los limites para X
    void  Mylimits(int miny,int maxy); // Establece los limites para Y
protected:
    void  MInicializa(const int tp, const C_dbl lim, int tp_mouse); // Inicializa el mouse
    int   Test_driver_exists(void); // Revisa si existe el driver del mouse
public:
    Mmovement    *Mmotion(void);
```

```
    Mresult     *Mreset(void);
    Mstatus     Mpos(void);
    Mstatus     Mpressed(int button);
    Mstatus     Mreleased(int button);
    void Muestra_mouse(int showstat); // Indica si se visualiza el mouse o no
    void Mmoveto(int x, int y); // Permite posicionar el mouse en la posición X y Y
    void Establece_limites(C_dbl lim) // Establece nuevos limites para el mouse
    {Mxlimits(lim.x1,lim.x2);Mylimits(lim.y1,lim.y2);}
    void Restaura_limites(void) // Restablece los limites del mouse
    {Establece_limites(Limites);}
    void  Mmove_ratio(int xsize,int ysize);
    void  Mspeed(int speed);
    void  Mbox(int left,int top,int righ,int bottom);
    int   Ismoving(void); // Revisa si el mouse ha sido movido
    void Mlightpen(int set); // Rutinas validas solo para modo gráfico
    void Setcursor(const int tp); // Selecciona el tipo de cursor del mouse
    Mstatus   Retorna_estado_mouse(void) {return V_Mstatus;} // Retorna el estado del
    mouse Mresult   Retorna_Mresult(void) {return V_Mresult;} // Retorna el estado del
    mous
    Mmovement  Retorna_Mmovement(void)  {return V_Mmovement;} // Retorna el mo-
    vimiento del mouse
};
////////////////////////////////////////////////////////
// Clase que controla al porta papeles                //
// Permite inicializar y controlar el porta papeles   //
//////////////////////////////////////////////////////// ////
#ifndef __PORTA_PA_HPP__
#define __PORTA_PA_HPP__
class Porta_papeles {
private:
    char      **Informacion;      // Puntero a la Informacion
    char       *Cad_temp;         // Cadena temporal
    unsigned int  i;              // Variable temporal
    unsigned int  Total_elementos;  // Total elementos del portapapeles
    unsigned int  Actual_elemento;  // Actual elemento del portapapeles
    unsigned int  Activo:1;         // Indica si esta activo el portapapeles
protected:
    void  Inicializa_porta_papeles(void); // Inicializa el portapapeles
    void  Borra_porta_papeles(void); // Borra el portapapeles
public:
    void    Crea_porta_papeles(unsigned int num_elem); // Crea el portapapeles con
    NUM_ELEM elementos
    void   Inserta_porta_papeles(const char *cad); // Inserta en el portapapeles la cadena
    CAD const  char *Retorna_cadena_porta_papeles(unsigned int num_elem); // Retorna
    el contenido del portapapeles numero NUM_ELEM
    unsigned int    Retorna_total_elementos(void){return Total_elementos;} // Retorna el
    total de elementos del portapapeles
```

```
    unsigned int   Retorna_Actual_elemento(void) {return Actual_elemento;} // Retorna el
    actual elemento dentro del portapapeles
    unsigned int   Retorna_Estado_porta_papeles(void) {return Activo;} // Retorna el es-
    tado del portapapeles
};
#endif

#ifndef __V_BASICA_HPP__
#define __V_BASICA_HPP__
#include "colores.hpp"
#include "mouse.hpp"
#include "r_var.hpp"
#define NO          0
#define SI          1
#define CERRAR      2
#define ABRIR       3
#define CON_FORMATO  0
#define SIN_FORMATO  1
class Ventana {
private:
    char  C_F_A; // Almacena el color del fondo que se uso anteriormente
    char  C_T_A;  // Almacena el color del texto que se uso anteriormente
    char  C_REF; // Indica el color con el que se limpiara la ventana al salir
    char *ARCH_TMP; // Guarda el nombre del archivo que se usara para grabar la pantalla
    char *ARCH_TMP1;  // Nombre de archivo temporal
    char *xcad;  // Cadena temporal para visualización de texto
    void Inicializar_var(void); // Inicializa las variables de la ventana
protected:
    char A_CAR, L_CAR;  // Ancho y largo del carácter a visualizar
    char N_car_visualizar;  // Numero de caracteres a visualizar
    char N_ren_visualizar;  // Numero de renglones a visualizar
    int  X1,Y1,X2,Y2;  // Limites de la ventana actual
    char C_VENT; // Almacena el color de la ventana
    char C_TEXT; // Almacena el color del texto a visualizar
    char C_FOND;  // Almacena el color del fondo del texto a visualizar
    char TLt_tipo; // Almacena el tipo de Letra actual dentro del la ventana
    int  TLt_mulx; // Almacena la escala en X
    char TLt_divx;
    char TLt_muly; // Almacena la escala en Y
    char TLt_divy;
    char Inicia_X;   // Indica donde iniciara la visualización dentro de la ventana
    char Inicia_Y;  // Indica donde iniciara la visualización dentro de la ventana
    unsigned int TLt_hori:1;  // Almacena la orientación de la letra
    unsigned int SALVAR_PANTALLA: 1; // Indica si se grabara la pantalla
    unsigned int VENTANA_VISIBLE: 1; // Indica si la ventana esta visible o no
    unsigned int VENTANA_ACTIVA:  1; // Indica si la ventana esta activa
    unsigned int MOUSE_DENTRO:1;     // Indica si el mouse esta dentro de la ventana o
    no
```

