5
2E;

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CONTADURIA Y ADMINISTRACION

"SISTEMA STD (DESARROLLO DE UN SISTEMA GRÁFICO DE SOPORTE A LA TOMA DE DECISIONES EMPLEANDO LA METODOLOGÍA ORIENTADA A OBJETOS)".

SEMINARIO DE INVESTIGACIÓN INFORMÁTICA
QUE PARA OBTENER EL TITULO DE:

LICENCIADO EN INFORMÁTICA

P R E S E N T A N:

LUIS AURELIO ELECHIGUERRA MENESES
ALBERTO FEDERICO LYNN
CARLOS DUGALDO NAVARRO ASCENCIO
SALVADOR VÁZQUEZ MEDINA

COORDINADOR DE SEMINARIO
L.A.E. LUIS EDUARDO LOPEZ CASTRO

México, D.F. 1996

Agradecimientos

Con Agradecimiento a la Facultad de Contaduría y Administración y gran afecto al grupo de catedráticos que coadyuvaron a nuestra formación profesional.

Un especial reconocimiento al Lic. Luis Eduardo López Castro por su aportación a nuestro desarrollo académico así como la conducción del presente trabajo.

A General Electric (Transportation Systems) por su gran apoyo y soporte.

A Cesar Hugo por su amistad y apoyo económico en la impresión de esta tesis.

En especial el mas profundo de los agradecimientos a la Familia Vázquez por su comprensión y apoyo en la realización de esta obra; de corazón GRACIAS.

A la Familia Federico gracias por continuar adelante y empujar a Alberto y a todos nosotros por seguir siempre adelante a pesar de las adversidades.

A las Familias Elechiguerra y Navarro por todo el apoyo vertido en nosotros a través de los años.

A nuestros compañeros de generación y en especial a Eduardo García Gómez por una amistad sincera e incondicional siempre estarás en nuestro corazón.

INTRODUCCIÓN 8

PROBLEMÁTICA ACTUAL 8

JUSTIFICACIÓN 10

OBJETIVOS 11

OBJETIVO GENERAL II

VENTAJAS DEL SISTEMA II

OBJETIVOS ESPECÍFICOS II

FUNDAMENTOS TEÓRICOS 13

DEFINICIÓN DEL TÉRMINO: "ORIENTADO A OBJETOS". 13

CARACTERÍSTICAS DE LOS OBJETOS. 13

DEFINICIÓN DE "DESARROLLO ORIENTADO A OBJETOS" 15

EL PRINCIPIO DEL DESARROLLO ORIENTADO A OBJETOS. 15

DESARROLLO DE LA METODOLOGÍA. 16

LOS TRES MODELOS DE LA METODOLOGÍA ORIENTADA A OBJETOS. 17

LOS ASPECTOS DEL TRABAJO ORIENTADO A OBJETOS 18

ABSTRACCIÓN. 18

ENCAPSULACIÓN. 19

COMBINACIÓN DE DATOS Y COMPORTAMIENTO 19

COMPARTICIÓN 19

ÉNFASIS EN LA ESTRUCTURA DE OBJETOS. 20

SINERGIA 20

1.- ANÁLISIS 22

I.I. MODELO DE OBJETOS. 23

I.I.I. IDENTIFICACIÓN DE CLASES DE OBJETOS. 24

I.I.2. PREPARACIÓN DEL DICCIONARIO DE DATOS 26

1.1.3. IDENTIFICACIÓN DE ASOCIACIONES 26
1.1.4. IDENTIFICACION DE ATRIBUTOS 27
1.1.5. REFINAMIENTO CON HERENCIA 29
1.1.6. PRUEBA DE LAS RUTAS DE ACCESO A DATOS 30
1.1.7. ITERACIÓN DEL MODELO DE OBJETOS 31
1.1.8. AGRUPACIÓN DE CLASES EN MÓDULOS 32

1.2. MODELO DINÁMICO 32
1.2.1. PREPARACIÓN DEL ESCENARIO 33
1.2.2. IDENTIFICACIÓN DE EVENTOS 34
1.2.3. PREPARACIÓN DEL SEGUIMIENTO DE UN EVENTO PARA CADA ESCENARIO 35
1.2.4. CONSTRUCCIÓN DE DIAGRAMAS DE ESTADO 36
1.2.5. RELACIONAR EVENTOS CON OBJETOS PARA VERIFICAR CONSISTENCIA 38

1.3. MODELO FUNCIONAL 39
1.3.1. IDENTIFICACIÓN DE VALORES DE ENTRADA Y SALIDA 40
1.3.2. CONSTRUCCIÓN DE DIAGRAMAS DE FLUJO DE DATOS 40
1.3.3. DESCRIPCIÓN DE FUNCIONES 42
1.3.4. IDENTIFICACIÓN DE RESTRICCIONES ENTRE OBJETOS 42
1.3.5. ESPECIFICACIÓN DE CRITERIOS DE OPTIMIZACIÓN 43

1.4. AGREGAR OPERACIONES 43
1.4.1. OPERACIONES DEL MODELO DE OBJETOS 43
1.4.2. OPERACIONES DE EVENTOS 43
1.4.3. OPERACIONES DE ACCIONES DE ESTADO Y ACTIVIDADES 44
1.4.4. OPERACIONES DE FUNCIONES 44
1.4.5. SIMPLIFICACION DE OPERACIONES 44

1.5. ITERACIÓN DEL ANÁLISIS 44
1.5.1. REFINAMIENTO DEL MODELO DE ANÁLISIS 45
1.5.2. RESTABLECIMIENTO DE REQUERIMIENTOS 45
1.5.3. FRONTERA ENTRE EL ANÁLISIS Y EL DISEÑO 46

2.- DISEÑO DEL SISTEMA 47

2.1. PREAMBULO DEL DISEÑO DE SISTEMAS 47
2.1.1. ORGANIZAR EL SISTEMA EN SUBSISTEMAS 48

CAPAS. 48
PARTICIONES 50
TOPOLOGÍA DEL SISTEMA. 50
2.1.2. IDENTIFICACIÓN DE CONCURRENCIAS 51
IDENTIFICACIÓN DE CONCURRENCIA INHERENTE. 51
2.1.3. ASIGNACIÓN DE SUBSISTEMAS A PROCESADORES Y TAREAS 51
ESTIMACIÓN DE LOS REQUERIMIENTOS DEL HARDWARE. 52
INTERCAMBIO DE HARDWARE - SOFTWARE. 52
ASIGNACIÓN DE TAREAS A PROCESADORES. 53
DETERMINACIÓN DE LA CONECTIVIDAD FÍSICA. 53
2.1.4. ADMINISTRACIÓN DE LOS ALMACENES DE DATOS. 54
VENTAJAS DE USAR UNA BASE DE DATOS. 54
DESVENTAJAS DE USAR UNA BASE DE DATOS. 54
2.1.5. MANEJO DE RECURSOS GLOBALES. 55
2.1.6. ELECCIÓN DE LA IMPLEMENTACIÓN DEL SOFTWARE DE CONTROL. 56
SISTEMAS DIRIGIDOS POR PROCEDIMIENTOS. 56
SISTEMAS DIRIGIDOS POR EVENTOS. 57
SISTEMAS CONCURRENTES 58
CONTROL INTERNO 58
2.1.7. MANEJO DE CONDICIONES DE LIMITE. 58
2.1.8. ASIGNAR PRIORIDADES INTERCAMBIABLES. 59

2.2. ARQUITECTURAS COMUNES. 60
2.2.1. TRANSFORMACIÓN EN LOTES 60
2.2.2. TRANSFORMACIÓN CONTÍNUA. 61
2.2.3. INTERFASES INTERACTIVAS. 62
2.2.4. SIMULACIÓN DINÁMICA 63
2.2.5. SISTEMAS DE TIEMPO REAL 64
2.2.6. MANEJADORES DE TRANSACCIONES 65

3.- DISEÑO DE OBJETOS 67

LOS PASOS DEL DISEÑO DE OBJETOS 68
3.1. COMBINACIÓN DE LOS TRES MODELOS. 69
INTERPRETACIÓN DEL DIAGRAMA DE ESTADOS. 69
INTERPRETACIÓN DEL DIAGRAMA DE FLUJO DE DATOS. 69

3.2. DISEÑO DE ALGORITMOS. 70
3.2.1. SELECCIÓN DE ALGORITMOS. 71
3.2.2. SELECCIÓN DE ESTRUCTURAS DE DATOS. 71
3.2.3. DEFINICIÓN DE CLASES Y OPERACIONES INTERNAS 72
3.2.4. ASIGNACIÓN DE RESPONSABILIDADES PARA OPERACIONES. 72

3.3. OPTIMIZACIÓN DEL DISEÑO. 73
3.3.1. AGREGAR ASOCIACIONES REDUNDANTES PARA ACCESOS EFICIENTES. 73
3.3.2. REORGANIZAR EL ORDEN DE EJECUCIÓN. 74
3.3.3. GUARDAR ATRIBUTOS DERIVADOS PARA EVITAR REPROCESAMIENTO. 74

3.4. IMPLEMENTACIÓN DE CONTROL. 75
3.4.1. LOS ESTADOS COMO UBICACIÓN DENTRO DE UN PROGRAMA. 76
3.4.2. MÁQUINAS DE ESTADO. 77
3.4.3. CONTROL COMO TAREAS CONCURRENTES. 77

3.5. AJUSTE DE LA HERENCIA 77
3.5.1. REORGANIZACIÓN DE CLASES Y OPERACIONES. 78
3.5.2. ABSTRACCIÓN DEL COMPORTAMIENTO COMÚN. 79
3.5.3. USO DE DELEGACIÓN PARA COMPARTIR LA IMPLEMENTACIÓN. 79

3.6. DISEÑO DE ASOCIACIONES. 80
ANÁLISIS DE LOS RECORRIDOS DE APLICACIONES. 80
ASOCIACIONES DE UN SENTIDO. 80
ASOCIACIONES DE DOS SENTIDOS. 81
ATRIBUTOS DE LIGAS. 81

3.7. REPRESENTACIÓN DE OBJETOS. 81

3.8 EMPACADO FÍSICO 82
3.8.1 OCULTAMIENTO DE INFORMACIÓN 82
COHERENCIA DE ENTIDADES. 83
CONSTRUCCIÓN DE MÓDULOS. 84
DOCUMENTACIÓN DE LAS DECISIONES DEL DISEÑO. 84

RESUMEN DE LA METODOLOGÍA 86

ANÁLISIS 86

DISEÑO DEL SISTEMA. 88

DISEÑO DE OBJETOS. 89

DEL DISEÑO A LA IMPLEMENTACIÓN 91

IMPLEMENTACIÓN USANDO UN LENGUAJE DE PROGRAMACIÓN 91

IMPLEMENTACIÓN USANDO UN SISTEMA MANEJADOR DE BASES DE DATOS 92

COMPARACIÓN ENTRE METODOLOGÍAS 93

ANÁLISIS Y DISEÑO ESTRUCTURADO (SA/SD) 93

COMPARACIÓN CON (OMT) 93

DESARROLLO ESTRUCTURADO DE JACKSON (JSD) 96

COMPARACIÓN CON OMT 97

LENGUAJES DE PROGRAMACIÓN ORIENTADOS A OBJETOS 99

CARACTERÍSTICAS DE LOS LENGUAJES ORIENTADOS A OBJETOS 99

HERENCIA MÚLTIPLE 99

LIBRERÍAS DE CLASES 99

EFICIENCIA 100

TIPIFICACIONES. FUERTES Y DÉBILES 101

MANEJO DE MEMORIA 101

ENCAPSULACIÓN 102

EMPAQUETADO 103

AMBIENTE DE DESARROLLO 104

METADATOS 105

CLASES PARAMETRIZADAS 106

ASERCIONES Y RESTRICCIONES. 106

PERSISTENCIA DE DATOS. 107

PANORAMA DE LENGUAJES ORIENTADOS A OBJETOS. 107

SMALLTALK. 107

C++ 109
EIFFEL 111
CLOS 112
COMPARACIÓN DE LOS LENGUAJES ORIENTADOS A OBJETOS. 114

APLICACIÓN DE LA METODOLOGÍA 116

Planteamiento del problema 116

Identificación de las clases de objetos: 118

Diccionario de datos 118

MODELO DE OBJETOS INICIAL 121

IDENTIFICACIÓN DE ATRIBUTOS 123
Refinamiento con herencia 123

MODELO DINÁMICO 124
CONSULTA 131

MODELO FUNCIONAL 132

OBTENCIÓN DE OPERACIONES 143

DISEÑO DE ALGORITMOS PARA CADA OPERACIÓN 144

BIBLIOGRAFÍA 163

ÍNDICE ANALÍTICO 167

INTRODUCCIÓN

PROBLEMÁTICA ACTUAL

En la actualidad, el proceso de toma de decisiones en las organizaciones se ha transformado en un factor vital no solamente para la buena marcha de un negocio, sino que además, debido a la creciente competencia de los mercados, la supervivencia de las empresas depende en buena medida de la calidad y oportunidad con la que las decisiones estratégicas del negocio son tomadas. Es por ello que resulta vital que las organizaciones tengan información oportuna y veraz del comportamiento de las operaciones que realizan.

Los sistemas de información se han vuelto cada vez más indispensables en cualquier organización, sin embargo el alcance que la mayoría de estos sistemas cubre es, en el mejor de los casos, el de satisfacer las necesidades de información de un área específica de la empresa. Esta información es transmitida en distintas formas a los ejecutivos encargados de tomar las decisiones, ya sea verbalmente, a través de reportes, resúmenes o gráficas. Pero en la mayoría de los casos, requiere un esfuerzo adicional de análisis de información para resumirla e incluso presentarla de manera que el tomador de decisiones pueda visualizarla y analizarla más fácilmente.

Algunos aspectos que dificultan este proceso de análisis y síntesis de información son:

Las organizaciones que cuentan con un área de Sistemas, están más preocupadas en darles mantenimiento continuo a los mismos, que en trabajar en nuevos sistemas que puedan agilizar la presentación de la información al tomador de decisiones. Por consiguiente, la oportunidad de la información se ve dificultada tanto por los retrasos en la captura de la información, como por la calidad y velocidad del mantenimiento a los sistemas de producción.

La mayoría de las organizaciones no marcan estándares en cuanto a sus herramientas informáticas, produciendo de esta manera una mezcla de

plataformas de hardware y software que en ocasiones resulta bastante complicada y costosa en lo que se refiere a su operación, integración y mantenimiento.
Las empresas en su gran mayoría han optado en automatizar paulatinamente sus áreas de información, dejando en segundo término la estandarización e institucionalización de sus datos. Las razones de esta automatización paulatina son generalmente presupuestales.

Las áreas de una empresa trabajan generalmente de manera aislada en cuanto a sistemas de información se refiere y de hecho definen necesidades específicas de información que en ocasiones no compaginan con los sistemas y requerimientos de información de otras áreas. El resultado de esta situación se ve reflejado en la pobre integración de los sistemas.

Resulta muy costoso para las empresas, tener personal exclusivamente dedicado al análisis, síntesis y presentación de información para la toma de decisiones.

JUSTIFICACIÓN

- La dificultad de ciertas organizaciones para tomar decisiones con base en un análisis de información consistente y confiable.

- El gasto de personal de las áreas responsables de analizar, sintetizar y presentar la información para la toma de decisiones, resulta bastante alto.

- Al existir una falta de estandarización de la información se presentan diferentes enfoques subjetivos en ocasiones, limitados en otras y por consiguiente erróneos.

- Cuando la información relevante para tomar una decisión no es proporcionada con la misma oportunidad por distintas áreas, el responsable de la decisión tiene que imaginarse esquemas que la sustenten. La mayoría de estos esquemas están basados en simple experiencia.

- Dado el dinamismo de cualquier negocio en la actualidad, factores que en su momento resultaban poco relevantes, se tornan importantes en cierto momento y no todas las empresas tienen la facilidad para identificarlos y analizarlos con oportunidad.

- El tomador de decisiones requiere de mucho tiempo adicional para analizar la información que le es presentada. En ocasiones, aunque la información es resumida, no puede ni siquiera revisarse en su totalidad.

- Para que el tomador de decisiones pueda darse una idea del comportamiento global de la empresa, requiere en ocasiones, de tiempo adicional para que las áreas responsables le proporcionen información histórica y otras muchas veces se tiene que echar mano de documentos o informes anteriores.

OBJETIVOS

OBJETIVO GENERAL

Apoyar a la toma de decisiones con una herramienta altamente paramétrica, que favorezca a la estandarización de la información, de manera tal que ésta pueda ser analizada y visualizada ágil y fácilmente por los responsables de la empresa.

VENTAJAS DEL SISTEMA

- Unificación de la información como punto de comparación entre áreas.
- La visualización del estado de distintos niveles de las áreas contenidas en el sistema.
- Mejoramiento substancial en la oportunidad de la información.
- La interfase gráfica evita que el usuario tenga que teclear instrucciones o aprender comandos, además de ser completamente compatible con "Microsoft Windows", lo cual reduce la curva de aprendizaje del usuario final.
- Debido al diseño paramétrico del sistema, éste no está ligado a ninguna empresa en particular ni a alguna rama de la industria, sino que es aplicable a todo tipo de empresa y al tipo de información que ésta maneje.

OBJETIVOS ESPECÍFICOS

a) Mejorar la consistencia de la información para la toma de decisiones y la confiabilidad de los análisis de la misma.

b) Reducción de costos en áreas encargadas del análisis, síntesis y presentación de información para la toma de decisiones.
c) Unificación de enfoques, de análisis y estandarización de la información de la empresa.

d) Notificar al tomador de decisiones de manera oportuna e incluso anticipada, del comportamiento "anormal" de factores que en un momento se declararon como irrelevantes y que por su tendencia puedan llegar a ser de vital importancia o representen un problema a futuro de no atenderse con oportunidad.

e) Reducción substancial de tiempo requerido por el tomador de decisiones para analizar la información.

f) Facilitar la consulta y el comportamiento de información histórica.

g) Mejorar la calidad de las decisiones combinando la oportunidad de la información con la experiencia de los responsables de ésta función.

FUNDAMENTOS TEÓRICOS

El diseño y modelado Orientado a Objetos es una nueva forma de plantear los problemas empleando modelos organizados alrededor de conceptos del mundo real. La construcción básica es llamada Objeto, el cual combina una estructura de datos y un comportamiento en una sola entidad. Explicado de manera breve, esta técnica parte de la construcción de un Modelo de Análisis en el cual se abstraen aspectos esenciales del dominio de una aplicación específica sin considerar implementaciones eventuales. El modelo contiene los objetos identificados en el dominio de la aplicación, incluyendo una descripción de las propiedades de los objetos y de su comportamiento. Posteriorménte se llevan a cabo las decisiones con respecto al diseño y se agregan al modelo de análisis los detalles necesarios para describir y optimizar la implementación. Finalmente, el modelo del diseño es implementado en un lenguaje de programación, manejador de base de datos, o en una plataforma de hardware.

DEFINICIÓN DEL TÉRMINO: "ORIENTADO A OBJETOS".

El término "Orientado a Objetos" significa organizar el software como una colección de objetos individuales que incorporan, estructuras de datos y un comportamiento específico.

Los cuatro aspectos o características generalmente reconocidos por un enfoque orientado a objetos son: Identidad, Clasificación, Polimorfismo y Herencia.

CARACTERÍSTICAS DE LOS OBJETOS.

IDENTIDAD.- Significa que los datos pueden cuantificarse en entidades individuales y distinguibles llamadas objetos. El párrafo de un documento, una ventana en cierta aplicación, la reina blanca en el juego de ajedrez son ejemplos de objetos. Cada objeto tiene su propia identidad inherente, lo cual

significa que dos objetos resultan distintos, aún en el caso de que los valores de sus atributos fueran idénticos.

En el mundo real un objeto simplemente existe, pero dentro de un lenguaje de programación, cada objeto posee un manejo único, por medio del cual es referenciado. El método para hacer referencia a un objeto puede variar, pero las referencias a objetos son uniformes e independientes del contenido de los mismos.

CLASIFICACIÓN.- Significa que los objetos con la misma estructura de datos (atributos) y comportamiento (operaciones) se agrupan en clases. Una clase describe un posible conjunto infinito de objetos individuales. Se dice entonces que cada objeto es una instancia de su clase. Cada instancia tiene sus propios valores para cada atributo, pero comparte los nombres de los atributos y operaciones con otras instancias de su clase. La forma en que los objetos se agrupen en clases es arbitraria y depende de la aplicación.

POLIMORFISMO.- Significa que la misma operación puede comportarse de manera diferente en clases distintas. Una operación es una acción o transformación que ejecuta un objeto o de la cual este es sujeto.

Una implementación específica de una operación en una clase es llamada método. Debido a que un operador orientado a objetos es polimórfico, puede haber más de un método que la implemente.

En el mundo real, una operación es simplemente una abstracción de comportamientos análogos entre diferentes tipos de objetos. Cada objeto "sabe como ejecutar sus propias operaciones".

HERENCIA.- Este término significa que las clases pueden compartir atributos y operaciones entre sí, basados en una relación jerárquica. Una clase puede ser definida de una manera muy amplia y ser refinada posteriormente en "subclases" más detalladas.

Cada subclase incorpora o hereda, todas las propiedades de su "superclase" y agrega sus propiedades particulares.

La posibilidad de agrupar propiedades comunes de diversas clases en una superclase común y heredar las propiedades de esta, reduce substancialmente la repetición dentro de diseños y programas, y constituye una de las principales ventajas de un sistema orientado a objetos.

DEFINICIÓN DE "DESARROLLO ORIENTADO A OBJETOS"

Es una nueva forma de plantear el desarrollo del software basado en abstracciones que existen en el mundo real. La esencia del desarrollo orientado a objetos es la identificación y organización de conceptos del dominio de una aplicación, en lugar de su representación final en un lenguaje de programación, sea este último, orientado a objetos o no.

Frederick Brooks dice *"La parte más difícil en el desarrollo de software es la manipulación de su esencia, debido a la complejidad inherente del problema, mas que de los accidentes de su conversión a un lenguaje de programación en particular, las cuales se deben a las imperfecciones temporales de nuestras herramientas".*

EL PRINCIPIO DEL DESARROLLO ORIENTADO A OBJETOS.

Actualmente, la mayor parte del esfuerzo dentro de la comunidad "orientada a objetos", se ha enfocado a resultados o productos en los lenguajes de programación. El énfasis de la literatura actual está en la implementación, mas que en el análisis y diseño.

Los lenguajes de programación orientados a objetos son muy útiles para remover restricciones producidas por la inflexibilidad de los lenguajes tradicionales de programación. Sin embargo, en cierto sentido, este énfasis significa un retroceso en la ingeniería de software, por estar enfocada excesivamente en mecanismos de implementación, antes que subordinar el proceso de razonamiento que lo soporta. Las fallas del diseño que aparecen durante la implementación, son más costosas de arreglar que aquellas que son encontradas antes de la misma.

El enfocarse en cuestiones de implementación, al principio restringe las opciones del diseño y conducen a una calidad inferior del producto. Solo cuando los conceptos inherentes de una aplicación son identificados, organizados y comprendidos, los detalles de las estructuras de datos y funciones pueden ser dirigidos de manera efectiva.

El desarrollo orientado a objetos es un proceso conceptual independiente del lenguaje de programación hasta las etapas finales. Es fundamentalmente una nueva forma de pensar y no una técnica de programación.

DESARROLLO DE LA METODOLOGÍA.

Consiste en la construcción de un modelo del dominio de una aplicación, para posteriormente, agregar al mismo los detalles de implementación durante el diseño del sistema. A este enfoque se le llama Técnica de Modelado de Objetos (OMT). La metodología tiene las siguientes fases:

1. **ANÁLISIS**. Partiendo de la exposición de un problema, el analista construye un modelo de la situación real, mostrando sus propiedades importantes. El modelo del análisis es una abstracción concisa y exacta de lo que el sistema debe hacer, no de como lo hará. Los objetos en el modelo deben ser conceptos del dominio de la aplicación y no conceptos de implementación tales como estructuras de datos. Un buen modelo puede ser comprendido y discutido por expertos en la aplicación los cuales no necesariamente son programadores. El modelo de análisis no debe contener ninguna decisión de implementación.

2. **DISEÑO DEL SISTEMA**. El diseñador del sistema en esta etapa, toma las decisiones de alto nivel acerca de la arquitectura total del sistema. Durante esta etapa, el sistema a desarrollar es organizado en subsistemas basados en la estructura del análisis y la arquitectura propuesta. El diseñador debe decidir que características de rendimiento deben optimizarse y se debe elegir una estrategia para atacar el problema y reservar recursos tentativos.

3. **DISEÑO DE OBJETOS**. El analista construye un modelo del diseño basado en el modelo del análisis, pero conteniendo detalles de la implementación. Estos detalles deben concordar con la estrategia

establecida durante el diseño del sistema. El enfoque del diseño de objetos es la estructura de datos y los algoritmos necesarios para implementar cada clase. Las clases de objetos identificadas en el análisis continúan siendo significativas, pero en esta fase, son complementadas con las estructuras de datos y los algoritmos elegidos para optimizar importantes medidas de desempeño.

4. **IMPLEMENTACIÓN**. Las clases de objetos y sus relaciones desarrolladas durante el diseño de objetos son finalmente traducidas a un lenguaje de programación, base de datos o implementación de hardware. La programación debe ser una parte relativamente menor y mecánica del ciclo de desarrollo, esto es debido a que las decisiones importantes deben realizarse durante el diseño. El lenguaje utilizado influye hasta cierto punto en decisiones de diseño, pero el diseño no debe depender de detalles específicos del lenguaje de programación.

Los conceptos que brinda la metodología de desarrollo orientada a objetos pueden ser aplicados a través de todo el ciclo de desarrollo del sistema, desde el análisis hasta la implementación. Las mismas clases pueden conservarse de fase a fase sin cambio en la notación, aunque cabe recordar que adquirirán detalles adicionales de implementación en etapas posteriores.

Algunas clases no son parte del análisis, pero son introducidas como parte del diseño o de la implementación. Por ejemplo, estructuras de datos tales como árboles, tablas de búsqueda y listas ligadas difícilmente se encuentran en el mundo real, pero se introducen como soporte a algoritmos particulares durante el diseño.

LOS TRES MODELOS DE LA METODOLOGÍA ORIENTADA A OBJETOS.

La metodología orientada a objetos utiliza tres tipos de modelos para describir un sistema : El modelo de objetos, que describe los objetos en el sistema y sus relaciones; el modelo dinámico, que describe las interacciones entre los objetos en el sistema; y el modelo funcional, que describe las transformaciones de los datos en el sistema. Cada modelo es aplicable durante todas las etapas de desarrollo y va adquiriendo detalles de implementación

conforme avanza el desarrollo. Una descripción completa de un sistema requiere de los tres modelos.

El modelo de objetos describe la estructura estática de los objetos en un sistema, así como sus relaciones. El modelo de objetos se conforma de diagramas de objetos. Un diagrama de objetos es una gráfica cuyos nodos representan clases de objetos, y cuyos arcos representan las relaciones entre las clases.

El modelo dinámico describe los aspectos de un sistema que cambia en el tiempo. Es utilizado para especificar e implementar aspectos de control en un sistema. El modelo dinámico está conformado de diagramas de estado. Un diagrama de estado es una gráfica cuyos nodos representan estados, y cuyos arcos representan transiciones de un estado a otro causadas por eventos.

El modelo funcional describe las transformaciones de los valores de los datos dentro de un sistema. Se conforma de diagramas de flujo de datos. Un diagrama de flujo de datos es una gráfica cuyos nodos representan procesos, y cuyos arcos representan flujos de datos.

LOS ASPECTOS DEL TRABAJO ORIENTADO A OBJETOS

ABSTRACCIÓN.

La abstracción consiste en enfocarse en los aspectos esenciales inherentes de una entidad, ignorando sus propiedades accidentales o circunstanciales. En el desarrollo de un sistema, el concepto de abstracción significa enfocarse en lo que un objeto es y hace, antes de decidir como debe ser implementado. El uso de la abstracción durante el análisis significa ocuparse solamente de los conceptos del dominio de la aplicación, y no tomar decisiones de diseño ni de implementación antes de que el problema sea bien comprendido. El uso adecuado de la abstracción permite que un mismo modelo pueda ser utilizado para el análisis, el diseño de alto nivel, la estructura del programa e incluso, la documentación.

ENCAPSULACIÓN.

La encapsulación (u ocultamiento de información) consiste en la separación de los aspectos externos de un objeto que son accesibles por otros objetos, de los detalles de implementación internos del objeto que están ocultos para otros objetos. La encapsulación permite modificar la implementación de una clase sin afectar a los clientes de dicha clase. De esta manera, puede modificarse la implementación para incrementar su desempeño, corregir algún problema, consolidar código, o portar la aplicación a un sistema diferente. La encapsulación no es exclusiva de los lenguajes orientados a objetos, pero la capacidad de combinar estructuras de datos y comportamiento en una sola entidad hace que la encapsulación resulte más clara y poderosa, que en lenguajes convencionales los cuales separan la estructura de datos del comportamiento.

COMBINACIÓN DE DATOS Y COMPORTAMIENTO

Al hacer una llamada a una operación no se necesita considerar cuántas implementaciones de la operación existen. El polimorfismo de un operador facilita la decisión de la implementación a utilizar en una llamada a la jerarquía de clases. Por ejemplo, en un lenguaje no orientado a objetos, el código que despliega figuras geométricas debe distinguir cada tipo de figura que será desplegada, como pueden ser un polígono o un círculo y llamar al procedimiento adecuado para desplegar cada figura. En un programa orientado a objetos, se llamaría simplemente a una operación denominada "dibuja"; la decisión del procedimiento a utilizar para cada figura, es hecha de forma implícita por cada objeto, basándose en su clase. Con ello, resulta innecesario repetir la decisión del procedimiento a utilizar cada vez que la operación es llamada en el programa. El mantenimiento al programa resulta mucho más sencillo, debido a que el código que hace la llamada no necesita ser modificado cada vez que se agrega una nueva clase.

COMPARTICIÓN

Las técnicas orientadas a objetos permiten la compartición de atributos y operaciones a distintos niveles de clases. La herencia tanto de las estructuras

de datos como del comportamiento permiten que estructuras comunes sean compartidas entre subclases similares, sin que esto implique redundancia. La compartición de código empleando la herencia es una de las principales ventajas de los lenguajes orientados a objetos. Aún más importante que el ahorro en código, es la claridad conceptual resultante que permite identificar aquellas operaciones que realizan lo mismo. Esto reduce del número de casos distintos que deben analizarse y comprenderse.

Los desarrollos orientados a objetos no solo permiten que la información se comparta dentro de una aplicación, sino que también ofrecen la posibilidad de reutilizar diseños y código en futuros proyectos. Aunque esta posibilidad ha sido enfatizada en exceso como justificación de la tecnología orientada a objetos, su desarrollo proporciona herramientas tales como la abstracción, encapsulación y herencia para construir bibliotecas de componentes reutilizables.

ÉNFASIS EN LA ESTRUCTURA DE OBJETOS.

La tecnología orientada a objetos hace énfasis en la especificación de lo que un objeto es, mas que en como es utilizado. Los usos de un objeto dependen mucho de los detalles de la aplicación y frecuentemente cambian durante el desarrollo. Conforme los requerimientos cambian, las características de los objetos se vuelven más estables que las formas de manejarlos. El desarrollo orientado a objetos enfatiza más la estructura de datos, y menos la estructura de los procedimientos, a diferencia de las metodologías tradicionales de descomposición funcional. Con respecto a este énfasis, el desarrollo orientado a objetos es similar a las técnicas de modelaje de bases de datos, aunque a diferencia de éste, agrega el concepto de comportamiento dependiente de la clase.

SINERGIA

Identidad, clasificación, polimorfismo y la herencia caracterizan a los principales lenguajes orientados a objetos. Cada uno de estos conceptos puede ser utilizado en forma aislada, pero juntos se complementan una a otro. Los beneficios de un enfoque orientado a objetos son mayores de lo que parece a

simple vista. El mayor énfasis en las propiedades de los objetos obliga al desarrollador de sistemas a pensar y definir más cuidadosamente y con mayor profundidad en lo que un objeto es y en lo que hace, con el resultado de que el sistema generalmente es más claro, de uso más general, y más robusto de lo que sería si el énfasis se hiciera únicamente en los datos y operaciones.

1.- ANÁLISIS

El primer paso en la metodología orientada a objetos se refiere a concebir un modelo del mundo real que sea preciso, conciso, comprensible y correcto. El propósito de un análisis orientado a objetos es modelar un sistema de tal manera que este pueda ser bien comprendido. Para hacer esto se deben examinar los requerimientos, analizar sus implicaciones y reestablecerlas de forma rigurosa. Primeramente se deben de abstraer los aspectos importantes del mundo real y dejar los pequeños detalles para después.

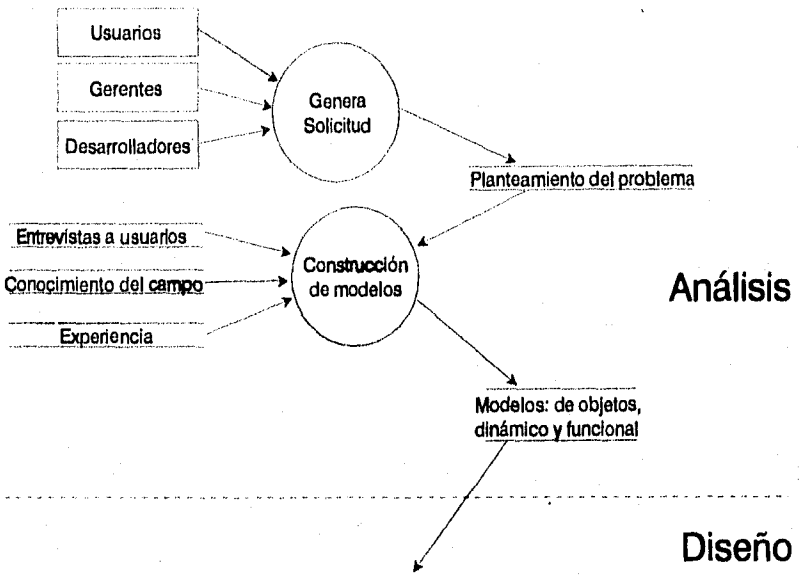
El resultado del análisis debe ser el entendimiento correcto de **QUE** hacer con el sistema, sin importar **COMO** debe de hacerse.

Para realizar el análisis orientado a objetos, es necesario crear 3 modelos que describen un sistema:

MODELO DE OBJETOS.- Describe la estructura estática de los objetos que participan en un sistema así como sus relaciones. Contiene los diagramas de objetos que son gráficas cuyos nodos representan clases de objetos y cuyos arcos representan relaciones entre clases.

MODELO DINÁMICO.- Nos muestra los aspectos de un sistema que cambia sobre el tiempo y los eventos que generan estos cambios. Es utilizado para especificar e implementar aspectos de control en un sistema. El modelo dinámico contiene diagramas de estado, los cuales son gráficas cuyos nodos representan estados y cuyos arcos representan transiciones entre estados causadas por eventos.

MODELO FUNCIONAL.- Describe las transformaciones de los valores de los datos dentro de un sistema. El modelo funcional contiene diagramas de flujo de datos, los cuales representan el procesamiento de datos. Dichos diagramas son gráficas cuyos nodos representan procesos y cuyos arcos representan flujo de datos.



Resumen del proceso de análisis

1.1. MODELO DE OBJETOS.

El primer paso en el análisis de requerimientos es construir un modelo de objetos; el modelo de objetos describe las clases de los objetos y sus relaciones entre ellos.

La estructura estática reflejada por el modelo de objetos normalmente esta mejor definida, es menos dependiente de los detalles de la aplicación, es más estable a medida que la solución evoluciona y es más fácil de entender.

En la construcción de este modelo, en principio, se deben identificar las clases, las asociaciones y como éstas afectan la estructura completa y al enfoque del problema, después se agregan atributos para hacer una descripción posterior de la red básica de clases y asociaciones. Después se combinan y organizan las clases utilizando la herencia; una vez terminado lo anterior se agregan operaciones a las clases para poder construir los modelos

dinámico y funcional y debido a que las operaciones modifican a los objetos, no pueden ser completamente especificadas hasta que su dinámica y funcionalidad estén comprendidas.

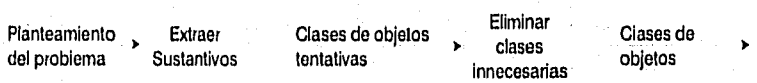
Los siguientes pasos deben ser desarrollados en la construcción de un modelo de objetos:

1.1.1. Identificación de las clases de objetos.
1.1.2. Preparación del diccionario de datos.
1.1.3. Identificación de asociaciones entre objetos.
1.1.4. Identificación de atributos y ligas de objetos.
1.1.5. Organización y simplificación de clases de objetos empleando la herencia.
1.1.6. Verificación de las rutas de acceso para consultas exitosas.
1.1.7. Iteración y refinación del modelo.
1.1.8. Agrupación de las clases en módulos.

1.1.1. IDENTIFICACIÓN DE CLASES DE OBJETOS.

El primer paso a desarrollar en la construcción del modelo de objetos, consiste en determinar las clases de objetos que son relevantes para la aplicación. Las clases de objetos pueden incluir tanto entidades físicas como entidades conceptuales, siempre y cuando tengan algún objetivo dentro del dominio de la aplicación, evitando al máximo aquellas relacionadas con la implementación del sistema.

Lo anterior se puede realizar listando las clases de objetos candidatas encontradas en la descripción escrita del problema, las clases a menudo corresponden a sustantivos de la descripción del problema.



Identificación de clases de objetos

Resulta normal que no todas las clases de objetos identificadas durante esta etapa serán necesarias o relevantes dentro del dominio de la aplicación. Para descartar estas clases del modelo de objetos, se pueden aplicar los siguientes criterios :

Clases Redundantes.- Ocurre cuando dos clases expresan la misma información. El nombre que resulte más descriptivo dentro del dominio de la aplicación debe ser mantenido.

Clases Irrelevantes.- Ocurre cuando una clase tiene poco o nada que ver con el problema. Estas clases deben ser eliminadas, asegurándose de que no resulten importantes en otro contexto de la aplicación.

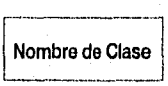
Clases Vagas.- Ocurre cuando una clase no es específica. Algunas clases tentativas pueden tener una pobre definición de sus límites, o muy extensas en su alcance.

Atributos.- Los nombres que describen objetos individuales deberán ser eliminados como clases y ser planteados como atributos de alguna otra.

Operaciones.- Si un nombre describe una operación que es aplicada a un objeto y que no se maneja por si misma, entonces no es una clase.
Papeles.- El nombre de una clase debe reflejar su naturaleza intrínseca y no el papel que juegue en una asociación.

Construcciones de implementación.- Aquellas construcciones ajenas al mundo real deben ser eliminadas del modelo de análisis, ya que no son necesarias en esta parte del análisis.

Notación para clase:



Las clases se denotan por rectángulos

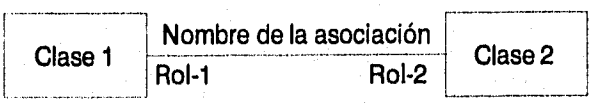
1.1.2. PREPARACIÓN DEL DICCIONARIO DE DATOS

En esta etapa se deben describir de forma precisa cada una de las clases de objetos a través de textos narrativos. En la descripción de las clases se debe incluir :

- El alcance de cada clase dentro del contexto del problema.
- La existencia de cualquier supuesto o restricciones en su uso.
- Las asociaciones, atributos y operaciones posibles de cada clase.

1.1.3. IDENTIFICACIÓN DE ASOCIACIONES

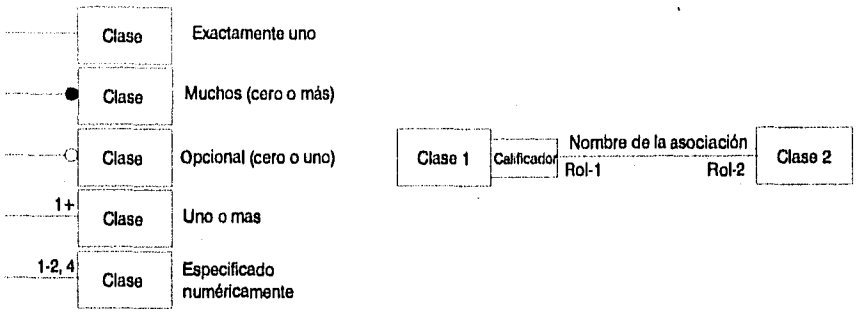
Cualquier dependencia entre dos o más clases es una asociación; la referencia desde una clase a otra, también es una asociación. Las asociaciones muestran dependencia entre clases en el mismo nivel de abstracción de las mismas clases.



Las asociaciones se denotan por una línea que una a las clases

Las asociaciones pueden ser implementadas de diferentes formas, pero las decisiones de cada aplicación deben ser mantenidas fuera del modelo de análisis para guardar la libertad del diseño.

La mayoría de las asociaciones pueden deducirse del planteamiento del problema (verbos que indican ubicación física, acciones, comunicación, pertenencia, etc.).
Debe especificarse la multiplicidad en las asociaciones y tratar de calificarlas cuando las relaciones no sean uno a uno.