```
    Mstatus  St_mouse;  // Indica el estado del mouse
public:
    Ventana(void); // Rutinas constructora y destructora de la clase ventana
    ~Ventana();
    virtual  void Define_ventana(const int x1, const int y1, const int x2, const int y2, const
    int cl = Blanco); // Define la ventana y su color
    virtual  void Ventana_centrada(const int lg_x, const int lg_y, const int cl, const int
    inc_x=0, const int inc_y=0); // Define una ventana centrada de longitud y color dados
    void Centra_texto(const unsigned int y, const char *texto); // Centra texto dentro de la
    ventana
    void Visualiza_texto(const unsigned int x,const unsigned int y, const char *texto, const
    int t_v = 0); // Visualiza una cadena dentro de la ventana
    void Visualiza_caracter(const unsigned int x,const unsigned int y, const char car, const
    int cl, const int t_v = 1); // Visualiza un carácter dentro de la ventana
    void Activa_graba_ventana(const int tp); // Activa grabar ventana
    void Activa_color_refresco(const int col) {C_REF = col;} // Indica el color de la ventana
    al cerrar
    void Actual_pos_ventana(unsigned int &x1, unsigned int &y1, unsigned int &x2,
    unsigned int &y2)
    // {x1 = X1, y1 = Y1, x2 = X2, y2 = Y2;}
    // Actual posición dentro de la ventana
    //  void Posicion_ventana(const int x, const int y)
    //   {x,y,X2-X1,Y2-Y1) // Mover la actual ventana una determinada cantidad de pixeles
    virtual  void Mover_ventana(const int x, const int y);
    virtual  void Mover_ventana_n_pixeles(const int x, const int y);
    virtual  void Cambiar_tamano(const int x1, const int y1, const int x2, const int y2); //
    Indica el nuevo tamaño de la ventana
    int  Largo_X(void) {return ((X2-X1)+1);} // Retorna el largo en X de la ventana
    int  Largo_Y(void) {return ((Y2-Y1)+1);} // Retorna el largo en Y de la ventana
    int  Largo_caracter(void) {return L_CAR;} // Retorna el largo del carácter actual
    int  Ancho_caracter(void) {return A_CAR;} // Retorna el ancho del carácter actual
    virtual  int  Mouse_dentro(void); // Indica si el mouse esta dentro de la ventana
    void Define_color_fondo(const int cl) {C_FOND = cl;} // Define el color de fondo al
    visualizar texto
    void Define_color_texto(const int cl) {C_TEXT = cl;} // Define el color del texto a
    visualizar
    void Define_color(const int cfd, const int ctx) {C_FOND = cfd, C_TEXT = ctx;} //
    Define el color del fondo y del texto a visulizar
    void Define_color_ventana(const int cl) {C_FOND = C_VENT = cl;} // Define el color
    de la ventana
    char Actual_color_texto(void)  {return C_TEXT;} // Indica el actual color del texto
    char  Actual_color_fondo(void)  {return C_FOND;} // Indica el actual color de fondo
    del texto
    char  Actual_color_ventana(void) {return C_VENT;} // Indica el actual color de la
    ventana
     void Tamano_letra(const int lg_x, const int lg_y); // Define el tamaño de la letra a
    visualizar en la ventana
```

```
    void Define_tipo_letra(const int tipo, const int horient = 0, const int multx = 1, const
    int divx = 0, const int multy = 0, const int divy = 0); // Define el tipo de letra, orientación
    y tamaño
    virtual void Limpia_recuadro(const int x1, const int y1, const int x2, const int y2, const
    int cl); // Limpia una porción de la ventana
    virtual void Dibuja(void) {Limpia_recuadro(0,0,Largo_X()-1,Largo_Y()-1,C
    _VENT);} // Dibuja la ventana
    void Oculta_ventana(const int tp); // Oculta la ventana o revisualiza esta
    virtual void Ventana_activa(const int tp) {VENTANA_ACTIVA = tp;} // Activa o
    desactiva la ventana
    virtual int Estado(void) {return VENTANA_ACTIVA;} // Retorna el estado Ventana
    (Activo o no)
    int Retorna_inicio_X(void) {return Inicia_X;} // Retorna el inicio del área de trabajo
    en X
    int Retorna_inicio_Y(void) {return Inicia_Y;} // Retorna el inicio del área de trabajo
    en Y
    void Graba_ventana(const char *arch, const int tp); // Graba la actual ventana, si tp es
    (0) Imagen con formato (1) imagen sin formato
    void Lee_ventana(const char *arch, const int tp); // Lee una imagen a la actual ventana,
    si tp es (0) Imagen con formato (1) imagen sin formato
    void Marco(const int x1, const int y1, const int x2, const int y2, const int c_r_i, const
    int c_r_s); // Dibuja un marco dentro de la ventana
    virtual int Posicion_mouse(int &x, int &y); // Retorna la posición del mouse dentro
    de la ventana
    unsigned int Presionado(const int boton); // Revisa si el botón es oprimido por el mouse
    dentro de la ventana
    virtual void Limpia_recuadro(const int x1, const int y1, const int x2, const int y2,const
    int cfd, const int c_r_i, const int c_r_s); // Limpia y marca el recuadro
    // Selecciona una porción de la ventana, regresando las coordenadas de esta área
    // Retorna (1) si se selecciono correctamente
    //        (0) si el usuario cancelo la selección por medio de la tecla ESC
    int Selecciona_porcion_ventana(C_dbl &vent);
};
/*
Clase Ventana:
  La clase VENTANA permite definir una ventana de trabajo, visualizar
  texto dentro de ella, cambiar el tamaño y tipo de la letra a
  visualizar, mover esta, cambiar el tamaño, grabar la imagen existente
  antes de abrir la ventana y activar la ventana.
  Forma de uso:
    Primero defina el objeto:
      Ya sea dinámico o estático
    Después defina:
      Activa_graba_ventana
        Por omisión esta apagado, si desea grabarla activarlo con 1.
        (recomendable)
      Activa_color_refresco
        Por omisión no esta activada, activela con el color del
        fondo antes de activar la ventana (sino hay texto entes
```