Notación de multiplicidad y asociaciones calificadas

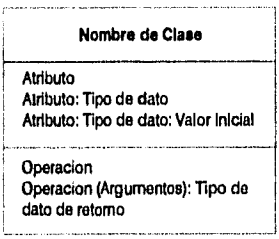
1.1.4. IDENTIFICACIÓN DE ATRIBUTOS

Podemos definir a los atributos como propiedades o cualidades de cada objeto, como por ejemplo: nombre, peso, etc. Se deben utilizar asociaciones para mostrar cualquier relación entre dos objetos.

Se deben considerar solamente los atributos que están directamente relacionados con la aplicación; no exagere al descubrir los atributos de los objetos y evite durante el análisis aquellos que sean únicamente derivados de la implementación.

Durante el análisis, se deben evitar los atributos que sirvan solo para la implementación y es muy importante darles a los atributos que permanezcan, un nombre con un significado correcto.
Los atributos derivados (como por ejemplo edad, que se deriva de la fecha de nacimiento y la actual) deberán ser omitidos o claramente marcados. Los atributos derivados, como pueden ser las asociaciones, resultan útiles al abstraer propiedades significativas de la aplicación completa, pero no se deben confundir con los atributos base, ya que estos últimos definen el estado del objeto.

Se deben identificar también, los atributos de la ligas, los cuales son exclusivos de las relaciones entre objetos, en lugar de ser propiedades de un objeto individual.



Notación de Clase con atributos

Para eliminar los atributos incorrectos o innecesarios se pueden seguir los siguientes criterios:

Objetos.- Si el atributo es muy independiente de la entidad, sería mejor tomarlo como valor o como otro objeto. En algunos contextos lo que es considerados un atributo, para otros puede ser un objeto, dependiendo de la importancia y dependencia dentro de la aplicación.

Calificadores.- Si el valor de un atributo depende de un contexto particular, debe considerarse como calificador. Es decir, si el atributo depende de una situación pero varia en otra, es un calificador.

Nombres.- Cuando un objeto es mejor referenciado por otro atributo que por su nombre, dicho nombre, realmente esta calificando a una asociación y no forma parte de los atributos de una clase. Un nombre es un atributo de un objeto cuando no depende de un contexto, particularmente cuando no necesita ser único.

Identificadores.- Los lenguajes orientados a objetos incorporan la noción de identificador de objetos para eliminar ambigüedades. Estos tipos de identificadores no deben ser listados en los atributos, solo deben mencionarse cada uno de los que existan en el dominio de la aplicación. Por ejemplo, un

número de cuenta bancario de un cliente es un atributo correcto. En contraste el atributo numero de transacción no debe listarse como atributo de objeto aunque es conveniente generar dicho número, pero hasta la implementación.

Atributos de la liga.- Si la propiedad depende de la presencia de una liga, entonces la propiedad es un atributo de una liga y no de uno de los objetos relacionados. Los atributos de liga son generalmente obvios en asociaciones muchos-a-muchos y no pueden ser incluidas en ninguna de las clases asociadas debido a su multiplicidad.

Valores Internos.- Si el atributo describe el estado interno de un objeto y éste es invisible fuera de el objeto, éste debe eliminarse del análisis.

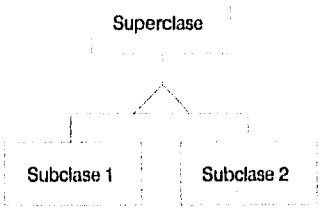
Detalles finos.- Omitir los atributos sin importancia que estén lejos de afectar la mayoría de las operaciones de la aplicación.

Atributos discordantes.- Si un atributo parece ser completamente distinto de otros atributos, esto puede indicar que la clase debe dividirse en dos clases distintas.

1.1.5. REFINAMIENTO CON HERENCIA

El siguiente paso en la construcción del modelo de objetos, consiste en organizar las clases utilizando el recurso de la herencia para compartir estructuras comunes. La herencia puede añadirse en dos direcciones: generalizando aspectos comunes de las clases existentes hácia una superclase o refinado las clases existentes hacia clases especializadas.

Se puede descubrir herencia de abajo hacia arriba encontrando clases con atributos similares, asociaciones u operaciones. Para cada generalización se define una superclase que comparte las características comunes.



Notación para Herencia

Las especializaciones de arriba hacia abajo se obtienen frecuentemente a partir del dominio de una aplicación. Se debe evitar un refinamiento excesivo. Si las especializaciones propuestas son incompatibles con una clase existente, dicha clase puede estar formulada de una manera inadecuada.

La herencia múltiple puede utilizarse para incrementar la posibilidad de compartir atributos, pero solo si es necesario, dado que con ello se incrementa la complejidad conceptual y de implementación.

Al utilizar herencia múltiple, es posible designar una superclase primaria la cual suministra la mayor parte de la estructura y comportamiento heredados.

1.1.6. PRUEBA DE LAS RUTAS DE ACCESO A DATOS

En esta etapa, el modelo de objetos debe revisarse para garantizar que las situaciones que ocurren en el mundo real sean satisfechas de manera consistente y confiable en el modelo.

Por ejemplo: Si en el mundo real, a un valor único proporcionado se produce un resultado único, entonces el modelo deberá garantizar que existe una ruta de acceso para obtener el valor esperado.

Si algo que parece simple en el mundo real, resulta complejo en el modelo, es que probablemente algo haya sido olvidado o definido incorrectamente en alguna de las etapas de la construcción del modelo de objetos.

1.1.7. ITERACIÓN DEL MODELO DE OBJETOS

El modelo de objetos difícilmente resulta correcto en la primera ocasión. Si alguna deficiencia es encontrada, se debe regresar a las primeras etapas para corregirlo.

Es importante recordar que algunos detalles solo pueden llegar después de completar los modelos dinámicos y funcional.

Algunas señales de objetos olvidados son:

- Atributos y operaciones desiguales en una clase. Es necesario dividir la clase en dos o mas para que cada parte sea coherente.
- Dificultad para generalizar de forma limpia; es probable que una clase desempeñe dos papeles, en cuyo caso deberá ser dividida.
- Operaciones que afectan a clases erróneas o indefinidas.
- Asociaciones duplicadas con el mismo nombre y propósito.
- Un papel forma substancialmente la semántica de una clase. En este caso, la asociación debe convertirse en una clase.

Algunas señales de clases innecesarias son:

- Falta de atributos, operaciones y asociaciones en una clase.

Algunas señales de asociaciones olvidadas son:

- Rutas de acceso olvidadas para ciertas operaciones. Se deben agregar nuevas asociaciones para que las peticiones de datos sean contestadas.

Algunas señales de asociaciones innecesarias son.

- Información redundante en las asociaciones. Se deben remover asociaciones que no agreguen nueva información o se deben marcar como asociaciones derivadas.
- Carencia de operaciones que recorran una asociación. Si no existen operaciones que utilicen alguna trayectoria, puede ser que la información no se necesite y por lo tanto la asociación tampoco.

Algunas señales de una colocación incorrecta de asociaciones son:

- Nombres de papeles que son demasiado amplios o demasiado limitados para sus clases. En este caso la asociación debe moverse hacia arriba o hacia abajo en la jerarquía de clases.

Algunas señales de una incorrecta colocación de atributos.

- Necesidad de accesar a un objeto por medio del valor de alguno de sus atributos. Se debe considerar en este caso el uso de una asociación calificada.

1.1.8. AGRUPACIÓN DE CLASES EN MÓDULOS

El último paso para completar el modelo de objetos es agrupar las clases en módulos; con la finalidad de contar con un tamaño más o menos uniforme que permita dibujar los diagramas en hojas divididas pero agrupadas en módulos. Cuando hay dos clases fuertemente acopladas éstas se pueden poner juntas en una misma hoja, aunque a veces hay que hacer cortes de página arbitrarios. Cuando hay varias clases de tamaño variable y por conveniencia se desean mostrar en hojas separadas, se podrán hacer, pero si existe una conexión entre dos o más clases, éstas deberán mostrarse en la medida de lo posible en la misma hoja. Cuando no sea posible y se deba mostrar en hojas separadas, se puede hacer un puente mostrando la conexión entre dos hojas, estos tipos de puentes se deben evitar o minimizar, tampoco se deben hacer líneas cruzadas entre clases o grupos de clases.

1.2. MODELO DINÁMICO

El modelo dinámico muestra el comportamiento del sistema en función del tiempo, así como de los objetos contenidos en él. Comienza con un análisis dinámico buscando eventos externamente visibles, estímulos y respuestas.

Posteriormente sumariza secuencias de eventos permitidos para cada objeto con un diagrama de estado. Los algoritmos no son relevantes durante el

análisis si existen manifestaciones no apreciables. La razón se basa en que los algoritmos son parte de la implementación.

El modelo dinámico es importante para sistemas interactivos. Para la mayoría de los problemas, las conexiones lógicas dependen de secuencias de iteraciones en sistemas de tiempo real, por lo tanto, se tienen requerimientos específicos de tiempo en iteraciones que deben ser consideradas durante el análisis.

Primero se preparan escenarios de diálogos típicos. Aunque éstos escenarios pueden no cubrir cualquier contingencia, aseguran al menos, que las interacciones comunes no sean desapercibidas.

Se deben extraer eventos de los escenarios, esto regularmente se hace, primero para una mejor identificación y posteriormente se asigna cada evento a su objeto destino. Se organizan las secuencias de eventos en un diagrama de estado y finalmente se comparan diagramas de estado para objetos distintos para estar seguros de que los eventos intercambiados por ellos sean iguales.

El conjunto de diagramas de estado resultante constituye el modelo dinámico.

En resumen, los siguientes pasos deben de ser desarrollados en la construcción del modelo.

1.2.1. Preparación de los escenarios para secuencias de interacciones típicas.
1.2.2. Identificación de los eventos entre objetos.
1.2.3. Preparación del seguimiento de un evento para cada escenario.
1.2.4. Construcción de el diagrama de estado.
1.2.5. Relacionar eventos con objetos para verificar consistencia.

1.2.1. PREPARACIÓN DEL ESCENARIO

En esta fase se deben preparar uno o más diálogos típicos entre el usuario y el sistema para obtener el comportamiento esperado del sistema. Estos escenarios muestran las interacciones principales, formatos de presentación externos e intercambios de información.

Es recomendable desarrollar el modelo dinámico por medio de escenarios, en lugar de describir el modelo general directamente, para asegurar que los pasos importantes no queden desapercibidos y que el flujo de las interacciones sea el correcto.
En ocasiones el planteamiento del problema describe toda la secuencia de interacciones pero en la mayoría de los casos el formato de las interacciones debe de inventarse.

Primeramente hay que preparar escenarios para casos normales, en los cuales no existen entradas de datos inusuales ni condiciones de error. Posteriormente, se deben considerar los casos especiales, tales como secuencias de entrada omitidas o valores mínimos y máximos. Finalmente se deben considerar interacciones que pueden extenderse sobre interacciones básicas, tales como peticiones de ayuda o consultas de estado.

Un escenario es una serie de eventos. Un evento ocurre cuando se produce un intercambio de información entre un objeto del sistema y un agente externo, como el usuario, un sensor o alguna tarea.

1.2.2. IDENTIFICACIÓN DE EVENTOS

En esta etapa se examinan los escenarios para identificar todos los eventos externos. Los eventos incluyen todas las señales, entradas, decisiones, interrupciones, transiciones y acciones de los usuarios o dispositivos externos. Las etapas de procesamiento internas no son eventos, excepto por los puntos de decisión que interactúan con el mundo externo.
Los escenarios se utilizan para encontrar eventos normales, aunque no se deben olvidar las condiciones de error y los eventos inusuales.

Una acción de transmisión de información hecha por un objeto es un evento. La mayoría de las operaciones e interacciones de objeto a objeto corresponden a eventos.

Los eventos que tienen el mismo efecto en el flujo de control se agrupan bajo un solo nombre, aún en el caso de que los valores de los parámetros difieran.

Debe decidirse cuando las diferencias en los valores son lo suficientemente importantes para ser distinguidas dependiendo de la aplicación. En ocasiones se puede construir un diagrama de estado antes de clasificar todos los eventos; algunas diferencias entre eventos pueden no tener efecto en el comportamiento y pueden ser ignorados.

Cada tipo de evento debe ser asignado a las clases de objetos que lo envían y lo reciben. En ocasiones un objeto se autoenvía un evento, en cuyo caso el evento es al mismo tiempo entrada y salida para la misma clase.

1.2.3. PREPARACIÓN DEL SEGUIMIENTO DE UN EVENTO PARA CADA ESCENARIO

Se debe mostrar cada escenario como un rastreo (seguimiento) de eventos, esto es, una lista ordenada de eventos entre diferentes objetos que se arma en forma de tabla con cada objeto participante en el escenario en una columna.
Si más de un objeto de la misma clase participa en el mismo escenario, se asigna una columna por separado para cada objeto participante.
Con rastrear una columna particular de la tabla, se pueden ver los eventos que afectan directamente un objeto en particular. Solo estos eventos pueden aparecer en el diagrama de estado para ese objeto.

Se deben mostrar los eventos entre un grupo de clases en un diagrama de flujo de eventos. Este diagrama engloba todos los eventos entre clases sin importar la secuencia, además, se incluyen los eventos de todos los escenarios contemplando los eventos de error.

El diagrama de flujo de eventos es la contraparte dinámica para el diagrama de objetos. Las rutas en los diagramas de objetos muestran posibles flujos de información, las rutas en el diagrama de flujo de eventos señalan posibles flujos de control.

1.2.4. CONSTRUCCIÓN DE DIAGRAMAS DE ESTADO

En esta etapa, se deben elaborar los diagramas de estado para cada clase de objeto, mostrando los eventos que este objeto recibe y envía. Cada escenario corresponde a una ruta o rastreo de eventos a través del diagrama de estado. Cada rama en el flujo de control es representada por un estado con una o mas transiciones o flujos de salida.

Se debe partir de los diagramas de rastreo de eventos que afectan a la clase que está siendo modelada. Se elige un rastreo que muestre una interacción típica y se consideran solamente los eventos que afectan a un objeto individual. Se organizan los eventos en una ruta cuyos arcos se etiquetan con los eventos de entrada y salida encontrados a lo largo de una columna en la tabla de rastreos. El intervalo entre cualquier par de eventos es un estado, se le da un nombre a cada estado si éste es significativo. El diagrama inicial será una secuencia de eventos y estados, si el escenario puede repetirse indefinidamente, se cierra la ruta en el diagrama de estados.

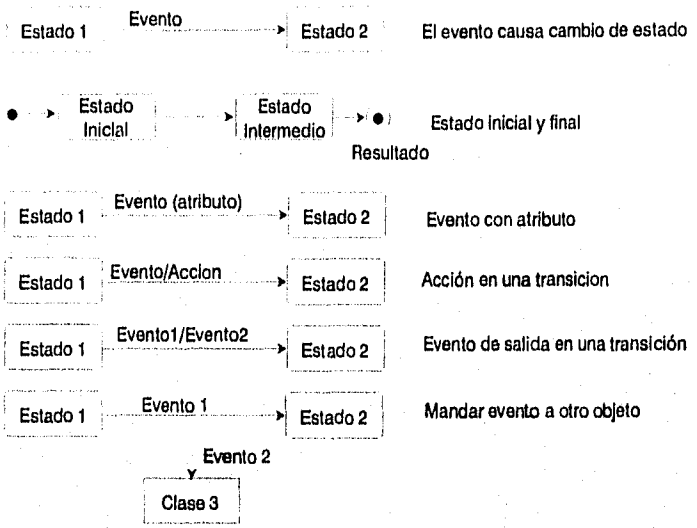
A continuación se encuentran los ciclos dentro de un diagrama de estados. Si la secuencia de eventos puede ser repetida indefinidamente, entonces forman un ciclo. Se reemplazan las secuencias finitas de eventos con ciclos donde sea posible. En un ciclo, el primer estado y el último son idénticos. Por lo menos un estado dentro de un ciclo debe tener múltiples transacciones que permitan abandonar el ciclo, de lo contrario el ciclo nunca terminaría.

La parte más difícil es decidir en cual estado, una ruta alterna se reintegra al diagrama existente. Dos trayectorias se unen en un estado si el objeto "desconoce u olvida" cual de las dos fue tomado. Cuando dos trayectorias resultan idénticas es factible utilizar parámetros para diferenciarlas. El adecuado uso de parámetros y transiciones condicionales puede simplificar considerablemente los diagramas de estado, pero con la desventaja de tener que mezclar información de estados y datos. Los diagramas de estado con mucha dependencia de datos pueden resultar muy confusos. Otra alternativa es particionar el diagrama de estado en dos diagramas concurrentes, usando un diagrama para la línea principal y otro para la información distinguible.

Después de que los eventos normales han sido considerados, se agregan casos límite y casos especiales.

El diagrama de estado de una clase se concluye cuando el diagrama cubre todos los escenarios y el diagrama maneja todos los eventos que puedan afectar al objeto de una clase en cada uno de sus estados.

Este proceso se repite para cada una de las clases de objetos. No todas las clases necesitan un diagrama de estado. Muchos objetos responden a eventos de entrada sin importar su estado anterior o manejan sus estados por medio de parámetros que no afectan el flujo de control.



Notación del Modelo Dinámico

1.2.5. RELACIONAR EVENTOS CON OBJETOS PARA VERIFICAR CONSISTENCIA

Cada evento debe tener un emisor y un receptor y puede que ocasionalmente sea el mismo objeto; no puede existir ningún estado en el diagrama que no tenga origen y destino.

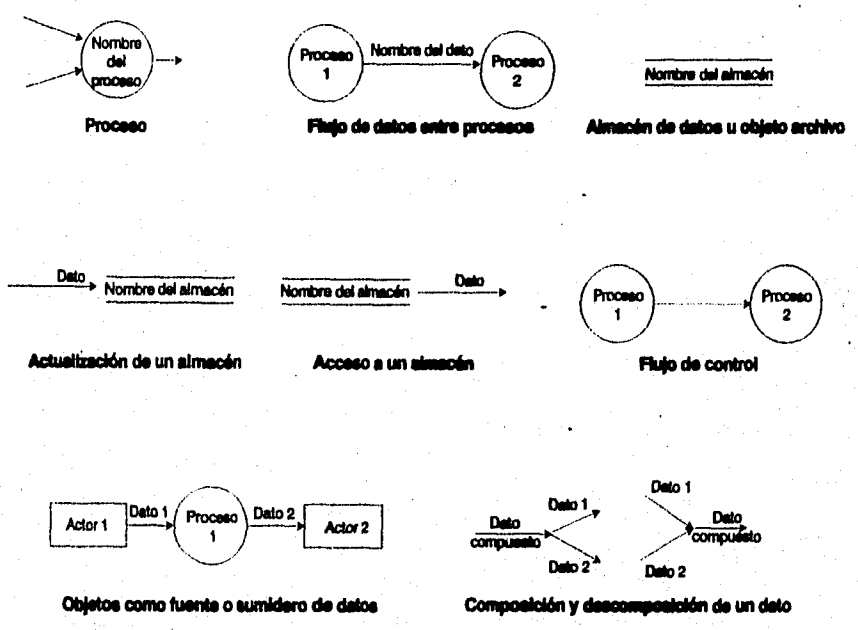
Es necesario seguir los efectos de un evento de entrada desde un objeto a otro, a través del sistema para asegurar que estos coincidan con los escenarios planteados.

El conjunto total de diagramas de estado para las clases de objetos constituye el modelo dinámico de una aplicación.

1.3. MODELO FUNCIONAL

El modelo funcional muestra como los valores son calculados, sin importar la secuencia, decisiones o estructuras de objetos. El modelo funcional muestra que valores dependen de cuales otros y que funciones los relacionan.

Los diagramas de flujo son útiles para mostrar dependencias funcionales. Las funciones son expresadas de muchas maneras, incluyendo el lenguaje natural, ecuaciones matemáticas o seudocódigo.



Notación en el modelo funcional

Los procesos en un diagrama de flujo corresponden a actividades o acciones en los diagramas de estado de las clases. Los flujos en un diagrama de flujo de datos corresponden a objetos o valores de atributos en un diagrama de objetos.

Los pasos a desarrollar para la construcción de un modelo funcional son los siguientes:

1.3.1. Identificación de valores de entrada y salida.
1.3.2 Construcción de diagramas de flujo de datos mostrando las dependencias funcionales.
1.3.3. Descripción de funciones.
1.3.4. Identificación de restricciones.
1.3.5. Especificación de criterios de optimización.