```
        es mejor de la ventana usar esta)
    Después defina:
      Define_ventana
        Define la ventana en una determinada área de la
        ventana dada por X1,Y1,X1,Y2
      Centra_texto
        Define una ventana centrada indicando la longitud
        de esta por medio de longitud en X y Y
    Después
      Dibuja
        Dibuja la ventana
    Después puede utilizar todos los demás comportamientos según sean
    las necesidades del programa. al destruirse el objeto restaura la
    ventana original (si se grabo esta) o redibuja con el color de
    refresco (si este fue activado)
ejemplo:
  Ventana *Vpresent = new Ventana;
  Vpresent-Activa_graba_ventana(1);
  Vpresent-Ventana_centrada(480,300);
  Vpresent-Dibuja();
  Vpresent-Define_color_texto(Negro);
  Vpresent-Define_tipo_letra(4,0,2,2,2,2);
  Vpresent-Centra_texto(60,NOMBRE_SISTEMA);
  Vpresent-Define_color_texto(Azul9);
  Vpresent-Define_tipo_letra(5,0,1,3,1,3);
  sprintf(xcad,"Version: %s @",VERSION_SISTEMA);
  Vpresent-Centra_texto(110,xcad);
  Vpresent-Define_color_texto(Azul1);
  Vpresent-Define_tipo_letra(7,0,1,3,1,3);
  Vpresent-Centra_texto(140,DESCRIPCION1);
  Vpresent-Centra_texto(155,DESCRIPCION2);
  Vpresent-Centra_texto(170,DESCRIPCION3);
  Vpresent-Define_tipo_letra(0,0,-1,0,0,0);
  Vpresent-Define_color_texto(Negro);
Vpresent-Visualiza_texto(20,220,"Autor: Oswaldo Toxtle Hernandez");
Vpresent-Define_tipo_letra(2,0,-2,0,0,0);
Vpresent-Define_color_texto(Rojo1);
sprintf(xcad,"Fecha de compilación: %s Hora: %s",__DATE__,__TIME__);
Vpresent-Visualiza_texto(20,280,xcad);
sprintf(xcad,"(@) Todos los derechos reservados, registro en tramite %s ",__DATE__);
Vpresent-Visualiza_texto(20,290,xcad);
Vpresent-Define_tipo_letra(0,0,-1,0,0,0);
while(Programa_activo && Tecla != ENTER && Tecla != ESC) Administrador();
delete Vpresent;
      */
#endif
#ifndef __V_MARCO_HPP__
```

```
#define __V_MARCO_HPP__
#include "v_basica.hpp"
// Clase Ventana con marco
class Ventana_marco: public Ventana {
private:
   char C_rec_inf;  // Color del recuadro superior
   char C_rec_sup;  // Color del recuadro inferior
   char C_base_a;  // Color de la base de la ventana activada
   char C_base_n;  // Color de la base de la ventana no activada
   unsigned int Mouse_dentro_ventana:1;  // Indica si el mouse esta adentro de la ventana
public:
   Ventana_marco(void); // Constructor de ventana con marco
   void  Marco_ventana(const int tb); // Dibuja el marco de la ventana según el TB
   virtual  void  Define_colores(const int c_v, const int c_r_i, const int c_r_s, const int
   c_b_a = Verde1, const int c_b_n = Negro); // Define los colores de la ventana con marco

    int  Mouse_dentro(void); // Revisa si el mouse esta dentro de la ventana con marco
   virtual  void  Dibuja(void); // Dibuja la ventana con marco
};
/*
Clase Ventana_marco:
La clase VENTANA_MARCO permite definir una ventana de trabajo con
marco el cual se activa si el mouse esta sobre la ventana, permite
hacer todas las actividades de la clase VENTANA y cambia el color
y forma del marco.
  Forma de uso:
    Primero defina el objeto:
      Ya sea dinámico o estático
    Después defina:
      Activa_graba_ventana
    Por omisión esta apagado, si desea grabarla activarlo con 1.
        (recomendable)
      Activa_color_refresco
  Por omisión no esta activada, actívela con el color del
  fondo antes de activar la ventana (sino hay texto antes
        es mejor de la ventana usar esta)
    Después defina:
      Define_ventana
        Define la ventana en una determinada área de la
        ventana dada por X1,Y1,X1,Y2
      Centra_texto
        Define una ventana centrada indicando la longitud
        de esta por medio de longitud en X y Y
    Después
      Dibuja
        Dibuja la ventana
    Después puede utilizar todos los demás comportamientos según sean
    las necesidades del programa. al destruirse el objeto restaura la
```

```
    ventana original (si se grabo esta) o redibuja con el color de
    refresco (si este fue activado)
Ejemplo
  Ventana_marco *Vpresent = new Ventana_marco;
  Vpresent-Activa_graba_ventana(1);
  Vpresent-Ventana_centrada(480,300);
  Vpresent-Dibuja();
  Vpresent-Define_color_texto(Negro);
  Vpresent-Define_tipo_letra(4,0,2,2,2,2);
  Vpresent-Centra_texto(60,NOMBRE_SISTEMA);
  Vpresent-Define_color_texto(Azul9);
  Vpresent-Define_tipo_letra(5,0,1,3,1,3);
  sprintf(xcad,"Version: %s  @",VERSION_SISTEMA);
  Vpresent-Centra_texto(110,xcad);
  Vpresent-Define_color_texto(Azul1);
  Vpresent-Define_tipo_letra(7,0,1,3,1,3);
  Vpresent-Centra_texto(140,DESCRIPCION1);
  Vpresent-Centra_texto(155,DESCRIPCION2);
  Vpresent-Centra_texto(170,DESCRIPCION3);
  Vpresent-Define_tipo_letra(0,0,-1,0,0,0);
  Vpresent-Define_color_texto(Negro);
Vpresent-Visualiza_texto(20,220,"Autor: Oswaldo Toxtle Hernandez");
Vpresent-Define_tipo_letra(2,0,-2,0,0,0);
Vpresent-Define_color_texto(Rojo1);
sprintf(xcad,"Fecha de compilaci¢n: %s Hora: %s",__DATE__,__TIME__);
Vpresent-Visualiza_texto(20,280,xcad);
sprintf(xcad,"(@) Todos los derechos reservados %s ",__DATE__);
Vpresent-Visualiza_texto(20,290,xcad);
Vpresent-Define_tipo_letra(0,0,-1,0,0,0);
while(Programa_activo && Tecla != ENTER && Tecla != ESC) {
Administrador();
Mouse_dentro();
  }
  delete Vpresent;
*/
#endif
#ifndef __V_TITULO_HPP__
#define __V_TITULO_HPP__
#include "v_marco.hpp"
#include "colores.hpp"
#include "mouse.hpp"
// Clase ventana con titulo
class Ventana_titulo: public Ventana_marco {
private:
   char pt_x1, pt_y1, pt_x2, pt_y2; // Almacena la posición del área de titulo
   char C_fdo; // Almacena el color de fondo, titulo y texto
   char C_tit_act;
```

```
    char C_tit_no_act;
    char C_txt;
    char *Titulo;
    char Lt_xtipo; // Almacena el tipo de Letra actual dentro del la ventana
    int  Lt_xmulx; // Almacena la escala en X
    char Lt_xdivx;
    char Lt_xmuly; // Almacena la escala en Y
    char Lt_xdivy;
    unsigned int  Lt_xhori; // Almacena la orientación de la letra
    unsigned int  Tit_centrado; // Indica si el titulo será centrado o no
    void Dibuja_titulo(void); // Dibuja el titulo de la ventana
public:
    Ventana_titulo(void); // Constructor y destructor de la clase
    ~Ventana_titulo();
    // Definición de la llamada de la ventana con titulo
    virtual void Ventana_centrada(const char *tit, const int lgx, const int lgy, const int
    inc_x=0, const int inc_y=0);
    virtual void Define_ventana(const char *tit, const int x1, const int y1, const int x2,
    const int y2);
    void Define_titulo(const char *tit); // Define el nombre del titulo
    void Define_posicion_titulo(const int x1, const int y1, const int x2, const int y2) {pt_x1
    = x1,pt_y1 = y1,pt_x2 = x2,pt_y2 = y2;} // Define la posición de la ventana dentro del
    titulo
    // Define la posición de la ventana
    // void Define_posicion_ventana(const int x, const int y)
    // {Ventana_marco::Posicion_ventana(x,y); Ventana_titulo::Dibuja();}
    void Cambia_tipo_letra_titulo(const int tipo, const int horient, const int multx, const
    int divx, const int multy, const int divy, const int centr = 1) {Lt_xtipo = tipo, Lt_xhori
    = horient, Lt_xmulx = multx, Lt_xdivx = divx, Lt_xmuly = multy, Lt_xdivy = divy,
    Tit_centrado = centr;} // Cambia el tipo de letra, orientación y tamaño del titulo
    virtual void Define_colores(const int cfdo, const int ctita, const int ctitna, const int
    ctxt) {C_fdo = cfdo, C_tit_act = ctita, C_tit_no_act = ctitna, C_txt = ctxt;} // Define los
    colores que existen en la ventana
    virtual void Dibuja(void) {Ventana_marco::Dibuja();Dibuja_titulo();}// Dibuja la
    ventana con el titulo
    virtual void Ventana_activa(const int tp) {Ventana_marco::Ventana_activa(tp);Dibu-
    ja_titulo();} // Activa o desactiva la ventana (Cambia el color del titulo)
    virtual void Cambiar_tamano(const int x1, const int y1, const int x2, const int
    y2)  {Ventana_marco::Cambiar_tamano(x1,y1,x2,y2); Ventana_titulo::Dibuja();}//
    Cambia el tamaño de la ventana
};
/*
Clase Ventana_titulo:
La clase VENTANA_TITULO permite definir una ventana de trabajo
con un titulo, así como todas las actividades que realiza la
clase VENTANA_MARCO así también manipular el color y titulo de
esta.
  Forma de uso:
```

Primero defina el objeto:
Ya sea dinámico o estático
Después defina:
Activa_graba_ventana
Por omisión esta apagado, si desea grabarla activarlo con 1. (recomendable)
Activa_color_refresco
Por omisión no esta activada, actívela con el color del fondo antes de activar la ventana (sino hay texto antes es mejor de la ventana usar esta)
Después defina:
Define_ventana
Define la ventana en una determinada área de la ventana dada por X1,Y1,X1,Y2
Centra_texto
Define una ventana centrada indicando la longitud de esta por medio de longitud en X y Y
Después
Dibuja
Dibuja la ventana
Después puede utilizar todos los demás comportamientos según sean las necesidades del programa. al destruirse el objeto restaura la ventana original (si se grabo esta) o redibuja con el color de refresco (si este fue activado)

Ejemplo:

```
Ventana_titulo *Vpresent = new Ventana_titulo;
Vpresent->Activa_graba_ventana(1);
Vpresent->Ventana_centrada("Acerca INTERFAZ",480,300);
Vpresent->Dibuja();
Vpresent->Define_color_texto(Negro);
Vpresent->Define_tipo_letra(4,0,2,2,2,2);
Vpresent->Centra_texto(60,NOMBRE_SISTEMA);
Vpresent->Define_color_texto(Azul9);
Vpresent->Define_tipo_letra(5,0,1,3,1,3);
sprintf(xcad,"Version: %s @",VERSION_SISTEMA);
Vpresent->Centra_texto(110,xcad);
Vpresent->Define_color_texto(Azul1);
Vpresent->Define_tipo_letra(7,0,1,3,1,3);
Vpresent->Centra_texto(140,DESCRIPCION1);
Vpresent->Centra_texto(155,DESCRIPCION2);
Vpresent->Centra_texto(170,DESCRIPCION3);
Vpresent->Define_tipo_letra(0,0,-1,0,0,0);
Vpresent->Define_color_texto(Negro);
Vpresent->Visualiza_texto(20,220,"Autor: Oswaldo Toxtle Hernandez");
Vpresent->Define_tipo_letra(2,0,-2,0,0,0);
Vpresent->Define_color_texto(Rojo1);
sprintf(xcad,"Fecha de compilación: %s Hora: %s",__DATE__,__TIME__);
```