1.3.1. IDENTIFICACIÓN DE VALORES DE ENTRADA Y SALIDA

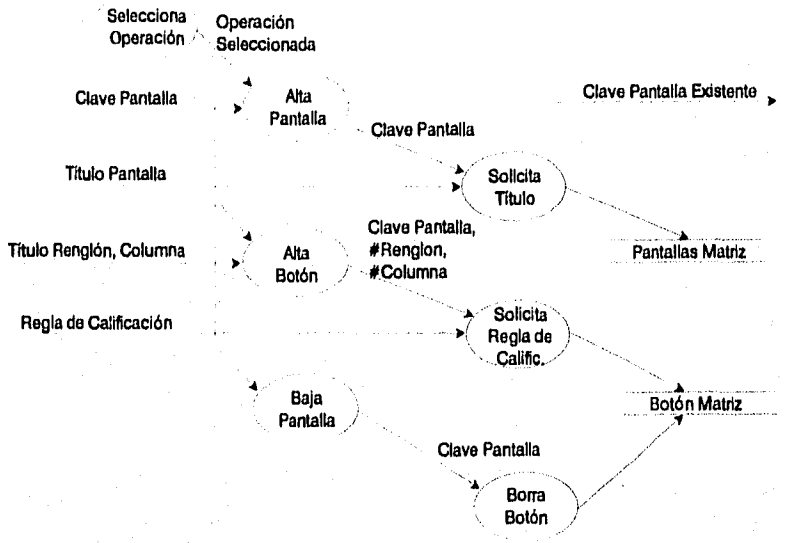
Se comienza por listar los valores de entrada y salida, éstos son parámetros de eventos entre el sistema y el mundo externo. También se debe examinar el planteamiento del problema encontrando cualquier entrada o salida que se haya olvidado.

1.3.2. CONSTRUCCIÓN DE DIAGRAMAS DE FLUJO DE DATOS

En esta fase se construye un diagrama de flujo de datos mostrando la forma en que cada valor de salida es calculado a partir de los valores de entrada. Un diagrama de flujo de datos normalmente se construye en capas. La capa superior debe consistir de un solo proceso o probablemente un proceso para cada entrada recibida, valores calculados y salidas generadas. Las entradas y salidas son suministradas y consumidas por objetos externos.

Dentro de cada capa del diagrama de flujo de datos, se trabaja al revés para determinar la función que calcula el valor de cada salida. Si las entradas de la operación son todas entradas del diagrama entero, entonces se ha terminado. En otro caso, si algunas de las entradas a la operación son valores intermedios deberán de ser monitoreadas al revés. Estos valores pueden ser monitoreado desde las entradas hacia las salidas, pero normalmente es mas difícil identificar todos los usos de una entrada que identificar todas las fuentes de las salidas.

Es necesario extenderse en cada proceso no trivial, comenzando de la capa superior hacia capas mas bajas en el diagrama de flujo. Si procesos del segundo nivel aun contienen procesos no triviales, éstos pueden expandirse recursivamente.
Es posible que un proceso de un diagrama de flujo de datos no produzca salidas inmediatas, sino que utilice depósitos de datos internos para guardar valores.



Ejemplo de Diagrama de flujo de datos

Los diagramas de flujo de datos especifican solo dependencias entre operaciones. No muestran decisiones o secuencias de operaciones.

Algunos valores de datos afectan decisiones en el flujo de control en el modelo dinámico. Las decisiones no afectan directamente los valores de salida en el modelo del diagrama del flujo de datos. Sin embargo, el introducir las funciones de decisión en el modelo de flujo de datos puede ser útil, aunque las salidas son señales de control, indicadas con una flecha punteada de salida.

Estas funciones son datos no visibles dentro del diagrama de flujo; sus salidas afectan el flujo de control en el modelo dinámico y no a los valores de salida directamente. Si se desea se puede dibujar una flecha de control hacia un proceso controlado por una decisión.

1.3.3. DESCRIPCIÓN DE FUNCIONES

Cuando el diagrama de flujo de datos ha sido suficientemente refinado, se tiene que hacer una descripción escrita de cada función. Esta puede ser en lenguaje narrativo, ecuaciones matemáticas, seudocódigo, tablas de decisión o alguna forma apropiada. Se debe poner énfasis en lo que la función hace, no en como implementarla.

La descripción puede ser narrativa o por procedimientos. La narrativa describe las relaciones entre valores de entrada y salida. La descripción detallada por procedimientos, especifica la función obteniendo un algoritmo para calcularla. El objetivo del algoritmo es solo especificar que hace la función; al llegar a la implementación, cualquier otro algoritmo que calcule el mismo valor puede ser sustituido.

Es preferible la descripción narrativa a la de procedimientos porque ésta no implica una implementación, aunque la de procedimientos es mucho más sencilla para escribirse y para ser usada.

1.3.4. IDENTIFICACIÓN DE RESTRICCIONES ENTRE OBJETOS

Las restricciones, son dependencias funcionales entre objetos que no están relacionados por una dependencia de entrada-salida. Las restricciones pueden ocurrir en dos objetos al mismo tiempo, entre instancias del mismo objeto a diferentes tiempos o bien entre instancias de diferentes objetos a diferentes tiempos. Las precondiciones en las funciones son restricciones que los valores de entrada deben satisfacer, las postcondiciones son restricciones que los valores de salida están garantizados a guardar.

1.3.5. ESPECIFICACIÓN DE CRITERIOS DE OPTIMIZACIÓN

Se deben especificar valores para ser maximizados, minimizados o de otra forma optimizados. Si existen diferentes criterios de optimización que estén en conflicto, hay que indicar como es decidido el intercambio de tareas. No hay que preocuparse por ser muy preciso puesto que generalmente no se está disponible para hacer esto y de cualquier manera, los criterios probablemente cambien antes de que el proyecto se concluya.

1.4. AGREGAR OPERACIONES

Las operaciones en lenguajes de programación orientados a objetos pueden corresponder a consultas acerca de atributos o asociaciones en el modelo de objetos, a eventos en el modelo dinámico y a funciones en el modelo funcional. Resulta más útil distinguir esta variedad de operaciones durante el análisis.

1.4.1. OPERACIONES DEL MODELO DE OBJETOS

Estas operaciones incluyen lecturas y escrituras de los valores de atributos y ligas de asociaciones. No requieren ser mostradas de manera explícita en el modelo de objetos, pero están implicadas por la presencia de un atributo. Durante el análisis se asume que todos los atributos son accesibles.

1.4.2. OPERACIONES DE EVENTOS

Cada evento enviado a un objeto corresponde a una operación en él. Dependiendo de la arquitectura del sistema, los eventos pueden implementarse directamente incluyendo un manejador de eventos como parte del sistema o pueden convertirse en métodos explícitos. Durante el análisis, los eventos se representan mejor como etiquetas en transiciones de estados y no deben listarse explícitamente en el modelo de objetos.

1.4.3. OPERACIONES DE ACCIONES DE ESTADO Y ACTIVIDADES

Toda acción y actividad en el diagrama de estados puede expresarse como función y debe estar definida como operación en el modelo de objetos.

1.4.4. OPERACIONES DE FUNCIONES

Cada función en el diagrama de flujo corresponde a una operación de un objeto o posiblemente a varios de ellos. Es conveniente organizar estas funciones dentro de operaciones en objetos.

Si la misma serie de ecuaciones o fragmentos de seudocódigo describe más de una función, entonces una nueva operación puede ser introducida para simplificar el modelo funcional.

1.4.5. SIMPLIFICACIÓN DE OPERACIONES

Se debe examinar el modelo de objetos buscando operaciones similares y variaciones de esta para formar una simple operación. Se debe tratar de difundir la definición de una operación para armonizar sus variaciones y casos especiales.

Se utiliza la herencia cuando sea posible para reducir el número de operaciones distintas. Se pueden introducir tantas nuevas superclases como se requiera para simplificar las operaciones. Se coloca cada operación en el nivel correcto dentro de la jerarquía de clases.

1.5. ITERACIÓN DEL ANÁLISIS

La mayoría de los modelos de análisis requieren tener más de una iteración para completarse. La mayoría de las definiciones de los problemas y aplicaciones no pueden tener un enfoque completamente lineal, debido a que las diferentes partes del problema interactúan entre sí. Para entender un problema con todas sus implicaciones, se debe atacar el análisis iterativamente, preparando una primera aproximación del modelo y entonces

iterar el análisis, conforme su entendimiento se incremente. No existe una línea clara entre el análisis y el diseño, de tal manera que debe exagerarse esta tarea.

1.5.1. REFINAMIENTO DEL MODELO DE ANÁLISIS

El modelo de análisis completo pudiera mostrar inconsistencias en alguno de sus modelos. Se deben iterar las diferentes fases para producir un diseño más claro y coherente.

Algunas de las actividades para lograr este objetivo son:

- Reexaminar las construcciones para encontrar conceptos erróneos.
- Revisar las generalizaciones hechas.
- Revisar los papeles que desempeñan los objetos.
- Remover objetos o asociaciones que ya no resultan importantes o que sean extrañas en esta etapa.

1.5.2. REESTABLECIMIENTO DE REQUERIMIENTOS

Cuando el análisis está completo, el modelo sirve de base para los requerimientos y define el alcance de futuras revisiones. La mayoría de los requerimientos formarán parte del modelo. Algunos requerimientos especifican restricciones de rendimiento. Estos deben de ser claramente establecidos en conjunto con el criterio de optimización. Otros requerimientos especifican el método de solución ; estos deben de separarse y ser rebatidos de ser posible..

El modelo final debe ser verificado con el usuario. El análisis sirve de base para la arquitectura, diseño e implementación. La definición original del problema debe ser revisada para incorporar las correcciones y la comprensión del problema alcanzado durante el análisis.

1.5.3. FRONTERA ENTRE EL ANÁLISIS Y EL DISEÑO

La meta del análisis es especificar completamente el problema y el dominio de la aplicación, sin introducir cualquier noción de implementación en particular, pero en la práctica es casi imposible evitar todas las influencias de la implementación. No existe en lo absoluto una línea entre las fases del diseño y el análisis y tampoco existe un análisis perfecto. El propósito de la reglas de la metodología es conservar la flexibilidad y permitir cambios posteriores, pero recordando que la meta del modelado es completar el trabajo total, la flexibilidad es solo un medio.

2.- DISEÑO DEL SISTEMA

2.1. PREÁMBULO DEL DISEÑO DE SISTEMAS

Durante el análisis, el enfoque que debemos tener es el de ¿Qué se debe hacer?, independientemente de como deba hacerse. Durante el diseño, las decisiones que se toman son para determinar la forma de resolver el problema, partiendo de un diseño de alto nivel y llegando gradualmente a los niveles de detalle.

En la primera etapa del diseño del sistema, se selecciona el enfoque básico de como se resolverá el problema. La arquitectura del sistema es la organización general de éste en componentes llamados subsistemas.

La arquitectura proporciona el contexto sobre el cual se harán decisiones de mayor detalle en etapas posteriores del diseño. Al tomar decisiones de alto nivel que afectan a todo el sistema, el diseñador particiona el problema en subsistemas que podrán ser atacados posteriormente y en forma independiente.

Durante el diseño se deben tomar las siguientes decisiones.

2.1.1. Organizar el sistema en subsistemas.
2.1.2. Identificar concurrencia inherente en el problema.
2.1.3. Asignar subsistemas a procesadores y tareas.
2.1.4. Elegir un enfoque para la administración de almacenes de datos.
2.1.5. Manejo del acceso a los recursos globales.
2.1.6. Elegir la implementación de control en el software.
2.1.7. Manejo de condiciones límite.
2.1.8. Asignar prioridades intercambiables.

2.1.1. ORGANIZAR EL SISTEMA EN SUBSISTEMAS.

El primer paso en el diseño de sistemas, es dividir el sistema en un pequeño número de componentes. Cada componente especializado de un sistema es llamado subsistema. Cada subsistema abarca aspectos del sistema que comparten algunas propiedades comunes. (funcionalidad similar, misma ubicación física , ejecución en el mismo tipo de hardware, etc.).
Un subsistema no es un objeto ni una función, pero si, un conjunto de clases, asociaciones, operaciones, eventos y restricciones que están interrelacionados y que tienen una interfase razonablemente bien definida y no muy amplia con respecto a otros subsistemas. Un subsistema generalmente esta identificado por los servicios que proporciona. Un servicio es un grupo de funciones relacionadas que comparten algún propósito común. Un subsistema define una forma coherente de analizar un aspecto del problema.

Cada subsistema posee una interfase bien definida. La interfase especifica la forma de todas las interacciones y el flujo de información a través de los límites del subsistema, pero no muestra como se implementa internamente el subsistema. Cada subsistema puede entonces ser diseñado independientemente sin afectar a los otros. Los subsistemas deben definirse de tal forma que la mayor parte de las interacciones estén dentro de ellos en lugar de estar sobre sus límites, para reducir la dependencia entre subsistemas.

Un sistema debe dividirse en un pequeño número de subsistemas, 20 son probablemente demasiados, pero es posible descomponer los subsistemas en componentes más pequeños. Al nivel más bajo de subsistemas se le llama módulo.

La descomposición de sistemas en subsistemas puede organizarse como una secuencia de capas horizontales o como particiones verticales.

CAPAS.

Las capas pueden verse como un conjunto ordenado de mundos virtuales, cada uno de los cuales se construye en los términos definidos por aquellos

que están por debajo de él y proporciona las bases de implementación para aquellos superiores a éste.

Ejemplo de un sistema en capas

Los objetos en cada capa pueden ser independientes, aunque frecuentemente existe alguna correspondencia entre objetos en diferentes capas. La relación se establece en un solo sentido ya que los subsistemas tienen conocimiento de aquella o aquellas capas debajo de él, pero no de las superiores. Una relación cliente-servidor existe entre capas inferiores (proveedores de servicios) y capas superiores (usuarios de servicios).

Las arquitecturas en capas pueden ser de dos formas:

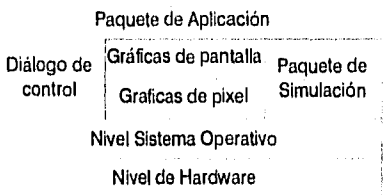
Cerradas o abiertas. En una arquitectura cerrada, cada capa es construida solo en los términos de su capa inferior inmediata. Esto reduce la dependencia entre capas y permite que los cambios sean más fáciles de hacer, debido a que la interfase de las capas solamente afecta a la capa siguiente. En una arquitectura abierta, una capa puede utilizar características de cualquiera de las capas inferiores de cualquier nivel. Esto reduce la necesidad de definir operaciones en cada nivel, lo cual resulta en un código más eficiente y compacto. Sin embargo, una arquitectura abierta no observa un principio de ocultamiento de información. Los cambios de un subsistema pueden afectar a otra capa superior, de tal manera una arquitectura abierta resulta menos robusta que una arquitectura cerrada. Ambas arquitecturas resultan útiles, el diseñador debe valorar el beneficio en eficiencia contra el beneficio en modularidad.

Un sistema construido en capas puede portarse a otras plataformas de hardware y software reescribiendo una sola capa. Es una buena práctica introducir al menos una capa de abstracción entre la aplicación y cualquier servicio proporcionando por el hardware o el sistema operativo.

PARTICIONES

Las particiones dividen verticalmente un sistema en diversos subsistemas independientes y débilmente acoplados que cumplen cada uno con un servicio especializado.

Un sistema puede descomponerse sucesivamente en subsistemas que utilicen tanto particiones como capas.



Sistema dividido en particiones y capas

Bajo esta estructura, los subsistemas tienen poco conocimiento de los demás. Un subsistema puede descomponerse sucesivamente en subsistemas ampliando capas y particiones. Las capas pueden particionarse y las particiones pueden dividirse en capas.

La mayor parte de los sistemas grandes requieren una combinación de capas y particiones.

TOPOLOGÍA DEL SISTEMA.

Cuando los subsistemas de más alto nivel son identificados, el diseñador debe mostrar el flujo de información entre los subsistemas con un diagrama de flujo de datos.

Algunas ocasiones, todos los sistemas interactúan entre sí, pero con frecuencia los flujos resultan muy simples. Es deseable el tratar de utilizar topologías simples cuando éstas se adapten al problema con el fin de reducir el número de interacciones entre los subsistemas.

2.1.2. IDENTIFICACIÓN DE CONCURRENCIAS

En el modelo de análisis, como en el mundo real y en el hardware, todos los objetos son concurrentes, sin embargo, en una implementación de sistemas, no todos los objetos de software son concurrentes debido a que un procesador debe atender muchos objetos.

En la práctica muchos objetos pueden ser implementados en un solo procesador si los objetos no pueden estar activos al mismo tiempo.

Una meta importante del diseño de sistemas, es identificar cuales objetos deben estar activados concurrentemente y cuales de ellos tienen una actividad mutuamente exclusiva.

IDENTIFICACIÓN DE CONCURRENCIA INHERENTE.

El modelo dinámico es la guía para poder identificar la concurrencia. Dos objetos son inherentemente concurrentes, si pueden recibir eventos al mismo tiempo sin ningún tipo de interacción. Si estos objetos son asíncronos, no pueden integrarse por una sola línea de control. Por ejemplo, el motor y el control de las alas de un aeroplano deben estar operando concurrentemente aunque también pueden hacerlo en forma independiente. La independencia de los subsistemas es deseable ya que pueden asignarse a diferentes unidades de hardware sin ningún costo por la comunicación.

2.1.3. ASIGNACIÓN DE SUBSISTEMAS A PROCESADORES Y TAREAS

Cada subsistema concurrente debe ser asignado a una unidad de hardware, ya sea con un procesador de propósito general o con unidad funcional especializada. El diseñador debe de:

- Estimar las necesidades de rendimiento y los recursos requeridos para satisfacerlas.
- Elegir la implementación de hardware y/o software para los subsistemas.

- Asignar los subsistemas de software a procesadores para satisfacer las necesidades de desempeño y minimizar la comunicación entre procesadores.
- Determinar la conectividad de las unidades físicas que implementan los subsistemas

ESTIMACIÓN DE LOS REQUERIMIENTOS DEL HARDWARE.

La decisión de utilizar múltiples procesadores o unidades funcionales de hardware, está basada en la necesidad de buscar un desempeño mas alto que la que un solo procesador pueda proporcionar. El número de los procesadores requeridos depende del volumen de procesos y la velocidad de la máquina.

El diseñador del sistema debe estimar el poder de procesamiento del CPU requerido, calculando la carga regular como producto del número de transacciones por segundo y el tiempo requerido para procesar una transacción. La estimación será generalmente imprecisa por lo cual debe incrementarse para considerar otros factores que puedan afectar el desempeño.

INTERCAMBIO DE HARDWARE - SOFTWARE.

El hardware puede ser considerado como una rígida pero altamente optimizada forma de software. El diseñador del sistema debe decidir que subsistemas serán implementadas en hardware y cuales en software.

Los subsistemas se implementan en hardware por 2 razones principales:

1. Existencia del hardware que proporciona la funcionalidad requerida.

2. Se requiere un desempeño mayor al que puede proporcionar un solo procesador.

ASIGNACIÓN DE TAREAS A PROCESADORES.

Las tareas de los subsistemas implementados en software, se deben asignar a procesadores debido a que:

- Ciertas tareas son requeridas en localizaciones físicas especificas, ya sea para controlar el hardware o para permitir una operación independiente o concurrente.
- El tiempo de respuesta o el índice de flujo de información excede los márgenes de comunicación entre una tarea y una pieza de hardware.
- Las velocidades de cálculo son demasiado grandes para un solo procesador, por lo que las tareas deben diseminarse entre diversos procesadores.

DETERMINACIÓN DE LA CONECTIVIDAD FÍSICA.

Una vez determinado el tipo y número (por lo menos aproximado) de unidades físicas, el diseñador debe elegir la disposición y forma de las conexiones entre las unidades físicas.

Deben tomarse las siguientes decisiones.

- Elegir la topología de conexión de las unidades físicas. Las asociaciones en el modelo de objetos corresponden a menudo con conexiones físicas.

- Elegir la topología de unidades repetidas. Si diversas copias de un tipo particular de unidad o grupos de unidades, son incluidas por razones de desempeño, su topología debe especificarse. El modelo de objetos y el modelo funcional no resultan guías útiles debido a que el uso de múltiples unidades, es primeramente una optimización del diseño no requerida por el análisis. La topología de unidades repetidas generalmente tiene un patrón regular, tales como una secuencia lineal, una matriz, un árbol, o una estrella.

- Elegir la forma de los canales de conexión y los protocolos de comunicación de forma general; aun cuando las conexiones todavía son lógicas y no físicas, éstas deben de ser consideradas.

2.1.4. ADMINISTRACIÓN DE LOS ALMACENES DE DATOS.

Los almacenes de los datos internos y externos en un sistema proporcionan puntos claros de separación entre subsistemas con interfases bien definidas. En general cada almacén de datos puede combinar estructuras de datos, archivos y bases de datos implementadas en memoria o en un dispositivo de almacenamiento secundario. Las diferentes clases de almacenamiento de datos tienen como puntos de comparación, el costo, el tiempo de acceso, la capacidad y la confiabilidad.