```
Vpresent-Visualiza_texto(20,280,xcad);
sprintf(xcad,"(@) Todos los derechos reservados, registro en tramite %s ",__DATE__);
Vpresent-Visualiza_texto(20,290,xcad);
Vpresent-Define_tipo_letra(0,0,-1,0,0,0);
while(Programa_activo && Tecla != ENTER && Tecla != ESC) Administrador();
delete Vpresent;
*/
#endif
#ifndef __V_ICONOS_HPP__
#define __V_ICONOS_HPP__
#include "v_titulo.hpp"
#include "icono.hpp"
// Clase Ventana con iconos para su control
class Ventana_iconos: public Ventana_titulo {
private:
   Icono    *Ico1;          // Puntero al primer icono
   Icono    *Ico2;          // Puntero al segundo icono
   Icono    *Ico3;          // Puntero al tercer icono
   char     *Nico1;         // Contiene el nombre del primer icono
   char     *Nico2;         // Contiene el nombre del segundo icono
   char     *Nico3;         // Contiene el nombre del tercer icono
   char     IconoActual;    // Indica el icono sobre el que se encuentra el mouse
   char     IconoPresionado;  // Indica el icono presionado
   char     Tipo;           // Indica la cantidad de iconos en la ventana
   void  Dibuja_iconos(void); // Dibuja los iconos de la ventana
   void  Carga_iconos(void); // Carga los iconos
public:
   Ventana_iconos(void); // Constructor de ventana con iconos
   ~Ventana_iconos();// Destructor de ventana con iconos
   virtual   void  Define_ventana(const char *tit, const int x1, const int y1, const int x2,
   const int y2, const int tp); // Define la ventana
   virtual   void  Ventana_centrada(const char *tit, const int lgx, const int lgy, const int
   tp, const int inc_x=0, const int inc_y=0); // Define la ventana
   virtual   void  Dibuja(void) {Ventana_titulo::Dibuja();Dibuja_iconos();}// Dibuja la
   ventana
   int   Itera(void); // Revisa el estado de los iconos
   int   Icono_actual(void) {return IconoActual;} // Indica el icono actual
   int   Icono_presionado(void)  {return IconoPresionado;} // Indica el icono presionado
   void  Nombre_iconos(const char *n1, const char *n2 = " ", const char *n3 = " "); //
   Define el nombre de los iconos
   virtual   void  Ventana_activa(const int tp)  {Ventana_titulo::Ventana_activa(tp); Di-
   buja_iconos();}// Activa o desactiva la ventana (solo titulo)
   virtual   void  Cambiar_tamano(const int x1, const int y1, const int x2, const int y2); //
   Cambia el tamaño de la ventana
};
/*
Clase Ventana_iconos:
La clase VENTANA_ICONOS permite definir una ventana de trabajo
```

con títulos e iconos teniendo todos los comportamientos de la ventana con títulos además de los comportamientos necesarios para manipular la ventana con iconos

Forma de uso:

Primero defina el objeto:

Ya sea dinámico o estático

Después defina:

Activa_graba_ventana

Por omisión esta apagado, si desea grabarla activarlo con 1. (recomendable)

Activa_color_refresco

Por omisión no esta activada, actívela con el color del fondo antes de activar la ventana (Si no hay texto antes es mejor de la ventana usar esta)

Después defina:

Define_ventana

Define la ventana en una determinada área de la *ventana dada* por X1,Y1,X1,Y2

Centra_texto

Define una ventana centrada *indicando la longitud* de esta por medio de longitud en X y Y

Después

Dibuja

Dibuja la ventana

Para *controlar la ventana* use:

Itera

Indica si se presionó algún *icono dentro de la ventana*

Icono_actual

Indica el icono sobre el que el mouse esta actualmente

Icono_presionado

Indica el numero de icono presionado dentro de la ventana

Después puede utilizar todos los demás comportamientos según sean las necesidades del *programa*. al destruirse el objeto restaura la ventana original (si se grabo esta) o redibuja con el color de refresco (si este fue activado)

Nota: la ventana puede visualizar uno o dos iconos, por omisión están puestos los iconos de cerrar y ayuda.
Usa agregación para los iconos.

Ejemplo:

```
Ventana_iconos *Vpresent = new Ventana_iconos;
Vpresent-Activa_graba_ventana(1);
Vpresent-Ventana_centrada("Acerca Fractal",480,300,0);
```

```
Vpresent-Dibuja();
Vpresent-Define_color_texto(Negro);
Vpresent-Define_tipo_letra(4,0,2,2,2,2);
Vpresent-Centra_texto(60,NOMBRE_SISTEMA);
Vpresent-Define_color_texto(Azul9);
Vpresent-Define_tipo_letra(5,0,1,3,1,3);
sprintf(xcad,"Version: %s @",VERSION_SISTEMA);
Vpresent-Centra_texto(110,xcad);
Vpresent-Define_color_texto(Azul1);
Vpresent-Define_tipo_letra(7,0,1,3,1,3);
Vpresent-Centra_texto(140,DESCRIPCION1);
Vpresent-Centra_texto(155,DESCRIPCION2);
Vpresent-Centra_texto(170,DESCRIPCION3);
Vpresent-Define_tipo_letra(0,0,-1,0,0,0);
Vpresent-Define_color_texto(Negro);
Vpresent-Visualiza_texto(20,220,"Autor: Oswaldo Toxtle Hernandez");
Vpresent-Define_tipo_letra(2,0,-2,0,0,0);
Vpresent-Define_color_texto(Rojo1);
sprintf(xcad,"Fecha de compilación: %s Hora: %s",__DATE__,__TIME__);
Vpresent-Visualiza_texto(20,280,xcad);
sprintf(xcad,"(@) Todos los derechos reservados, %s ",__DATE__);
Vpresent-Visualiza_texto(20,290,xcad);
Vpresent-Define_tipo_letra(0,0,-1,0,0,0);
while(!Vpresent-Itera() && Programa_activo && Tecla != ENTER && Tecla != ESC)
Administrador();
delete Vpresent;
*/
#endif
#ifndef __BOTON_HPP__
#define __BOTON_HPP__
#include "v_marco.hpp"
#include "mouse.hpp"
// Clase Boton
class Boton: public Ventana_marco {
protected:
   unsigned int Boton_activo:1;  // Indica si el boton esta activo o no
public:
   void  Abrir(const unsigned int x, const unsigned y, const int lgx = 36, const int lgy = 36,
   const int cf = Gris9)  {Ventana_marco::Define_ventana(x,y,x+lgx,y+lgy,cf);} // Indica
   la posición donde se visualizara el boton
   int  Oprimido(void); // Revisa si el boton es oprimido por el mouse dentro de la ventana
   virtual  void  Dibuja(void) = 0; // Dibuja el boton
   virtual  void  Activar(const int tp) {Boton_activo = tp;} // Activa el boton
   virtual  int  Estado(void) {return Boton_activo;} // Retorna el estado del boton
};
      /*
La clase de BOTON hereda todos los comportamientos de la
clase Ventana_marco, así como los comportamientos pertinentes
```

del manejo de botones.
Forma de uso:
Primero defina el objeto:
Ya sea dinámico o estático
Después defina:
Abrir
Indica la posición del icono dentro de la pantalla
y la longitud de este
Activar
Indica si el boton estará o no activo
Después:
Dibuja
Dibuja el boton (solo el recuadro)

Para *controlar* la ventana use:
Oprimido
Revisa si fue presionado el boton
Estado
Devuelve el estado del boton

Después puede utilizar todos los demás comportamientos según sean las necesidades del programa. al destruirse el objeto restaura la ventana original (si se grabo esta) o redibuja con el *color* de *refresco* (si este fue activado)
*/
#endif
#ifndef __COMBO_B__
#define __COMBO_B__
#include "icono.hpp"
#include "definic.hpp"
class Combo_box {
private:
Icono *Ic;
Ventana_marco *Vt;
char *cadena;
C_pt ini;
C_pt lg;
unsigned int Actual;
unsigned int selec;
const char *Opc;
const char *Opc_des;
public:
Combo_box(void); // Constructor de la clase
~Combo_box(); // Destructor de la clase
// Parámetros de la clase
void Parametros(const char *Arr_menu[], const C_pt vent, const unsigned int act, const char *opc = "", const char *opc_des = "");