Los archivos son una forma de almacenamiento de datos barata, simple y permanente, sin embargo, para realizar las operaciones de archivo es necesario implementar más código para llegar a un nivel adecuado de abstracción.

Las bases de datos manejadas por sistemas manejadores de bases de datos (DBMS) son otra clase de almacenamiento de datos y son las que en la actualidad son más usadas. Algunos tipos de DBMS están diseñados en el modelo jerárquico, de red o relacional, orientado a objetos y/o lógicas .

VENTAJAS DE USAR UNA BASE DE DATOS.

- Trabajan con muchas herramientas de infraestructura. Distribución de datos, posibilidad de compartir múltiples usuarios y múltiples aplicaciones integridad, estabilidad, soporte de transacciones y recuperación de datos en caso de caídas fortuitas.
- Interfase común para todas las aplicaciones. Cada aplicación accesa a un subconjunto de información que necesita e ignora el resto.
- Portabilidad entre plataformas.
- Lenguaje estándar de acceso. El SQL, (lenguaje estructurado de consultas) es el lenguaje soportado por las bases de datos más comerciales.

DESVENTAJAS DE USAR UNA BASE DE DATOS.

Los DBMS algunas veces tienen desventajas en su uso que evitan que su utilización sea la adecuada en problemas reales.

- *Sobrecarga del rendimiento.* Cuando las aplicaciones demandan muchos recursos, el diseñador del sistema debe evaluar la posibilidad de que el proveedor incremente el desempeño o pueda desarrollar una solución a la medida.
- *Insuficiente funcionalidad para aplicaciones avanzadas.* Los DBMS fueron diseñados principalmente para aplicaciones de negocios en los que se tiene mucha información pero con una estructura sencilla, por lo que las aplicaciones que necesitan una estructura de datos mas compleja, el uso de un DBMS resulta mas complicado.
- *Interfases inconvenientes con lenguajes de programación.* Los DBMS soportan operaciones orientadas a conjuntos que están expresadas mediante lenguajes no procedurales. La mayoría de los lenguajes son procedurales por naturaleza y pueden accesar un renglón de la tabla del DBMS relacional a la vez.

Algunas de estas desventajas podrán desaparecer cuando se implemente un DBMS orientado a objetos.

2.1.5. MANEJO DE RECURSOS GLOBALES.

El diseñador debe identificar los recursos globales y determinar los mecanismos para controlar el acceso a ellos. Los recursos globales incluyen: unidades físicas, como procesadores, unidades de cinta, satélite de comunicaciones, espacio en disco, tipo de pantallas y Mouse; nombres lógicos, como los identificadores de los objetos, nombres de archivos y nombres de clases y por último accesos a datos compartidos como bases de datos.

Si el recurso es un objeto físico, éste puede autocontrolarse estableciendo un protocolo de obtención de acceso dentro de un sistema concurrente. Si el recurso es una entidad lógica, como un identificador de objeto o una base de datos, existe el peligro de conflictos de acceso al trabajar en un ambiente compartido.

2.1.6. ELECCIÓN DE LA IMPLEMENTACIÓN DEL SOFTWARE DE CONTROL.

Aunque no es necesario que todos los subsistemas utilicen la misma implementación, generalmente el diseñador elige un solo estilo de control para todo el sistema. Existen dos tipos de flujo de control en un sistema de software: con control interno ó con control externo.

El control externo es el flujo de eventos externamente visibles entre los objetos del sistema. Existen tres tipos de control para eventos externos:

1) Secuencial dirigido por procedimientos.
2) Secuencial dirigido por eventos.
3) Concurrente.

El estilo de control adoptado depende de los recursos disponibles y del patrón de interacciones en la aplicación.

El control interno es el flujo de control dentro de un proceso. El diseñador puede elegir descomponer un proceso en diversas tareas por razones de claridad lógica y/o desempeño. A diferencia de los eventos externos, las transferencias internas de control, tales como llamadas a procedimientos o llamadas entre tareas, se encuentran bajo la dirección del programa y pueden estructurarse por conveniencia. Existen 3 tipos comunes de flujo de control:

1) Llamadas a procedimientos.
2) Llamadas entre tareas semiconcurrentes.
3) Llamadas entre tareas concurrentes.

SISTEMAS DIRIGIDOS POR PROCEDIMIENTOS.

En un sistema dirigido por procedimientos el control se establece dentro del código del programa.

Los procedimientos producen peticiones de entrada de datos externos y esperan a que estos sean alimentados. Cuando ocurre la entrada, el control continua con el procedimiento que hizo la llamada. La localización del

contador del programa y la pila de llamadas de procedimientos y variables locales definen el estado del sistema.

La mayor ventaja de un control dirigido por procedimientos es que resulta fácil de implementar con lenguajes convencionales; la desventaja es que requiere que la concurrencia inherente en los objetos tenga que ser ajustada a un flujo de control secuencial. El diseñador debe convertir los eventos en operaciones entre objetos. Las entradas de tipo asíncrono no pueden ser fácilmente adaptadas bajo éste esquema, debido a que el programa debe solicitar explícitamente una entrada de datos. El enfoque dirigido por procedimientos es adaptable solo si el modelo de estados muestra una combinación regular de eventos de entrada y salida. Las interfases de usuario flexibles y sistemas de control resultan difíciles de construir utilizando este enfoque.

SISTEMAS DIRIGIDOS POR EVENTOS.

En un sistema secuencial dirigido por eventos, el control está en manos de un despachador o monitor proporcionado por el lenguaje de programación, por un subsistema o bien, por el sistema operativo. Los procedimientos de la aplicación se conectan a los eventos y son llamados por el despachador cuando ocurre el evento correspondiente ("Call back"). Todos los procedimientos regresan el control al despachador, en vez de retener el control hasta que llegue la entrada. Los eventos son manejados directamente por el despachador. Debido a lo anterior, el estado del programa no puede ser determinado utilizando el contador del programa o la pila. Los procedimientos deben utilizar variables globales para mantener su estado o, en su defecto, el despachador debe mantener el estado local de ellas.

Los sistemas dirigidos por eventos permiten utilizar patrones de control más flexibles que los sistemas dirigidos por procedimientos. Simulan la operación de procesos cooperativos en una sola línea de control. Un procedimiento fuera de esta línea de control puede bloquear la aplicación completa, por lo que se debe tener cuidado.

Se debe utilizar un sistema dirigido por eventos para el control externo cuando sea posible, en vez de utilizar sistemas dirigidos por procedimientos,

dado a que el mapeo de eventos en construcciones de programas es mucho más simple y poderoso que en un sistema dirigido por procedimientos. Los sistemas dirigidos por eventos también son más modulares y pueden manejar condiciones de error mejor que los sistemas dirigidos por procedimientos.

SISTEMAS CONCURRENTES

En un sistema concurrente, el control se establece de manera concurrente en diversos objetos independientes. Los eventos son manejados directamente como mensajes de un solo sentido entre objetos. Una tarea puede esperar por alguna entrada, mientras otras continúan ejecutándose. El sistema operativo generalmente debe proveer un mecanismo de "cola de eventos" para que éstos no sean perdidos si la tarea se encuentra ocupada cuando llegue el evento. Además el sistema operativo debe resolver el tráfico entre las tareas. Ninguno de los actuales lenguajes orientados a objetos mas conocidos soportan directamente este tipo de control

CONTROL INTERNO

Las interacciones internas entre los objetos pueden ser vistas en forma similar a las interacciones externas entre objetos, (eventos entre objetos) y el mismo mecanismo de implementación puede ser usado. Existe una diferencia importante: Las interacciones externas involucran la espera de eventos debido a que los diferentes objetos son independientes y no pueden forzar a otros a responder; las operaciones internas son generadas por objetos como parte del algoritmo de implementación; por lo que sus patrones de respuesta son predecibles. La mayoría de las operaciones internas pueden por lo tanto, ser vistas como llamadas a procedimientos, en los cuales el procedimiento que llama, produce una petición y espera por la respuesta.

2.1.7. MANEJO DE CONDICIONES DE LIMITE.

Aunque la mayoría de los esfuerzos en el diseño de muchos sistemas están dirigidos al comportamiento estable de éste, el diseñador debe considerar

condiciones de límite tales como, inicialización, terminación y fallas. Las siguientes clases de problemas deben ser consideradas:

- *Inicialización*: El sistema puede ser llevado desde un estado inicial inactivo a una condición estable de operación. Entre las cosas que pueden ser inicializadas se incluyen constantes, parámetros, variables globales, tareas y posiblemente la jerarquía de clases misma. Durante la inicialización solo un subconjunto de la funcionalidad del sistema es encuentra disponible.

- *Terminación*: Generalmente es más simple que la inicialización ya que muchos objetos internos pueden ser simplemente abandonados. En esta tarea se deben liberar todos aquellos recursos externos que se tengan reservados y si se trata de un sistema concurrente, la tarea debe avisar a otras de su terminación.

- *Fallas*: Las fallas se consideran como una terminación no planeada del sistema, una falla puede ser producto de un error del usuario, por falta de recursos, o por una anomalía externa. Un buen diseñador planea este tipo de fallas.

2.1.8. ASIGNAR PRIORIDADES INTERCAMBIABLES.

El diseño de sistemas debe establecer prioridades que tienen que conducir hacia posibles cambios durante el diseño. Al diseñador se le requiere que elija entre metas deseables pero incompatibles. En ocasiones es necesario sacrificar la funcionalidad total de una pieza de software para poder utilizarla prematuramente. Muchas veces en los subsistemas es mas importante la velocidad del proceso que la capacidad de memoria, en otras es el entendimiento, la presentación, etc.

Ocasionalmente el planteamiento del problema especifica cuales metas tienen mayor prioridad, pero generalmente el trabajo lo debe hacer el diseñador reconciliando los deseos incompatibles del usuario y decidiendo como se asignarán las prioridades.

Los intercambios no se realizan totalmente durante el diseño pero las prioridades para realizarlos se establecen durante ésta etapa.

2.2. ARQUITECTURAS COMUNES.

Existen diferentes prototipos de arquitecturas de sistemas que son comunes en los sistemas existentes. Cada uno de estas es fácilmente aplicable a cierto tipo de sistemas. Si se tiene una aplicación con características similares, es posible ahorrar esfuerzos utilizando la arquitectura correspondiente, o al menos como punto inicial del diseño.

Los tipos de sistemas incluyen:

- *Transformación en lote*: Una transformación de datos ejecutada una sola vez para un conjunto completo de entradas.
- *Transformación continua*: Es una transformación de datos realizada continuamente conforme cambia la entrada.
- *Interfase interactiva*: Es un sistema caracterizado por interacciones externas.
- *Simulación dinámica*: Un sistema que simula la utilización de objetos del mundo real.
- *Sistema en tiempo real*: Sistema caracterizado por restricciones estrictas de tiempo.
- *Manejador de transacciones*: Sistema consistente en guardar y actualizar datos, usualmente incluyendo accesos concurrentes desde diferentes direcciones físicas.

Estas son solo algunas de las formas comunes de arquitecturas. Algunos problemas requieren un nuevo tipo de arquitectura, pero la mayoría pueden utilizar alguna de las existentes o al menos una variación de estas. Es posible utilizar en algún problema también una combinación de ellas.

2.2.1. TRANSFORMACIÓN EN LOTES

Es una transformación secuencial de entrada a salida, en la cual las entradas son suministradas al comienzo y la meta es calcular la respuesta; No existe interacción con el mundo externo.

El modelo de estados es trivial o inexistente para estos problemas y el modelo de objetos puede ser simple o complejo. El aspecto más importante de una

transformación en lote, es el modelo funcional que especifica como los valores de entrada son transformados en valores de salida. Esta área es probablemente la mejor señalada por las metodologías actuales que enfatizan los diagramas de flujo de datos y la descomposición funcional. Sin embargo el uso de modelos de objetos para estructuras de datos mejora en métodos previos de representación de datos para problemas que tienen datos complejos.

Los pasos para diseñar una transformación en lote son:

- Dividir la transformación total en etapas, cada una desarrollando una parte de la transformación.
- Definir clases intermedias de objetos para los flujos de datos entre cada par de etapas sucesivas.
- Extender cada etapa hasta que las operaciones estén preparadas para la implementación.
- Reestructurar los flujos finales para optimización.

2.2.2. TRANSFORMACIÓN CONTÍNUA.

La transformación contínua es un sistema en el cual las salidas dependen activamente de entradas cambiantes y deben ser periódicamente actualizadas. A diferencia de la transformación en lotes en la cual las salidas son calculadas una sola vez, las salidas en una transformación contínua deben ser actualizadas frecuentemente.

El modelo funcional junto con el modelo de objetos definen el valor a calcular; el modelo dinámico es menos importante porque la mayor parte de la estructura de la aplicación proviene del flujo estático de datos y no de las interacciones discretas.

Debido a que es imposible hacer un reproceso completo cada vez que cambia el valor de entrada, la arquitectura para la transformación contínua debe facilitar el procesamiento incremental. La transformación puede ser implementada como un flujo de funciones. El efecto de cada cambio en cada valor de entrada debe propagarse a través de todo el flujo. Para hacer posible el procesamiento incremental, deben definirse objetos que retengan valores

intermedios y valores redundantes que pueden ser introducidos para mejorar el desempeño.

La sincronía de valores en el flujo puede ser importante para sistemas de alto rendimiento. En tales casos, las operaciones se realizan en tiempos bien definidos y el flujo de operaciones debe ser cuidadosamente balanceado de modo que los valores lleguen al lugar correcto en el tiempo correcto para evitar los cuellos de botella.

Los pasos para diseñar un flujo para una transformación contínua son:

- Dibujar un diagrama de flujo de datos para el sistema.
- Definir objetos intermedios entre cada par de etapas sucesivas.
- Diferenciar cada operación para obtener cambios incrementales en cada etapa.
- Agregar objetos intermedios adicionales para optimizarlo.

2.2.3. INTERFASES INTERACTIVAS.

Es un sistema que está caracterizado por interacciones entre el sistema y agentes externos, los cuales son independientes del sistema, y por lo tanto, sus entradas no pueden ser controladas, aunque el sistema puede solicitar respuestas de ellos. Una interfase interactiva, normalmente incluye solo parte de una aplicación entera, la que puede ser manipulada independientemente de la parte computacional de la aplicación, lo más importante para una interfase interactiva son los protocolos de comunicación entre el sistema y los agentes externos, la sintaxis de interacciones posibles, la presentación de las salidas, el flujo de control dentro del sistema y el fácil entendimiento de la interfase por parte del usuario, el desempeño y la manipulación de errores.

Las interfases interactivas están caracterizadas por el modelo dinámico, los objetos que en el modelo de objetos representan elementos de interacción tales como señales de entrada y salida, así como formatos de presentación. El modelo funcional describe que funciones de la aplicación serán ejecutas en respuesta a una secuencia de eventos de entrada. pero la estructura interna de las funciones usualmente es poco importante para el comportamiento de la

interfase. Un sistema interactivo se preocupa por la apariencia externa y no profundiza en la estructura semántica.

Los pasos en el diseño de una interfase interactiva son:

1) Aislar los objetos que forman la interfase de los objetos y que definen la semántica de la aplicación.

2) Utilizar en la medida de lo posible, objetos predefinidos para interactuar con agentes externos.

3) Usar el modelo dinámico como estructura del programa, las interfases interactivas son mejor implementadas usando controles concurrentes (multitarea) o controles dirigidos por eventos. (Interrupciones call-back).

4) Aislar los eventos físicos de los eventos lógicos. Con frecuencia un evento lógico corresponde a múltiples eventos físicos.

5) Especificar ampliamente las funciones de la aplicación, que son llamadas por la interfase, asegurándose que la información para implementarlas esté presente.

2.2.4. SIMULACIÓN DINÁMICA

Una simulación dinámica modela o rastrea objetos en el mundo real. Las metodologías tradicionales construidas en base a diagramas de flujo de datos resultan pobres para representar estos problemas, debido a que las simulaciones involucran muchos objetos distintos que se actualizan constantemente a si mismos en vez de actualizarse en una sola transformación. Las simulaciones son el sistema mas simple de diseñar empleando el enfoque orientado a objetos. Los objetos y operaciones provienen directamente del contexto de la aplicación. El control puede implementarse de dos maneras: Mediante un controlador explícito externo a los objetos de la aplicación, que pueda simular una máquina de estados, o mediante el intercambio de mensajes entre objetos, en forma similar a una situación en el mundo real.

A diferencia de los sistemas interactivos, los objetos internos en una simulación dinámica corresponden a objetos en el mundo real, por lo que el modelo de objetos resulta muy importante y frecuentemente complejo.

Los pasos para diseñar una simulación dinámica son:

1) Identificar los actores, objetos activos del mundo real a partir del modelo de objetos. Los actores poseen atributos que periódicamente son actualizados.

2) Identificar eventos discretos, los cuales corresponden a interacciones discretas con el objeto. Los eventos discretos pueden ser implementados como operaciones en el objeto.

3) Identificar dependencias continuas. Los atributos del mundo real pueden depender de otros atributos del mundo real o variar continuamente con el tiempo, estos atributos deben ser actualizados a intervalos periódicos utilizando técnicas de aproximación numérica para minimizar los errores de cuantificación.

4) Generalmente una simulación es dirigida por un ciclo de tiempo en base a una pequeña escala de tiempo. Los eventos discretos entre los objetos pueden intercambiarse continuamente como parte del ciclo de tiempo.

El problema más difícil con las simulaciones es generalmente el de proporcionar el adecuado desempeño.

2.2.5. SISTEMAS DE TIEMPO REAL

Un sistema de tiempo real es un sistema interactivo para el cual las restricciones de tiempo en acciones son particularmente estrechas o en el cual una mínima perdida de tiempo no puede ser tolerada. Para acciones críticas el sistema debe garantizar la respuesta dentro de un intervalo absoluto de tiempo. Algunos casos típicos son: Control de procesos, dispositivos de comunicaciones o relevadores de sobrecarga. Para garantizar los tiempos de respuesta debe determinarse el escenario del comportamiento del sistema en el peor de los casos.

El diseño de sistemas de tiempo real es complejo e incluye temas como manejadores de interrupciones, prioridad de tareas y coordinación de múltiples procesadores.

2.2.6. MANEJADORES DE TRANSACCIONES

Es un sistema de base de datos el cual tiene como función principal almacenar y accesar información. La información proviene del dominio de la aplicación. La mayoría de los manejadores de transacciones tienen que operar con múltiples usuarios y concurrencias. Una transacción debe ser manejada como una sola entidad sin interferencia de otras transacciones.

El modelo de objetos es el más importante; el modelo funcional en un sistema manejador de transacciones es menos importante debido a que las operaciones tienden a ser predefinidas y enfocadas a la actualización y consulta de información. El modelo dinámico muestra el acceso concurrente de información distribuida. La distribución es una parte inherente del problema del mundo real y debe ser modelada como parte del análisis.

Frecuentemente se puede utilizar un sistema manejador de base de datos ya existente en cuyos casos el principal problema es construir el modelo de objetos y seleccionar la descomposición de las transacciones que deben ser consideradas como unidades del sistema.

Los pasos para diseñar un manejador de transacciones son:

1) Convertir el modelo de objetos directamente dentro de una base de datos.

2) Determinar las unidades de concurrencia, esto es, los recursos que inherentemente o por especificación no pueden ser compartidos. Las clases nuevas deben introducirse conforme se requieran.

3) Determinar la unidad de transacción, esto es, el conjunto de recursos que deben ser accesados juntos durante una transacción. Típicamente una transacción tiene éxito o fracasa completamente.

4) Diseñar el control de concurrencia para las transacciones. La mayoría de los sistemas de manejo de base de datos incorporan este control. El sistema pudiera necesitar la posibilidad de reintentar las transacciones fallidas varias veces antes de cancelar la transacción.

3.- DISEÑO DE OBJETOS

La fase de diseño de objetos determina las definiciones completas y asociaciones utilizadas en la implementación, así como las interfases y algoritmos de los métodos empleados para implementar las operaciones. Asimismo, agrega clases de objetos internas exclusivas de la implementación y optimiza las estructuras de datos y algoritmos. Esta fase es equivalente a la fase de diseño preliminar del tradicional ciclo de vida del desarrollo de sistemas.

Es en el diseño donde se produce un cambio en el énfasis de los conceptos del dominio de la aplicación hacia conceptos de computadora. Los objetos descubiertos durante el análisis sirven como esqueleto del diseño, aunque el diseñador debe elegir entre distintas alternativas de implementación con el objeto de minimizar el tiempo de ejecución, uso de memoria, etc. Las operaciones identificadas durante el análisis deben expresarse como algoritmos, descomponiendo las operaciones complejas en operaciones internas más simples. Las clases, atributos y asociaciones deben implementarse como estructuras de datos específicas. Las nuevas clases de objetos deben introducirse para almacenar resultados intermedios durante la ejecución de un programa.

Los modelos de objetos, funcional y dinámico son los productos básicos que se requieren para llevar a cabo la etapa de diseño.

El modelo de objetos describe las clases de objetos en el sistema incluyendo los atributos y operaciones que soporta. La información del modelo de objetos debe estar presente en el diseño de alguna forma. Generalmente el mejor y más simple enfoque consiste en llevar las clases directamente desde el análisis al diseño. El diseño de objetos se convierte así en un proceso de agregación de detalles y elaboración de decisiones de implementación. En ocasiones un objeto en el análisis no aparece de forma explícita en el diseño, sino que se encuentra distribuido entre otros objetos para mejorar la eficiencia de la implementación posterior. Frecuentemente se agregan también, clases nuevas y redundantes para mejorar la eficiencia.

El modelo funcional describe las operaciones que el sistema debe implementar. Durante el diseño se debe decidir la forma en la que cada operación debe implementarse, eligiendo el algoritmo adecuado para la operación y descomponiendo las operaciones complejas en operaciones más simples. Esta descomposición es un proceso iterativo que debe repetirse sucesivamente hasta los niveles más bajos de abstracción. Para elegir los algoritmos adecuados y el nivel de descomposición, se deben tomar en cuenta factores importantes de optimización, tales como la facilidad de implementación y el desempeño óptimo deseado.

El modelo dinámico describe la forma en la que el sistema responde a los eventos externos. La estructura de control para un programa se deriva primordialmente del modelo dinámico. El flujo de control dentro de un programa puede efectuarse de forma explícita (a través de un programa interno de trabajo que reconoce eventos y los relaciona con llamadas a operaciones) o implícitamente (eligiendo algoritmos que ejecutan las operaciones en el orden especificado por el modelo dinámico).

Al elegir la arquitectura del sistema, se debe considerar la estrategia de implementación. Se debe elegir también el flujo de control y datos a través del sistema y dividir el sistema en subsistemas. Si la arquitectura involucra varios procesadores, se debe decidir como serán distribuidos los objetos en los distintos procesadores. La elección de la arquitectura influirá también en la decisión de cómo transformar los eventos en operaciones.

LOS PASOS DEL DISEÑO DE OBJETOS

3.1. Combinar los tres modelos para obtener las operaciones en las clases.
3.2. Diseño de los algoritmos para implementar las operaciones.
3.3. Optimización de las rutas de acceso a los datos.
3.4. Implementación de controles para interacciones externas.
3.5. Ajuste de la estructura de las clases para incrementar la herencia.
3.6. Diseño de asociaciones.
3.7. Determinación de la representación de objetos.
3.8. Empacar las clases y asociaciones en módulos.

3.1. COMBINACIÓN DE LOS TRES MODELOS.

Después de concluir el análisis se obtienen los modelos de objetos, dinámico y funcional, aunque de los tres modelos, el modelo de objetos es la base principal alrededor de la cual se construye el diseño. El modelo de objetos del análisis puede no mostrar operaciones. El diseñador debe convertir las acciones y actividades del modelo dinámico y los procesos del modelo funcional en operaciones conectadas a las clases en el modelo de objetos.

INTERPRETACIÓN DEL DIAGRAMA DE ESTADOS.

Cada diagrama de estados describe la historia de un objeto. Una transición en el diagrama es un cambio de estado y se transforma en una operación del objeto. Se puede asociar una operación con cada evento recibido por un objeto. En un diagrama de estados, la acción ejecutada por una transición depende tanto del evento como del estado del objeto; en consecuencia un algoritmo que implementa una operación depende también del estado del objeto.

El evento enviado por un objeto pudiera representar una operación en otro objeto. Los eventos frecuentemente ocurren en pares, donde el primer evento arranca una acción y el segundo evento regresa el resultado o la señal de la terminación de la acción.

INTERPRETACIÓN DEL DIAGRAMA DE FLUJO DE DATOS.

Una acción o actividad iniciada por una transición en un diagrama de estados puede expandirse en un diagrama de flujo entero en el modelo funcional. La red de procesos dentro de un diagrama de flujo de datos representa el cuerpo de una operación. Los flujos en el diagrama son valores intermedios en la operación. El diseñador debe convertir la estructura gráfica del diagrama en una secuencia lineal de pasos en un algoritmo. El proceso en el diagrama de flujo de datos constituye una suboperación. Algunos de ellos, pero no necesariamente todos, pueden ser operaciones en el objeto destino original o en otros objetos.

Se puede determinar el objeto destino de una suboperación observando los siguientes criterios :

- Si el proceso extrae el valor a partir de un flujo de entrada, el destino es el mismo flujo de entrada.
- Si el proceso tiene un flujo de entrada y un flujo de salida del mismo tipo, y el valor de salida representa una versión actualizada del flujo de entrada, entonces el flujo de entrada-salida es el destino.
- Si un proceso construye un valor de salida a partir de diversos valores de entrada, entonces la operación es una operación de clase (constructor) en la clase de salida.
- Si un proceso tiene una entrada de un almacén de datos, o una salida hacia él, entonces el almacén de datos es el destino del proceso.

Cuando cualquier clase o clases de objetos alimentan operaciones internas a la operación de una clase destino, a esta última se le denomina "cliente".

3.2. DISEÑO DE ALGORITMOS.

Cada operación especificada en el modelo funcional debe formularse como un algoritmo. La especificación del análisis dice lo que la operación hace desde el punto de vista de sus clientes, pero el algoritmo muestra como se hace. El algoritmo puede subdividirse en llamadas o en operaciones más simples, hasta que las operaciones de más bajo nivel sean lo suficientemente simples para implementarlas sin necesidad de un refinamiento posterior.

El diseñador del algoritmo debe:

- Seleccionar los algoritmos que minimicen el costo de implementación de las operaciones.
- Seleccionar las estructuras de datos adecuadas a los algoritmos.
- Definir nuevas clases internas y operaciones de ser necesarias.
- Asignar la responsabilidad de las operaciones a las clases apropiadas.

3.2.1. SELECCIÓN DE ALGORITMOS.

Muchas de las operaciones son lo suficientemente simples que su especificación en el modelo funcional constituye de hecho un algoritmo satisfactorio debido a que la descripción de lo que se hace también muestra la forma de hacerlo.

La mayoría de las funciones poseen definiciones matemáticas o procedurales simples. Frecuentemente una definición simple es también el mejor algoritmo para calcularla.

Algunas consideraciones en la selección entre algoritmos alternativos son:

- Complejidad computacional.- Es esencial considerar la forma en la que el tiempo de ejecución se incrementa conforme aumenta el número de entradas y el costo de procesamiento de cada una de ellas.

- Facilidad de implementación.- Resulta perjudicial mejorar el desempeño de funciones no críticas si éstas pueden implementarse rápidamente como algoritmos más simples.

- Flexibilidad.- La mayoría de los programas requieren extenderse tarde o temprano. Un algoritmo altamente optimizado frecuentemente sacrifica la facilidad de modificación al mismo. Una altemativa para contrarrestar esta situación consiste en desarrollar 2 implementaciones para las operaciones críticas. Una basada en un algoritmo simple e ineficiente, que pueda implementarse rápidamente y ser utilizado para validar el sistema; y un algoritmo complicado pero eficiente, cuya correcta implementación pueda checarse contra el algoritmo simple.

3.2.2. SELECCIÓN DE ESTRUCTURAS DE DATOS.

La selección de los algoritmos realizada involucra la selección de las estructuras de datos con las que trabajarán. Durante el análisis el trabajo se concentró en la estructura lógica de la información en el sistema, pero es durante el diseño donde se eligen las estructuras de datos que permitan elaborar algoritmos eficientes. Las estructuras de datos no agregan

información al modelo del análisis, aunque organizan los datos en una forma conveniente para los algoritmos que los utilizan. Dichas estructuras de datos incluyen arreglos, listas, colas, filas, conjuntos, diccionarios, árboles, etc.

3.2.3. DEFINICIÓN DE CLASES Y OPERACIONES INTERNAS

Durante la elaboración de los algoritmos, pueden ser necesarias nuevas clases de objetos para guardar resultados intermedios o temporales. También pueden inventarse nuevas operaciones de bajo nivel durante las descomposiciones de las operaciones de alto nivel.

Una operación compleja puede definirse en términos de sus operaciones de más bajo nivel en objetos más simples. Estas últimas pueden definirse durante el diseño de objetos dado que la mayoría de ellas no son visibles de forma externa.

Cuando se llega a este punto durante la fase de diseño, se puede tener necesidad de agregar nuevas clases que no fueron mencionadas explícitamente en la descripción del problema. Estas clases de nivel más bajo son los elementos de implementación sobre los cuales se construyen las clases de la aplicación.

3.2.4. ASIGNACIÓN DE RESPONSABILIDADES PARA OPERACIONES.

Muchas operaciones tienen objetos destino que resultan obvios, pero algunas operaciones pueden ejecutarse en diversas partes en un algoritmo, por alguno de los distintos objetos involucrados.

Cuando una clase tiene significado en el mundo real, sus operaciones son generalmente claras. Durante la implementación, sin embargo, las clases internas introducidas no corresponden a objetos en el mundo real. A partir de que las clases internas son inventadas con fines de implementación, estas resultan algo arbitrarias, y sus límites son más una cuestión de conveniencia, que una necesidad lógica.

3.3. OPTIMIZACIÓN DEL DISEÑO.

El modelo de diseño básico utiliza el modelo de análisis como marco para la implementación. El modelo de análisis captura la información lógica del sistema, mientras el modelo del diseño debe agregar detalles para soportar un eficiente acceso de información. Un análisis semánticamente correcto pero ineficiente, puede ser optimizado para hacer la implementación más eficiente. Sin embargo, el diseño optimizado es menos claro y tiene menos posibilidades de ser reutilizarlo en otro contexto. El diseñador debe lograr un balance apropiado entre eficiencia y claridad.

Durante la optimización del diseño, se debe:

1. Agregar asociaciones redundantes para minimizar el costo de los accesos.
2. Reorganizar el trabajo computarizado para lograr una mayor eficiencia.
3. Almacenar los atributos derivados para evitar reprocesar expresiones complicadas.

3.3.1. AGREGAR ASOCIACIONES REDUNDANTES PARA ACCESOS EFICIENTES.

Durante el análisis, no es deseable la redundancia en la red de asociaciones, dado que las asociaciones redundantes no agregan ninguna información adicional. Durante el diseño, sin embargo, se evalúa la estructura del objeto para una implementación. Las asociaciones que fueron útiles durante el análisis no forman necesariamente la red de trabajo más eficiente cuando se consideran los patrones de acceso y frecuencias relativas de distintos tipos de acceso.

Para analizar el uso de las trayectorias en la red de asociaciones, se debe examinar cada operación y observar las asociaciones que debe recorrer para obtener su información. Se deben identificar tanto las asociaciones que son recorridas en ambos sentidos, como aquellas que son recorridas en una sola dirección.

Para cada operación se debe tomar en cuenta:

- La frecuencia con que es llamada.

- Su costo de ejecución.

- La fracción de éxitos en la clase final al aplicar un criterio de selección. Si la mayoría de los objetos son rechazados durante el recorrido por alguna razón, el ciclo probado resulta ineficiente para encontrar objetos destino.

3.3.2. REORGANIZAR EL ORDEN DE EJECUCIÓN.

Después de ajustar la estructura del modelo de objetos para optimizar recorridos frecuentes, el siguiente paso es optimizar el algoritmo en sí. Hasta este momento, las estructuras de datos y algoritmos están directamente relacionadas entre sí, aunque la estructura de datos debe considerarse primero.

Una clave para optimizar el algoritmo es eliminar rutas muertas tan pronto como sea posible. En algunas ocasiones el orden de un ciclo debe invertirse de la especificación original descrita en el modelo funcional.

3.3.3. GUARDAR ATRIBUTOS DERIVADOS PARA EVITAR REPROCESAMIENTO.

Los datos que son redundantes por derivarse de otros datos, pueden ser almacenados en su forma procesada para evitar la sobrecarga de volverlo a procesar. Se pueden definir nuevos objetos y clases para retener esta información. La clase que contiene los datos almacenados debe actualizarse si alguno de los objetos de los que depende cambia.

Los atributos derivados deben ser actualizados cuando los valores base cambian. Existen 3 formas para reconocer cuando se necesita una actualización:

1) Por actualización explícita.
2) Procesamiento periódico.
3) Uso de valores activos.

Actualización explícita.- Cada atributo derivado en términos de uno o más objetos base. El diseñador determina los atributos derivados que serán afectados por cada cambio a un atributo base e inserta el código dentro de la operación de actualización en el objeto base para actualizar de forma explícita los atributos derivados que dependen de él.

Procesamiento periódico.- Algunas ocasiones es posible simplemente reprocesar todos los atributos derivados de forma periódica; sin tener que hacerlo cada vez que un valor base cambia. El reprocesamiento de todos los atributos derivados puede ser más eficiente que la actualización incremental, dado que los atributos derivados pudieran depender de diversos atributos base y debieran actualizarse más de una vez empleando el enfoque incremental. Asimismo, el procesamiento periódico es más simple que la actualización explícita y menos susceptible de fallas. Por otra parte, si el conjunto de datos modifica de forma incremental a algunos cuantos objetos a la vez, el procesamiento periódico no resulta práctico debido a que muchos atributos derivados deben ser procesados cuando solo algunos son afectados.

Valores activos.- un valor activo es un valor que tiene valores dependientes. Cada valor dependiente se autoregistra con el valor activo, el cual contiene un conjunto de valores dependientes y operaciones de actualización.

3.4. IMPLEMENTACIÓN DE CONTROL.

El diseñador debe refinar la estrategia de implementación de los modelos de estado y eventos presentes en el modelo dinámico. Como parte del diseño del sistema, se seleccionará la estrategia básica para realizar el modelo dinámico. Durante el diseño de objetos se debe detallar esta estrategia.

Existen tres enfoques básicos para implementar el modelo dinámico:

1. Uso de la ubicación dentro del programa para mantener el estado (Sistemas dirigidos por procedimientos).
2. Implementación directa de un mecanismo de máquinas de estado (Sistemas dirigidos por eventos).
3. Uso de tareas concurrentes.

3.4.1. LOS ESTADOS COMO UBICACIÓN DENTRO DE UN PROGRAMA.

Este es un enfoque tradicional de representación de control dentro de un programa. La ubicación de control dentro de un programa define de forma explícita el estado del programa. Cualquier máquina de estado finito puede implementarse como un programa. Cada transición de estado corresponde a una instrucción de entrada. Después de que la entrada es leída, el programa se ramifica dependiendo del evento de entrada recibida. La carencia de modularidad de este enfoque es su más grande desventaja.

Una técnica de conversión de un diagrama de estado a código, es el siguiente:

1) Identificar la ruta de control principal. Se parte del estado inicial, se identifica una ruta a través del diagrama que corresponde a la secuencia de eventos normalmente esperados. Se escriben los nombres de los estados a lo largo de toda la trayectoria como una secuencia lineal y se convierte en secuencias de instrucciones de un programa.

2) Identificar trayectorias alternas que se ramifican desde la ruta principal y se reúnan con ella después. Estas se convertirán en instrucciones condicionales en el programa.

3) Identificar las trayectorias de regreso que se ramifiquen desde el ciclo principal y que se reúnan con el posteriormente. Estas se convertirán en ciclos en el programa. Si existen múltiples rutas de retroceso que no se cruzan, estas se convertirán en ciclos anidados.

4) Los estados y transiciones que restan corresponden a condiciones de excepción. Estas pueden manejarse por medio de diversas técnicas incluyendo subrutinas de error, manejo de excepciones, uso de banderas de status, etc.

3.4.2. MÁQUINAS DE ESTADO.

El enfoque más directo para implementar control es tener una forma de representar y ejecutar de forma explícita máquinas de estado.

Este enfoque permite avanzar rápidamente del modelo de análisis hacia un prototipo del sistema, a partir de una definición de las clases del modelo de objetos y máquinas de estado del modelo dinámico.

La creación de mecanismos de máquinas de estado no resulta complicado si se utiliza un lenguaje orientado a objetos y debe considerarse como una alternativa práctica si no se cuenta con un paquete que maneja máquinas de estado.

3.4.3. CONTROL COMO TAREAS CONCURRENTES.

Un objeto puede implementarse como tarea en un lenguaje de programación o en un sistema operativo. Este enfoque es el más general, ya que preserva la concurrencia inherente de objetos reales. Los eventos se implementan como llamadas entre tareas empleando las facilidades del lenguaje o del sistema operativo. Las tareas usan la localización dentro del programa para mantener el rastro de su estado.

Algunos lenguajes como "Concurrent Pascal" o "Concurrent C ++", soportan concurrencia, aunque la aceptación de tales lenguajes en ambientes de producción todavía esta limitada. Los lenguajes orientados a objetos más importantes no soportan todavía el manejo de concurrencia.

3.5. AJUSTE DE LA HERENCIA

Conforme el diseño de objetos avanza, las definiciones de clases y operaciones pueden ser ajustadas para aumentar el grado de herencia. El diseñador debe:

1. Reorganizar y ajustar las clases y operaciones para incrementar el grado de herencia.

2. Abstraer el comportamiento común de grupos de clases.
3. Utilizar la técnica de delegación para compartir comportamiento cuando la herencia sea válida semánticamente.

3.5.1. REORGANIZACIÓN DE CLASES Y OPERACIONES.

En algunas ocasiones la misma operación está definida a través de diversas clases y puede fácilmente ser heredada de un antecesor común, pero frecuentemente las operaciones en diferentes clases son similares, más no idénticas. Por medio de modificaciones parciales de las definiciones de operaciones o clases, las operaciones pueden ajustarse de tal forma que puedan ser cubiertas por medio de una sola operación heredada.

Antes de que la herencia pueda ser utilizada, cada operación debe tener la misma interfase y semántica, así como el mismo número y tipo de argumentos. Los siguientes tipos de ajuste pueden utilizarse para incrementar la posibilidad de manejar herencia:

- Algunas operaciones pudieran tener menos argumentos que otras. En este caso, se pueden agregar argumentos aunque para algunas clases se ignoren al momento de ejecución.

- Algunas operaciones pudieran tener menos argumentos debido a que son casos especiales de argumentos más generales. Se implementan en este caso las operaciones haciendo la llamada a la operación general con los valores de parámetros apropiados.

- Atributos similares en diferentes clases pudieran tener diferentes nombres. Se deben dar a los atributos el mismo nombre y moverlos a una clase común superior.

- Una operación pudiera estar definida en diversas clases en un grupo pero estar indefinidas en otras clases. En este caso se les define una clase superior común y se declaran como no-operaciones en las clases donde estaban indefinidas.

3.5.2. ABSTRACCIÓN DEL COMPORTAMIENTO COMÚN.

Las oportunidades para utilizar herencia no son siempre reconocidas durante la fase de análisis; vale la pena reexaminar el modelo de objetos buscando elementos comunes entre clases. En adición, las nuevas clases y operaciones se definen durante el diseño. Si un conjunto de operaciones y/o atributos parecen ser repetidos en dos clases, es probable que las dos clases sean variaciones especializadas de la misma cosa al ser vistas desde un nivel más alto de abstracción.

Cuando un comportamiento común es identificado, se puede crear una superclase común que implementa las cualidades compartidas, dejando solamente las cualidades especializadas en las subclases. Esta transformación del modelo de objetos es llamada abstracción hacia superclases o comportamientos comunes.

La abstracción de superclases brinda el beneficio de permitir compartir atributos y operaciones, así como la reutilización del diseño para otros proyectos y la modularidad.

3.5.3. USO DE DELEGACIÓN PARA COMPARTIR LA IMPLEMENTACIÓN.

La herencia es un mecanismo para implementar generalización, en la cual la conducta de una superclase es compartida por sus subclases. El compartir el comportamiento es justificable solamente cuando ocurre una verdadera relación de generalización, esto es, cuando se puede decir que la subclase es una forma de la superclase. Cuando una clase "B" hereda la especificación de la clase "A", se puede asumir que cada instancia de la clase "B" es una instancia de la clase "A", dado que se comportan de igual manera.

Cuando se intenta utilizar la herencia como técnica de implementación, se puede cumplir la misma meta en una forma más segura haciendo de una clase un atributo de otra. De esta forma, el objeto puede de una manera selectiva hacer uso de las funciones deseadas de otra clase empleando delegación en lugar de herencia.

La delegación consiste en atrapar la operación en un objeto y enviarla a otro que es parte o está relacionado al primero. Solamente las operaciones definitivas son delegadas al segundo objeto, evitando así el peligro de heredar operaciones significativas por accidente.