```
    void  Dibuja(const char *Arr_menu[]);  // Dibuja la clase
    void  Controla(const int tecla, const char *Arr_menu[]);  // Controla la selección dentro
    de la clase
    unsigned int Retorna_opcion(void) {return (Actual+1);}// Retorna la opción seleccio-
    nada
    void  Fija_actual_opcion(unsigned int act)  {if(act) Actual = act-1; else Actual = 0;}
};
/*
Ejemplo de uso
static const char *M_archivos[] = {"
Descripci¢n Proyecto","
Graba Pantalla","
Lee Pantalla","
Edita Archivo","
Visualiza Archivo","
Termina Programa","
0"
};
void mi_funcion(void)
{
Combo_box ab;
C_pt vent;
int unsigned selec;
vent.x = 100, vent.y = 100;
ab.Parametros(M_archivos, vent,2);
ab.Dibuja();
while(Caracter != 'x') {
NCO-Administrador();
ab.Controla(Tecla, M_archivos);
}
select = ab.Retorna_opcion()
}
*/
#endif
```

CONCLUSIONES

1. ¿Por que no existe especialización en ingeniería de software?

Hasta ahora la ingeniería de software esta siendo considerada como una parte medular para el desarrollo de sistemas, apenas las empresas toman consciencia de las metodologias

2.¿Qué se debe enseñar en las aulas de clases ahora con lo orientado a objetos?

Partiendo de que cuando existió la programación Lineal se enseño algoritmia, después llego la programación estructurada, y se enseño estructura de datos ahora con la programación orientada a objetos ¿que se debe enseñar?.

Nuestra propuesta son las siguientes:

- ·Modelos formales de objetos y clases.- Especificaciones formales de objetos y clases
- ·Componentes de Software Reusables.- Esto se basa en software implementado en la cual se tiene que tener la misma semántica de objetos
- ·Patrones.- Esto es parecido a recetas, que solucionan problemas y que por regla deben estar implantados (el patrón) en 2 sistemas funcionando

3.¿Para que sirve la metodología y el paradigma Orientado a Objetos?

Los objetivos del software actual hacen imposible crear software Industrial-Fuerte (de gran escala) por lo tanto se hace indispensable una metodología para coordinar el trabajo así como la comunicación entre desarrolladores.

El paradigma orientado a objetos nos ayuda a lidiar con la complejidad de los sistemas de gran escala, es decir ayuda a "manejar" la complejidad ya que la forma de abstraer el mundo físico, es muy similar a como nosotros aprendemos y comprendemos el mundo. Además es más fácil modelar la parte dinámica de los sistemas así como sus relaciones, reduciendo así significativamente lo complicado. Que en la forma estructurada.

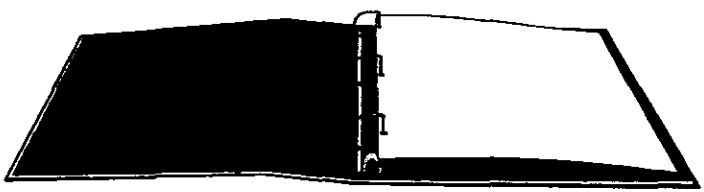
4.La orientación a objetos surgió como una evolución de la programación así como del modelado (análisis y diseño) de los sistemas, pero se ha vuelto una revolución en la forma de pensar. Ya que ha impactado de tal manera el mundo en general y la prueba de esto. Es el hecho de que se fabrique hardware Orientado a Objetos, y las empresas se organicen utilizando el paradigma Orientado a Objetos por ejemplo el Centro de Investigación y Desarrollo de la compañía General Electric. Además de que a abierto nuevas brechas tecnológicas y coadyuvado a otras como por ejemplo (la creación de los lenguajes "Visuales" y Bases de Datos OO,a ayudado a la visualización científica, así como a la realidad virtual y a la simulación)

5.Estamos en la transición de lo estructurado a lo Orientado a Objetos.

Cuando los desarrolladores puedan manejar perfectamente los conceptos de herencia, agregación, metaclase, clases abstractas, etc. es decir entender su semántica no solo el hecho tecnológico (su sintaxis, implementación) y empiecen a diseñar bajo esa perspectiva.

Además de que se llegue a un consenso mundial en la terminología de lo Orientado a Objetos hasta entonces estaremos del otro lado.

GLOSARIO



Abstracción.- Operación por medio de la cual nuestro espíritu luego de haber distinguido los caracteres esenciales de un grupo de hechos, los separa de las propiedades secundarias para generalizarlas.

Algoritmia.- Ciencia del calculo aritmético y algebraico, teoría de los números.

Algoritmo.- Sucesion de determinadas operaciones que permiten efectuar un cierto cálculo en número determinado de pasos.

Apelar.- Recurrir a una persona o cosa para hallar favor, solución o remedio: apelar a un último recurso.

Apriori.- (voces lat., partiendo de lo anterior). Antes de la experiencia, antes de los hechos. Dícese de lo que se admite fundándolo en datos anteriores a la experiencia o que no provienen de ella. Anterior a cualquier conocimiento profundizado; al primer contacto.

Atributo.- Propiedad inherente a un objeto sin la cual no puede existir o ser concebido.

Aunar.- Poner juntas o armonizar varias cosas. 2. Unificar.

Axioma.- Principio o proposición tan evidente que no necesita explicación o demostración. Actualmente se le considera como punto de partida para razonamiento posterior; a partir de los cuales se deducirán otros enunciados que recibirán el nombre de teorema.

Capacidad.- Propiedad de poder contener cantidad de algo. Aptitud o suficiencia para alguna cosa.- Talento o disposición para comprender bien las cosas.

Característica.- Perteneciente o relativo al carácter. Aplicase a la cualidad que da carácter o sirve para distinguir una persona o cosa.