3.6. DISEÑO DE ASOCIACIONES.

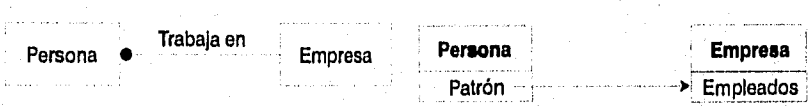
Durante la fase de diseño de objetos se debe formular una estrategia para implementar las asociaciones descritas en el modelo de objetos. Se puede seleccionar una estrategia global para implementar todas las asociaciones de forma uniforme, o se puede seleccionar una técnica particular para cada asociación, tomando en cuenta la forma en que será utilizada en la aplicación.

ANÁLISIS DE LOS RECORRIDOS DE APLICACIONES.

Se ha asumido hasta el momento que las asociaciones son inherentemente bidireccionales lo cual es verdadero en un sentido abstracto. Si algunas asociaciones en la aplicación son recorridas en una sola dirección, su implementación puede simplificarse. En un prototipo, siempre se utilizan asociaciones bidireccionales, de manera tal que se pueda agregar un nuevo comportamiento y expandir o modificar la aplicación rápidamente.

ASOCIACIONES DE UN SENTIDO.

Pueden implementarse como apuntadores (atributos que contienen la referencia a un objeto). Si la multiplicidad es "uno", se utiliza un solo apuntador. Si la multiplicidad es "muchos", se utiliza un conjunto de apuntadores.

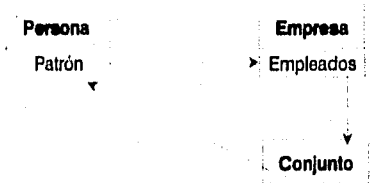


Implementación de asociación en un sentido con apuntadores

ASOCIACIONES DE DOS SENTIDOS.

Existen básicamente tres tipos de enfoques para implementarlas:

- Implementarlas como atributos en una dirección solamente y efectuar una búsqueda cuando se requiera el recorrido de regreso.
- Implementarlas como atributos en ambas direcciones.
- Implementarlas como una asociación distinta de objetos.



Implementación de asociación en dos sentidos con apuntadores

ATRIBUTOS DE LIGAS.

Si una asociación tiene atributos de ligas, su implementación depende de su multiplicidad. Si la asociación es de uno a uno, los atributos de liga pueden almacenarse como atributos en cualquiera de los objetos. Si la asociación es de muchos a uno, los atributos de liga pueden almacenarse como atributos del objeto con multiplicidad "muchos". Si la asociación es de muchos a muchos, el atributo de liga no puede asociarse en cualquiera de los objetos, el mejor método es generalmente implementar la asociación como una clase distinta en la cual cada instancia representa una liga con sus atributos.

3.7. REPRESENTACIÓN DE OBJETOS.

La implementación de objetos se realiza directamente la mayoría de las ocasiones, sin embargo, el diseñador debe elegir cuando debe usar tipos primitivos de representación de objetos, así como también debe saber cuando combinar grupos de objetos.

Las clases pueden ser definidas en términos de otras clases, pero eventualmente todo puede ser implementado en términos de la construcción de tipos de datos primitivos, tales como: enteros, cadenas y tipo numerados.

Representación alterna para un atributo

3.8 EMPACADO FÍSICO

Los programas están hechos a partir de unidades físicas discretas que pueden ser editadas, compiladas, importadas ó manipuladas. En algunos lenguajes como C y FORTRAN, las unidades son archivos fuente. Los lenguajes orientados a objetos tienen varios grados de empacado. En un gran proyecto, el cuidadoso particionamiento de una implementación en paquetes, es importante para permitir a diferentes personas cooperar en los trabajos del programa. El empacado involucra los siguientes puntos:

1. El ocultamiento de información interna desde la vista externa.
2. La coherencia de las entidades.
3. Construcción de módulos físicos.

3.8.1 OCULTAMIENTO DE INFORMACIÓN

Una meta del diseño, es tratar a las clases como cajas negras cuyas interfases externas son públicas pero cuyos detalles internos están ocultos.

El diseñador debe decidir que atributos deben ser accesibles desde afuera de la clase. Estas decisiones deben registrarse en el modelo de objetos agregando una anotación {Privada} en los atributos que serán escondidos.

Llevado al extremo, el método en una clase puede recorrer todas las asociaciones del modelo de objetos para localizar y accesar a otro objeto en el sistema. Esta visibilidad sin restricciones es valida durante el análisis, pero los métodos que conocen mucho del modelo entero resultan frágiles, dado que cualquier cambio en la representación los invalida.

Cada operación debe tener un conocimiento limitado del modelo entero, incluyendo la estructura de clases, asociaciones y operaciones. Los siguientes principios de diseño ayudaran a limitar el alcance del conocimiento de cualquier operación:

- Asignar a cada clase la responsabilidad de ejecutar operaciones y proporcionar información que le pertenezca.
- Llamar a una operación para accesar atributos pertenecientes a otra clase.
- Evitar recorrer asociaciones que no estén conectadas a la clase actual.
- Definir las interfases al nivel más alto posible.
- Esconder objetos externos a la frontera del sistema definiendo clases de interfase abstractas.
- Evitar la aplicación de un método al resultado de otro, a menos que la clase resultado sea realmente un proveedor de métodos para la clase que hace la llamada.

COHERENCIA DE ENTIDADES.

Un principio importante de diseño es el de la coherencia de entidades. Una entidad (clase, operación o módulo), se considera coherente si es organizada en un plan consistente y todas sus partes juntas satisfacen una meta común.

Un solo método debe contener una política y una implementación. La política es la elaboración de decisiones dependientes del contexto. La implementación es la ejecución de algoritmos totalmente especificados. La política involucra

la toma de decisiones, interacción con el mundo exterior, e interpretación de casos especiales y contiene a las instrucciones de entrada-salida, condicionantes y acceso a los almacenes de datos. Una implementación ejecuta exactamente la aplicación, sin tomar decisiones, suposiciones o desviaciones.

La separación de política e implementación incrementa grandemente la posibilidad de reutilización. Los métodos de implementación no contienen dependencias del contexto, por lo que pueden ser reutilizados con relativa facilidad.

Una clase no debe servir a muchos propósitos a la vez. Si es muy complicada debe dividirse empleando generalización o agregación.

CONSTRUCCIÓN DE MÓDULOS.

Los módulos deben ser definidos de tal forma que sus interfases sean mínimas y estén bien definidas. La interfase entre dos módulos consiste en la asociación que relaciona las clases de un módulo con la clase en otro, así como las operaciones que accesan a las clases a través de los límites del módulo.

La conectividad del modelo de objetos puede utilizarse como guía para particionar módulos. Una regla rígida es que las clases que están conectadas de manera cercana por asociaciones deben estar en el mismo módulo; mientras que las clases que no están conectadas fuertemente pueden estar en módulo separados.

Los módulos deben contar con algo de cohesión funcional o unidad de propósito y representar tipos similares de cosas en una aplicación.

DOCUMENTACIÓN DE LAS DECISIONES DEL DISEÑO.

El documento del diseño debe ser una extensión del documento de análisis de requerimientos. De esta forma el documento del diseño incluirá una

descripción revisada y más detallada del modelo de objetos tanto en forma gráfica como en forma textual.

Es conveniente separar el documento del diseño del documento del análisis, debido al cambio en el punto de vista de un usuario externo al del diseñador, ya que el documento del diseño contiene muchas optimizaciones y detalles de implementación.

RESUMEN DE LA METODOLOGÍA

La distinción entre el análisis y el diseño en ocasiones puede parecer como arbitraria y confusa. Las siguientes reglas pueden guiar las decisiones acerca del alcance adecuado del análisis y el diseño.

El modelo de análisis debe incluir información que sea significativa desde la perspectiva del mundo real y debe presentar una vista externa del sistema. Debe de ser entendible para el usuario y proporcionar bases útiles para obtener verdaderos requerimientos para el sistema.

En contraste, el modelo de diseño es llevado a cabo en base a la relevancia de la implementación. Por ello, el diseño debe ser razonablemente eficiente y práctico de codificar. En la práctica muchas partes del análisis frecuentemente pueden ser implementadas directamente sin cambio; por lo tanto, puede existir un considerable traslape entre el análisis y el diseño.

ANÁLISIS

La meta del análisis es desarrollar un modelo de lo que el sistema hará. El modelo está expresado en términos de objetos y relaciones, flujo dinámico de control y transformaciones funcionales. Sus etapas de esta fase se indican a continuación :

1) Escribir u obtener una descripción inicial del problema.

2) Construir un modelo de objetos.
- Identificar clases de objetos.
- Iniciar el diccionario de datos incluyendo la descripción de las clases, atributos y asociaciones.
- Agregar asociaciones entre clases.
- Agregar atributos para objetos y ligas.
- Organizar y simplificar clases de objetos utilizando la herencia.
- Probar las rutas de acceso empleando escenarios e iterar los pasos anteriores según sea necesario.

- Agrupar las clases dentro de módulos.

⇒ Modelo de objetos = Diagrama de modelo de objetos + Diccionario de datos.

3) Desarrollar un modelo dinámico.

- Preparar escenarios de secuencias típicas de interacción.
- Identificar eventos entre objetos y preparar un seguimiento de eventos para cada escenario.
- Preparar un diagrama de flujo de eventos para el sistema.
- Desarrollar un diagrama de estado para cada clase que tenga un comportamiento dinámico importante.
- Revisar la consistencia y conclusión de eventos compartidos entre los diagramas de estados.

⇒ Modelo dinámico = Diagramas de estado + Diagrama global de flujo de eventos.

4) Desarrollar un modelo funcional.

- Identificar valores de entrada y salida.
- Utilizar diagramas de flujo como sea necesario para mostrar dependencias funcionales.
- Describir que hace cada función.
- Identificar restricciones.
- Especificar criterios de optimización.

⇒ Modelo funcional = Diagramas de flujo de datos + Restricciones.

5) Verificar, iterar y refinar los tres modelos.

- Agregar al modelo de objetos operaciones clave que hayan sido descubiertas durante la preparación del modelo funcional. No se deben mostrar todas las operaciones durante el análisis, dado que podría hacer confuso el modelo de objetos; solo se deben mostrar las operaciones más importantes.
- Verificar que las clases, las asociaciones, los atributos y las operaciones sean consistentes y estén completas en el nivel de abstracción

seleccionado. Comparar los tres modelos con la definición del problema y probar los modelos empleando escenarios.

- Desarrollar escenarios más detallados (incluyendo condiciones de error) como variaciones de los escenarios básicos. Utilizar escenarios del tipo "Qué pasa si...?" para verificaciones posteriores de los modelos.
- Iterar los pasos anteriores tantas veces sea necesario para completar el análisis.

⇒ **Documento del Análisis** = Definición del problema + Modelo de Objetos + Modelo Dinámico + Modelo Funcional.

DISEÑO DEL SISTEMA.

Durante el diseño del sistema, la estructura de alto nivel del sistema es elegida. Las etapas de esta fase se indican a continuación :

1) Organizar el sistema en subsistemas.

2) Identificar concurrencia inherente en el problema.

3) Asignar los subsistemas a procesadores y tareas.

4) Seleccionar la estrategia básica para implementar los almacenes de datos en términos de estructuras de datos, archivos y bases de datos.

5) Identificar recursos globales y determinar mecanismos para controlar los accesos a éstos.

6) Elegir un enfoque para implementar controles.
   - Utilizar la posición dentro del programa para mantener el estado ó.
   - Implementar directamente una máquina de estados ó.
   - Emplear tareas concurrentes.

7) Considerar condiciones límite.

8) Establecer prioridades de intercambio.

⇒ Documento del Diseño del Sistema = Estructura de arquitectura básica del sistema así como decisiones de estrategia de alto nivel

DISEÑO DE OBJETOS.

Durante el diseño de objetos se revisa el modelo de análisis y se produce una base detallada para la implementación. Se toman decisiones de lo que se necesita para realizar el sistema sin descender a detalles propios de un lenguaje o sistema de bases de datos en particular. El diseño de objetos inicia un cambio de la orientación al mundo real del modelo de análisis, en dirección a una orientación computacional requerida para una implementación práctica.

1) Obtener las operaciones para el modelo de objetos, a partir de los modelos funcional y dinámico.
   - Encontrar una operación para cada proceso en el modelo funcional.
   - Definir una operación para cada evento en el modelo dinámico dependiendo de la implementación de control.

2) Diseñar los algoritmos para implementar las operaciones.
   - Elegir algoritmos que minimicen el costo de la implementación de operaciones.
   - Seleccionar estructuras de datos adecuadas a los algoritmos.
   - Definir nuevas clases internas y operaciones como sea necesario.
   - Asignar responsabilidades para operaciones que no estén claramente asociadas con una clase.

3) Optimizar rutas de acceso a datos.
   - Agregar asociaciones redundantes para minimizar el costo de acceso.
   - Reorganizar los cálculos para una mejor eficiencia.
   - Guardar valores derivados para evitar recálculos y expresiones complicadas.

4) Implementar el control de software fortaleciendo el enfoque elegido durante el diseño del sistema.

5) Ajustar la estructura de clases para incrementar la herencia.

- Reorganizar y ajustar clases y operaciones.
- Abstraer comportamientos comunes de los grupos de clases.
- Delegar para compartir comportamiento donde la herencia sea invalida semánticamente.

6) Diseñar la implementación de las asociaciones.
- Analizar el recorrido de las asociaciones.
- Implementar cada asociación ya sea como un objeto distinto o agregando atributos de objetos hacia una o ambas clases en la asociación.

7) Determinar la representación exacta de los atributos de los objetos.

8) Agrupar las clases y asociaciones en módulos.

⇒ Documento del Diseño de Objetos = Modelo de Objetos Detallado + Modelo Dinámico + Modelo Funcional Detallado

DEL DISEÑO A LA IMPLEMENTACIÓN

Escribir el código representa una extensión del proceso de diseño. Escribir el código debe ser una tarea casi mecánica, dado que las decisiones más complicadas debieron ya ser realizadas durante el diseño. El código debe de ser una simple traducción de las decisiones del diseño a las peculiaridades de un lenguaje en particular. Se pueden tomar decisiones mientras se escribe el código, pero éstas solo deben afectar pequeñas partes del programa, de manera tal que estos puedan ser modificados fácilmente.

IMPLEMENTACIÓN USANDO UN LENGUAJE DE PROGRAMACIÓN

La mayor parte de los lenguajes de programación son capaces de expresar los tres aspectos de la especificación del software: estructura de datos, flujo dinámico de control y transformaciones funcionales.

La estructura de datos es expresada en un subconjunto declarativo (no procedural) de un lenguaje.

El flujo de control puede ser expresado proceduralmente (condicionales, ciclos y llamadas) o de forma no procedural (reglas, tablas y restricciones). Los lenguajes tradicionales son puramente procedurales, aunque el programador puede implementar construcciones no-procedurales como datos. Los lenguajes no procedurales, tales como sistemas basados en reglas, sistemas de mantenimiento de restricciones, o lenguajes de programación lógica, soportan diferentes formas de organización de programas.

El soporte para tareas de control concurrentes está ausente en la mayoría de los lenguajes. El trabajo multitareas y las comunicaciones entre procesos son manejados por los sistemas operativos modernos, pero deben de ser accesados desde los programas mediante riesgosas llamadas a subrutinas. La concurrencia puede ser simulada también dentro de los programas usando rutinas comunes, módulos de control o manejadores de eventos.

Las transformaciones funcionales son expresadas en términos de operadores primitivos del lenguaje, así como con llamadas a subprogramas.

La forma más fácil de implementar un diseño orientado a objetos, es usando un lenguaje orientado a objetos, pero hasta los lenguajes orientados objetos varían en su grado de soporte de conceptos orientados a objetos.

Aun cuando un lenguaje no orientado a objetos puede ser utilizado, un diseño de esta naturaleza es benéfico. Los conceptos orientados a objetos pueden ser manejados con construcciones de un lenguaje no orientado a objetos. El empleo de un lenguaje no orientado a objetos requiere de mayor cuidado y disciplina para conservar la estructura orientada a objetos del programa.

IMPLEMENTACIÓN USANDO UN SISTEMA MANEJADOR DE BASES DE DATOS

Cuando la preocupación principal es el constante acceso de datos, en lugar de las operaciones con ellos, una base de datos es muchas veces la forma mas apropiada de implementar. El principal enfoque de una base de datos es la estructura y las restricciones en los datos. Los comandos típicos de una base de datos operan con conjuntos de datos. Las operaciones en una base de datos son mucho menos procedurales que las instrucciones de lenguajes de programación convencional, aunque son más procedurales que los lenguajes basados en reglas. Los sistemas de bases de datos proporcionan operaciones concurrentes con los datos por diferentes usuarios como parte de su estructura fundamental.

Algunos de los recientes sistemas de bases de datos orientados a objetos tratan de integrar un lenguaje orientado a objetos con una base de datos en un paquete simple. Aunque los sistemas de base de datos orientadas a objetos prometen un mejor desempeño y un uso más fácil, aun no maduran como lo han hecho los sistemas de bases de datos relacionales y pueden presentar problemas de integración con aplicaciones convencionales existentes.

COMPARACIÓN ENTRE METODOLOGÍAS

En esta sección se resumen otros enfoques de Ingeniería de Software y se comparan contra la técnica de Modelaje de Objetos (OMT).

ANÁLISIS Y DISEÑO ESTRUCTURADO (SA/SD)

Actualmente, las metodologías de ingeniería de software mas utilizadas, son las basadas en diagramas de flujo de datos. Diversas variaciones del enfoque de flujo de datos son utilizadas en la práctica. La metodología del Análisis y Diseño Estructurado (SA / SD) es un enfoque representativo del enfoque de flujo de datos. Yourdon, Constantine, DeMarco y otros han escrito acerca de (SA /SD). Ward y Mellor han agregado extensiones para el manejo de aplicaciones de tiempo real a (SA/SD). Esta metodología se encuentra bien documentada y es aplicable a muchos problemas.

Las metodologías (OMT) y (SA/SD) incorporan componentes de modelaje similares. Ambas metodologías soportan distintas vistas del sistema, como son los modelos de objetos, dinámico y funcional. Estas metodologías difieren en el grado de énfasis que ponen en varios de los componentes de modelaje. En la metodología OMT se enfatiza el modelo de objetos. La comprensión de los objetos del mundo real y sus relacionan proporcionan el contexto para entender el comportamiento dinámico y funcional. En la metodología SA/SD, el énfasis se encuentra en la descomposición funcional, dado que el sistema es visto como proveedor de una o más funciones hacia el usuario final.

COMPARACIÓN CON (OMT)

(SA/SD) y (OMT) tienen mucho en común, ambas metodologías usan construcciones de modelaje similares, la diferencia entre los dos es principalmente cuestión de estilo y énfasis. En el (SA/SD), el modelo funcional es el más importante. En contraste, (OMT) considera el modelo de

objetos como el mas importante después el modelo dinámico y finalmente el funcional.

(SA/SD) organiza un sistema alrededor de procedimientos. Por lo contrario (OMT) organiza el sistema alrededor de objetos del mundo real u objetos conceptuales que existen en la vista del mundo que tiene el usuario. Al emplear OMT, la mayoría de los cambios en los requerimientos son cambios en funciones en vez de ser en objetos, por lo tanto un cambio puede ser desastroso para el diseño basado en procedimientos. Los cambios en las funciones son fácilmente adaptados y no afectan al modelo de objetos, empleando OMT.

Un diseño realizado con SA/SD posee una clara definición de los límites del sistema. Sin embargo, resulta bastante complicado redefinir el diseño cuando los límites del sistema crecen. Resulta más fácil extender las fronteras de un sistema empleando OMT, ya que solo se tienen que agregar objetos y relaciones al modelo existente. Esto hace que un diseño hecho con OMT sea más adaptable al cambio.

La analogía directa entre objetos en un diseño orientado a objetos y los objetos del dominio del problema, conduce a sistemas que son más fáciles de entender. Esto hace que el diseño sea más intuitivo y simplifica notablemente la conversión de requerimientos a código. También se logra que el diseño pueda ser comprendido por personas que no forman parte necesariamente del equipo de trabajo del mismo diseño.

En (SA/SD) la descomposición de un proceso en subprocesos es algo arbitraria. Dependiendo del responsable, se pueden producir diferentes descomposiciones. En (OMT) la descomposición está basada en objetos provenientes del dominio del problema; es por ello que diferentes desarrolladores de programas diferentes tienden a descubrir objetos similares en el mismo dominio. Esto incrementa la facilidad de reutilizar componentes de un proyecto en futuros proyectos.

Un enfoque orientado a objetos integra de una mejor forma bases de datos con el código de programación. En contraste, una metodología de diseño procedural es inexperta para lidiar con bases de datos. (SA/SD) tiene

dificultad para unir el código de programación organizado funcionalmente con bases de datos.