Categorias.- En filosofía nociones lógicas fundamentales que reflejan las propiedades esenciales, los aspectos, las relaciones más generales entre los fenómenos reales.

Complejo.- Dicese de lo que se compone de elementos diversos.- Sistema compuesto de elementos distintos que guardan entre sí relaciones determinadas constituyendo un todo cerrado y autónomo.

Complicado.- Fuera de los actos, de los sentidos y potencias de hombre. De difícil comprensión.

Concepto.- Forma del pensamiento humano que permite captar los caracteres generales, esenciales, de las cosas y de los fenómenos de la realidad objetiva. El proceso del conocimiento por el hombre comienza por la percepción de los sentidos, la etapa siguiente es la formación del concepto, es una forma de abstracción.

Conciencia.- Forma superior específicamente humana, del reflejo de la realidad objetiva. La conciencia del hombre es una función de "Ese fragmento especialmente complejo de la materia que se llama cerebro humano".

Cualidad.- Cada una de las circunstancias o caracteres naturales o adquiridas, que distinguen a las personas o cosas.- Una de las categorías que se le puede predicar al ser. Las cualidades primarias son inseparables de los cuerpos (extensión, figura, movilidad y corporeidad). Las cualidades secundarias son posibilidades del objeto de producir sensaciones en el receptor de forma que se les atribuye a los cuerpos (color, sonido, gusto).

Definición.- Propocisión que expone con claridad y exactitud los caracteres genéricos y diferenciales de una cosa material o inmaterial. Explica la esencia o naturaleza de algo.

Definir.- Fijar con claridad, exactitud y precisión la significación de una palabra o la naturaleza de una cosa.

Dicotomía.- División, oposición entre dos cosas.

Dilatarse.- Propiedad de los cuerpos de aumentar su volumen, ocupando mayor espacio.

Esencia y Fenómeno.- Categorías filosóficas que reflejan diferentes aspectos de los objetos, de los procesos de la realidad objetiva. La esencial expresa las características fundamentales, su naturaleza interna, los procesos profundos que se desarrollan. El fenómeno es la manifestación exterior de la esencia, la forma exterior en que los objetos y procesos aparecen en la superficie.

Especificación.- Definición precisa y ordenada que describe la lógica y la finalidad de las funciones de proceso que efectúa un programa.

Especificar.- Explicar, declarar con individualidad una cosa.- Fijar o determinar de modo preciso.

Espectro.- (lat. spectrum, simulacro). FÍS. Conjunto de las líneas resultantes de la descomposición de una luz compleja. FÍS. Distribución de la intensidad de una onda, acústica o electromagnética, o de un haz de partículas, en función de la frecuencia o de la energía.

Estrategia.- Arte de dirigir las operaciones o acciones militares, políticas o económicas.

Evolución y Revolución.- La evolución es una acumulación lenta y gradual de cambios cuantitativos; la revolución es un cambio brusco, radical y cualitativos.

Expandir.- Propiedad de un cuerpo de poder ocupar un espacio mayor del que ocupa.

Extender.- Hacer que una cosa aumentando su superficie ocupe mayor lugar o espacio que el que antes ocupaba dar mayor amplitud.

Forma y contenido.- Categorías de la dialéctica materialista. Por contenido se entiende el aspecto más importante del objeto, lo que caracteriza su esencia íntima, el fondo que se manifiesta en sus caracteres y sus propiedades. La forma es la organización interna del contenido lo que une a un todo los elementos del contenido.

Heurística.-(Hallar) Ciencia de la investigación y deducción aplicadas a una rama particular de la lógica. Es un Neologismo muy usado.2 Disciplina que trata de establecer las reglas de la investigación.

Idea.- Reflejo de la realidad en la conciencia.

Imaginación.- Reflejo original de la realidad objetiva en la conciencia representación figurada de los fenómenos reales e irreales.

Inducción y deducción.- Inducción razonamiento que va de lo particular de los hechos a las generalizaciones. Deducción razonamiento que va de lo general a lo particular.

Intuición.- Facultad particular de contemplación espiritual que permite conocer la verdad sin que intervenga una actividad racional o lógica.

Metáfora.- Forma de dicción que consiste en expresar una idea valiéndose de otra, con la cual guarda analogía o semejanza. Por ejemplo el báculo de la vejez, la flor de la vida.

Método.- Procedimiento para alcanzar un determinado fin.

Metodología.- Ciencia del método.- Conjunto de métodos que se siguen en una investigación científica o en una exposición doctrinal.

Neologismo.- Palabra, expresión o acepción de creación reciente, que aparece o se adopta en una lengua.

Noción.- Conocimiento o idea que se tiene de una cosa; conocimiento elemental.

Norma.- Orden concreto e invariable que guardan las cosas naturales.- Método de hacer una operación.

Ortogonal.- 2 rectas o curvas que forma ángulo recto o 2 planos cuando se cortan forman un ángulo recto.

Paradoja.- Aserción falsa o inexacta, que se presenta con apariencia verdadera. En filosofía.- proposición que aun cuando representa un concepto verdadero lo expresa con términos al parecer contradictorios a lo que se quiere decir; de donde resulta algo a primera vista inverosímil. En Física se da este nombre a todo razonamiento que se halla aparentemente en contradicción con las leyes fundamentales de la naturaleza o los axiomas más elementales de la razón.

Pensamiento.- Producto superior de una materia orgánica particular llamada cerebro.

Percepción.- Reflejo directo de los objetos del mundo real que actúan sobre nuestros sentidos.

Perpendicular.- Que forma un ángulo recto con una recta o plano. 2 Posición de una recta respecto a otra, de modo que sean iguales entre si los ángulos que forman ambas

Perplejo.- Indeciso, confuso.

Perspectiva.- Forma de representar por medio del dibujo, en un plano, los objetos tal y como se ven a cierta distancia y en una posición dada. 2. Quiere decir la

forma en como se concibe la idea. 3. Tener en perspectiva una idea o proyecto, quiere decir tener concebida una idea o proyecto cuya realización esta algo distante.

Proceder.- Pasar a poner en ejecución una cosa a la cual precedieron algunas diligencias o tares.- Continuar en la ejecución de algunas cosas que piden trato sucesivo.

Procedimiento.- Método de ejecutar algunas cosas.

Profusión.- Abundancia excesiva.

Propiedad.- Atributo o cualidad esencial de una persona o cosa. Una de las categorías predicables de todo objeto.

Razonamiento.- Forma del pensamiento que consiste en extraer un juicio nuevo (conclusión) que deriva necesariamente de juicios dados (premisa).

Regla.- Norma que ha de regir la conducta de los hombres en el estudio de una ciencia, en la práctica de un arte, o en el ejercicio.

Significado.- Parte fundamental junto al significante del concepto lingüístico, es la idea o representación mental que tenemos de una cosa.

Significar.- Ser una cosa, ser una palabra, frase, expresión. Signo de una idea o de un pensamiento o de una cosa material.

Sinergia.- Cooperación y conexión de varias actividades para realizar una misma función. 2. Física. Acción simultanea de dos fuerzas cuyo momento resultante es igual a la suma de los componentes.

Subyacente.- Que yace o está debajo de otra cosa.

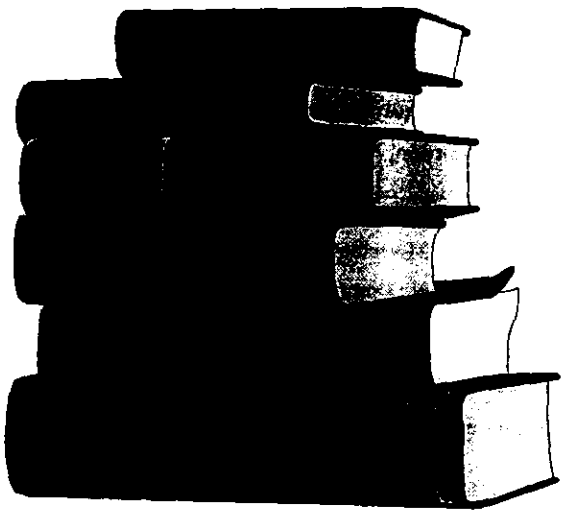
Taxonomía.- Ley de ordenación o manera como deben ser dispuestos los objetos o hechos. En sentido estricto es la parte de la ciencia que dicta las leyes o principios de clasificación de los objetos naturales.

Verdad.- Conformidad de una noción, de una idea con el objeto conocimiento que refleja fielmente la realidad objetiva.

Yuxtaponer.- Poner una cosa junto a otra o poner dos cosas juntas.

Yuxtaposición.- Acción y efecto de yuxtaponer.

BIBLIOGRAFÍA



Programmers at Work
Lammers, Susan, Tempus Books of Microsoft Press; 1989

Object-Oriented Design with Applications
G. Booch, Benjamin Cummings; 1988.

Teoría General de los Sistemas
Ludwing Von Bertalanfy, fondo de cultura económica.

Realización de Páginas Web (info negocios)
Melanie Peyrot Raab, Licenciatrua Diseño Gráfico.

Objeto-Oriented Systems Analysis
S. Shlaer & S Mellor; Yourdon Press; 1992.

Object-Orented Analysis. Second edition
P. Coad / E. Yourdon; Yourdon Press; 1991

Object-Oriented Modeling and Design
J. Rumbaugh / et al.; Prentice Hall; 1991

Object-Oriented Software Engineering
L. jacobson / et al.; Addison Wesley; 1992

Designing Object-Oriented software
R. Wirfs-Brock / B. Wilkerson / L. Wiener; Prentice Hall; 1990

Object-Oriented Design
P. Coad / E. Yourdon; Yourdon Press; 1991.

Software Engineering: a Practitioners's Approach
R. Pressman; McGraw-Hill; 1892.

Software Engineering. Fourth edition.
Sommerville; Addision Wesley 1992.

Advances in Object-Oriented software Engineering
D. Mandrioli / B. Meyer; Prentice Hall; 1992.

Object-Oriented Software Construction
B. Mayer, Prentice Hall; 1988.

Pensamiento y Análisis del Comportamiento Objetual. (guía de curso)
Luis Nava; Unisys de México – Centro Educacional; 1994.

Reusability & Softwaare Construction: C & C++
J. Smith; Wiley; 1990.

Object-Oriented Systems Design and Integrated Approach.
E. Yourdon, Yourdon Press; 1994.

Designing the User Interface
Ben Shneiderman; Addison-Wesley; 1992.

Análisis y Diseño Orientado a Objetos
James Martin / J. Odell; prentice Halli; 1992.

Fundamentals of Software Engineering
Dino Mandrioli, Carlo Ghezzi y Mehdi Jazayeri, Prentice-Hall, 1ª Ed. 1991

Revistas


Nueve Reglas de Oro en el Diseño de Interfaces Humano-Computadora
Gonzáles J. Y Sandoval P., Soluciones Avanzadas, N° 10 Junio de 1994.


"X-Window y los Sistemas de Ventanas"
Guerrero, G; Soluciones Avanzadas. N° 2, enero- febrero 1993.


Borland C++ 3.1 Programación Orientada a Objetos
Ted Faison; 1993.

Folletos


Guía para el uso del Color
Hewlett Packard; 1992.


Borland C++ 4/4.5 Iniciación y Referencia
Luis Joyanes Aguilar; McGraw-Hill; 1996.

Notas de Elaboración:

Este Documento fue elaborado en una maquina Pentium MMX a 200 Mhz con Sistema Operativo Windows 95 versión 4.00.950ª Usando Microsoft Word 97, para preparar el documento se salvo en versión Word 2.0 para Windows. Para mejorar la Presentación se uso Corel Ventura Publisher Versión 5.0. Las figuras y diagramas fueron elaboradas con Corel Draw Versión 5.0. Para la Parte de Programación se uso Borland C++ Versión 4.02. Para la impresión se uso impresora de inyección de tinta Epson Stylus Color 400