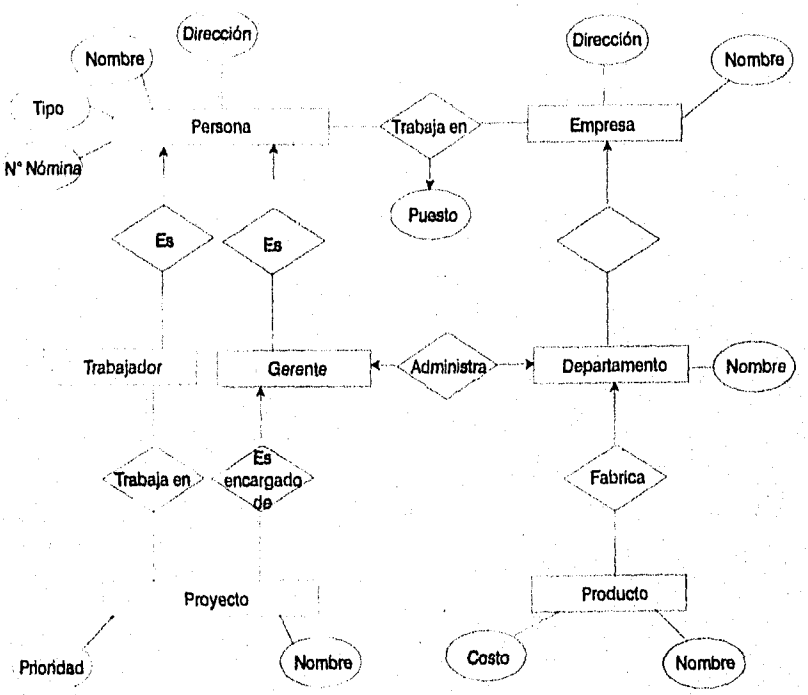


Diagrama de entidad relación

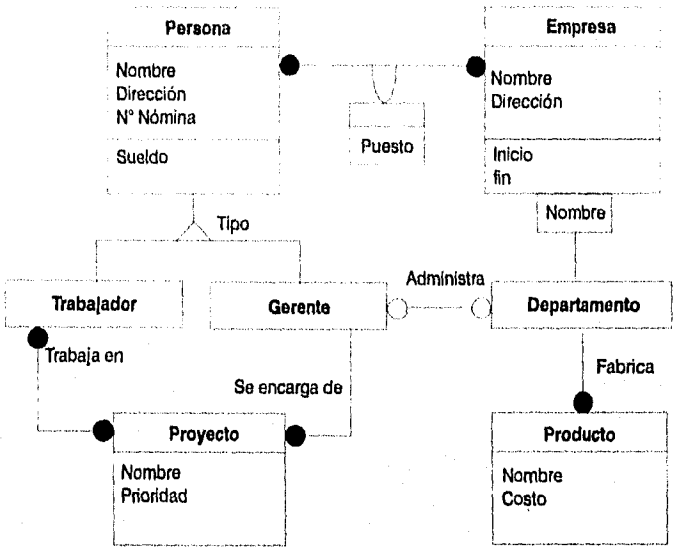


Diagrama de objetos de OMT

DESARROLLO ESTRUCTURADO DE JACKSON (JSD)

Esta es otra metodología que tiene diferente estilo a (SA/SD) o (OMT), (JSD) fue desarrollada por Michel Jackson y es especialmente popular en Europa. (JSD) no hace distinción entre análisis y diseño, en su lugar, agrupa ambas fases y las llama especificación. (JSD) divide el desarrollo del sistema en dos etapas: especificación e implementación. (JSD) primero determina el ¿Que? y posteriormente el ¿Como?.(JSD) Es especialmente útil para aplicaciones en las cuales el tiempo es importante.

(JSD) usa modelos gráficos como SA/SD o OMT, y otras técnicas, sin embargo es menos orientado gráficamente que las otras.

COMPARACIÓN CON OMT

Algunos autores se refieren a JSD como si fuese orientado a objetos. JSD comienza con una consideración del mundo real. El propósito del sistema es el de brindar funcionalidad y cómo esta funcionalidad se ajusta al mundo real. Un modelo JSD describe el mundo real en términos de entidades, acciones y ordenamientos de acciones. Las entidades aparecen generalmente como sustantivos y las acciones como verbos. Un desarrollo empleando JSD consiste de 6 pasos secuenciales :

1) Definir entidades y acciones
2) Definir Estructuras de Entidades.
3) Desarrollo del Modelo Inicial.
4) Desarrollo de Funciones
5) Establecimiento de Tiempos del Sistema
6) Implementación.

El enfoque JSD es muy complejo y difícil de comprender en su totalidad. JSD es más obscuro que las técnicas basadas en flujo de datos u orientadas a objetos.

Una razón de la complejidad de JSD es su fuerte confianza en el seudocódigo; los modelos gráficos son mas fáciles de entender. JSD es también complejo porque está especialmente diseñado para manejar problemas de tiempo real. Por este tipo de problemas, JSD puede producir un diseño superior. Sin embargo, la complejidad de JSD es innecesario para problemas simples.

JSD es útil para los siguientes tipos de aplicaciones:

- Sistemas concurrentes donde los procesos deben sincronizarse uno con otro.
- Sistemas de tiempo real
- Programando computadoras en paralelo. El paradigma de JSD basado en muchos procesos puede ser útil aquí.

JSD es débil para otras aplicaciones como:

- Análisis de alto nivel. JSD no brinda un fuerte entendimiento del problema. JSD es inefectivo en labores de abstracción y simplificación. JSD manipula meticulosamente los detalles pero no ayuda al desarrollador a captar la esencia del problema.
- Diseño de bases de datos.
- Software convencional corriendo bajo un sistema operativo. Las abstracciones de JSD hacia cientos o miles de procesos es confusa e innecesaria.

LENGUAJES DE PROGRAMACIÓN ORIENTADOS A OBJETOS

CARACTERÍSTICAS DE LOS LENGUAJES ORIENTADOS A OBJETOS

Los lenguajes orientados a objetos varían en el grado en que soportan los conceptos de el enfoque Orientado a Objetos. No existe un solo lenguaje que cubra todas las necesidades de este enfoque. Algunos lenguajes examinados son : C++, Eiffel, Objective-C, Smalltalk y CLOS que están comercialmente disponibles. Otros tales como Trellis-Owl y DSM son lenguajes en investigación. Gemstone y Orion son lenguajes de bases de datos.

HERENCIA MÚLTIPLE

Algunos lenguajes, como C++, CLOS, Eiffel y DSM soportan herencia múltiple. Si el lenguaje seleccionado soporta esta característica, es factible traducir directamente el diseño a la implementación.

La herencia múltiple introduce la posibilidad de conflictos entre los nombres de atributos y los nombres de operaciones. Una clase puede tener más de un antecesor con el mismo atributo. Existen diversos enfoques que los lenguajes adoptan para resolver este conflicto. Por ejemplo, el compilador de Eiffel rechaza programas con tales conflictos. CLOS, por otra parte, posee un protocolo propio que resuelve estas situaciones. De cualquier forma, es conveniente evitar tales conflictos desde el diseño.

LIBRERÍAS DE CLASES

La mayoría de los lenguajes orientados a objetos, incluyen una librería de clases genéricas muy útiles, que pueden ser usadas sin cambio, o ajustadas a una necesidad específica durante la creación de las subclases. La

disponibilidad de librerías de clases permite que muchos componentes no tengan que ser reimplementados por el programador.

Los tipos de clases más útiles implementan estructuras de datos de uso general, tales como conjuntos, arreglos, listas, colas, pilas, diccionarios, árboles, etc. Estas clases, denominadas como clases contenedoras, sirven como marco de trabajo para organizar conjuntos de otros objetos.

EFICIENCIA

Los lenguajes Orientados a Objetos tienen la mala reputación de su ineficiencia debido a que los primeros lenguajes (Smalltalk y lenguajes basados en LISP) trabajaban como intérpretes y no como compiladores. La disponibilidad de lenguajes compiladores y de intérpretes más eficientes ha dado a los desarrolladores más elementos en la elección del lenguaje apropiado. El uso de un lenguaje orientado a objetos con una desarrollada librería de clases frecuentemente permite producir código que corre más rápido en comparación con el código obtenido con un lenguaje no orientado a objetos.

Un aspecto de los lenguajes Orientados a Objetos que parece ineficiente es el método de resolución al momento de ejecución para implementar operaciones polimórficas. El método de resolución es el proceso que relaciona una operación dada en un objeto con un método específico. Esto podría parecer que se requiere una búsqueda en el árbol de herencia en el momento de ejecución para encontrar la clase que implemente la operación para un objeto dado. Sin embargo, la mayoría de los lenguajes orientados a objetos optimizan el mecanismo de búsqueda para hacerlo más eficiente. Esto es, que conforme la estructura de la clase permanezca sin cambio durante la ejecución del programa, el método adecuado para cada operación puede almacenarse localmente en la subclase.

En un lenguaje fuertemente tipificado, como C++, el costo del método de resolución en tiempo de ejecución puede ser reducido a una sola llamada a la estructura, lo cual resulta casi insignificante comparándolo contra el costo de llamar a un procedimiento. Cada clase posee una estructura que contiene métodos que son accesibles por objetos de la clase. Un apuntador a cada

método es almacenado en una posición conocida junto con la estructura. Un apuntador a la estructura del método para la clase es almacenado en cada objeto. El método de resolución es ejecutado recuperando la estructura del método.

TIPIFICACIONES. FUERTES Y DÉBILES

Los lenguajes orientados a objetos varían grandemente en su enfoque de tipificación. El término tipificación se refiere a si los valores de las variables y los atributos son meramente conocidos por ser objetos (tipificación débil) o pueden ser declarados más precisamente como pertenecientes a una clase o alguno de sus descendientes (tipificación fuerte). Smalltalk es un lenguaje débilmente tipificado (Todas las variables son objetos de clases no especificadas). Eiffel, DMS, Objective-C, y C++ son lenguajes fuertemente tipificados. CLOS y Objective-C soportan tipificaciones fuertes, pero no requieren su uso (Una variable que hace referencia a un objeto puede declararse de manera opcional, ya que el compilador se encarga de checar si es realmente utilizada o no).

La fuerte tipificación en un lenguaje sirve para dos propósitos : Proporciona un soporte activo al programador para detectar argumentos de métodos e instrucciones de asignación incorrectas. También incrementa las oportunidades de optimización.

Aunque las tipificaciones débiles son flexibles y poderosas, permiten prácticas de codificación peligrosas. Las modernas teorías de programación se orientan en la dirección hacia fuertes tipificaciones. La premisa detrás de las fuertes tipificaciones se basa en que resulta más fácil detectar fallas al momento de compilar un programa, en lugar de detectarlas al momento de ejecución.

MANEJO DE MEMORIA

La libertad con la cual los objetos pueden ser creados y accesados provoca un problemas cuando el espacio en memoria es requerido. La mayoría de los

lenguajes orientados a objetos reservan memoria a partir de la memoria disponible, en lugar de utilizar bloques fijos de memoria global o un stack. Un sistema con manejo de memoria dinámica puede correr fuera de memoria a menos que los objetos que ya no son necesarios sean descargados. El principal problema radica en determinar cuando un objeto ya no se necesario. La severidad del problema depende del tipo de aplicación y de la arquitectura de memoria existente. Muchos programas que tienen acceso a grandes cantidades de memoria virtual pueden pasar por alto estos problemas. Por otra parte, una aplicación altamente interactiva que se espera pueda correr indefinidamente, siempre debe considerar el manejo de memoria sin importar que tan grande sea el espacio de memoria disponible.

Existen dos enfoques para el manejo de memoria : Uno en el cual el manejo se hace automáticamente por el lenguaje en tiempo de ejecución, y otro hecho explícitamente a través de instrucciones de descarga escritas por el programador. El enfoque más utilizado es el de manejo automático, el cual releva al programador de la responsabilidad de decidir cuando liberar memoria .

ENCAPSULACIÓN

La encapsulación consiste en la separación de los aspectos externos de un objeto que son accesibles por otros objetos, de los detalles de implementación internos del objeto que están ocultos para otros objetos. La encapsulación permite modificar la implementación de una clase sin afectar a los clientes de dicha clase. De esta manera, puede modificarse la implementación para incrementar su desempeño, corregir algún problema, consolidar código, o portar la aplicación a un sistema diferente. La encapsulación no es exclusiva de los lenguajes orientados a objetos, pero la posibilidad de combinar estructuras de datos y comportamiento en una sola entidad hace que la encapsulación resulte más clara y poderosa, que en lenguajes convencionales los cuales separan la estructura de datos del comportamiento.

Una forma en la que la encapsulación puede ser violada ocurre cuando el código asociado con una clase accesa directamente los atributos de otra clase. Los acceso directos hacen suposiciones acerca del formato de almacenamiento y la localización de los datos. Estos detalles de

implementación deben esconderse dentro de una clase. Por ejemplo, el valor de un atributo puede ser calculado a partir de otra información. La forma más propia de accesar un atributo de otro objeto es "solicitándolo" a través de la operación del objeto, en lugar de simplemente "extraerlo". Muchos lenguajes, tales como Smalltalk, prohiben el acceso directo a los atributos de otro objeto. Otros como C++ o Trellis-Owl permiten que los atributos sean declarados ya sea privados o públicos. Eiffel proporciona quizás el control más preciso de encapsulación a través de la instrucción "export", la cual lista los atributos que pueden ser leídos y las operaciones que pueden se ejecutadas. La declaración de C++ "friend" permite una pérdida limitada de encapsulación para accesar a clases específicas o funciones.

La encapsulación puede ser violada a través del cuidadoso uso de la herencia. Una subclase hereda los atributos de su superclase, pero si los métodos de la subclase accesan directamente a los atributos de la superclase, entonces la encapsulación de la superclase es violada. CLOS, Owl, Eiffel y C++ permiten a una clase restringir la visibilidad de sus subclases.

Con frecuencia resulta útil escribir algunas operaciones "privadas" para uso interno por medio de otros métodos de la misma clase. Es deseable restringir la visibilidad de tales operaciones de tal forma que otras clases no puedan utilizarlas.

EMPAQUETADO

Una clase no es una construcción adecuada para sistemas de gran tamaño. La mayoría de los lenguajes orientados a objetos carecen de mecanismos de particionamiento para controlar la visibilidad entre clases. Se sugiere el uso de módulos que contengan clases como mecanismo de estructuración durante el análisis.

Un problema relacionado es que los lenguajes orientados a objetos generalmente requieren que los nombres de las clases sean únicos. Un caso típico ocurre cuando dos o más aplicaciones que fueron desarrolladas independientemente son combinadas. Por ejemplo, dos desarrolladores que trabajan independientemente definen versiones distintas de una clase llamada "Símbolo" con distintas operaciones y semántica.

En el lenguaje ADA y en CLOS, la construcción "package" proporciona los medios para estructurar un sistema en componentes separados con sus propios nombres. En ADA, los nombres declarados en la especificación de un paquete no son visibles fuera de éste, a menos que se solicite de forma explícita, permitiendo tanto el control del nombre como de las dependencias entre paquetes.

Los lenguajes orientados a objetos se podrían beneficiar con la adición de la posibilidad de empaquetado. En particular, la posibilidad de anidar los componentes de un sistema permite un mejor control de la visibilidad.

AMBIENTE DE DESARROLLO

Las herramientas que están disponibles para visualizar y editar código, compilación, depuración e integración de sistemas, tienen gran impacto en la productividad del programador. Las herramientas de soporte son especialmente importantes en un lenguaje orientado a objetos debido a la dificultad que implica la administración de sistemas de gran tamaño.

Un visualizador permite explorar el código fuente de una forma estructurada, pudiendo mostrar las clases que se encuentran presentes y las operaciones definidas para cada clase.

El compilador o el intérprete, según sea el caso, constituye la herramienta más importante de implementación. La velocidad con la que es factible hacer pequeños cambios a los programas y probarlos, puede verse afectada dependiendo de sí el código es compilado, interpretado o traducido. Un intérprete permite una depuración más flexible. Smalltalk y Lisp permiten a través del intérprete que la ejecución de un programa sea interrumpida, editar el código y continuar posteriormente le ejecución. Objective C ofrece un intérprete opcional en adición al compilador.

Algunos compiladores traducen un lenguaje orientado a objetos a un lenguaje intermedio (por ejemplo : C), el cual posteriormente es compilado. C++ fue implementado primeramente como traductor de C, aunque los compiladores de C++ actualmente están disponibles. La traducción de código puede

presentar problemas para los depuradores simbólicos dado qùe el código orientado a objetos puede resultar incomprensible. Al evaluar el depurador, se debe identificar si esté último muestra el código orientado a objetos original, o el código intermedio. Este último en consecuencia, resulta más difícil de entender. El depurador debe ser capaz de inspeccionar valores de atributos y evaluar expresiones orientadas a objetos.

Las herramientas de construcción de sistemas son esenciales para proyectos mayores, pero también resultan valiosas para programadores individuales. La construcción de sistemas puede consumir mucho tiempo si un simple cambio en un módulo requiere que sus módulos clientes sean recompilados también.

METADATOS

Los metadatos son datos acerca de los datos. Los metadatos que se encuentran presentes al momento de ejecución, permiten que una aplicación pueda evaluar e incluso modificar, sus propias estructuras y capacidades, incluyendo en ello las operaciones y objetos que soportan, los atributos que poseen, o los tipos de atributos.

Los lenguajes que contienen descriptores de clase explícitos (Smalltalk y DSM), contienen metadatos en tiempo de ejecución relativos a las clases. El objeto de una clase actúa como una plantilla para la creación de nuevas clases. La clase puede contener también descripciones de los atributos y operaciones, tales como el nombre, tipo o los argumentos.

En principio, una representación de los metadatos puede incrustarse directamente en el código. Sin embargo, si el diseño cambia por alguna razón, el código que depende de los datos incrustados se vuelve inválido. El uso de metadatos al tiempo de ejecución en lugar de su uso en tiempo de compilación, permite la construcción de sistemas más extensibles con procedimientos más genéricos y estructuras más abstractas.

CLASES PARAMETRIZADAS

Las clases parametrizadas o genéricas, permiten que una plantilla parametrizada pueda ser escrita y aplicada a diversas clases que difieren solamente en los tipos de parámetros. Por ejemplo, una clase genérica llamada Lista puede manejar instancias tales como : Una lista de Puntos, o una lista de enteros. El método genérico "Agregar un Elemento" en la lista depende implícitamente del tipo de elemento. Las variables locales dentro del método son de tipo genérico y dependen de la instancia. Las clases parametrizadas están presentes en lenguajes como ADA e Eiffel. Las plantillas parametrizadas han sido propuestas en C++.

ASERCIONES Y RESTRICCIONES.

Generalmente existen algunas suposiciones críticas acerca del comportamiento de una clase, operación, o método, las cuales pueden expresarse en términos matemáticos. Estas suposiciones toman la forma de aserciones y deben ser verdaderas en puntos particulares durante la ejecución (por ejemplo, precondiciones o postcondiciones).

Las aserciones deben ser escritas dentro del programa de forma tal que puedan ser opcionalmente compiladas y checadas al momento de ejecución. Esta técnica puede ser utilizada en muchos lenguajes (por ejemplo : mediante la macro assert disponible en muchas implementaciones de C). Particularmente en el lenguaje Eiffel se encuentra bien soportada.

Las restricciones son más que condiciones que deben evaluarse. Pueden ser vistas como una forma de expresar en forma de declaración lo que de otra forma tendría que ser escrito como código procedural. El lenguaje puede entender las restricciones que se declaran y asegurar que éstas permanezcan verdaderas al momento de ejecución. El lenguaje Prolog posee estas capacidades.

PERSISTENCIA DE DATOS.

Si se requiere que los datos se conserven después de la ejecución de un programa, es necesario utilizar un almacén de datos permanente. Existen varias razonas para justificar la conservación de los datos :

Los datos conservados proporcionan el mecanismo más fácil de pasar datos de un programa a otro

La conservación de los datos permite que un mismo programa pueda continuar un proceso en una fecha posterior.

Los almacenes de datos son frecuentemente útiles para propósitos históricos.

Los distintos enfoques para proporcionar servicios de conservación de datos : archivos, dispositivos especiales, bases de datos, y el soporte intrínseco del lenguaje. Algunos de los recientes lenguajes orientados a objetos son especialmente convenientes para este uso.

PANORAMA DE LENGUAJES ORIENTADOS A OBJETOS.

Habiendo considerado diversas características de los lenguajes, examinemos como estas han sido combinadas en varios lenguajes orientados a objetos comercialmente disponibles: Smalltalk, C++, Eiffel, Clos, y lenguajes de programación orientada a objetos de bases de datos.

SMALLTALK.

Smalltalk fue el primer lenguaje popular orientado a objetos. Lo desarrolló Xerox Parac, y su éxito engendró muchos otros lenguajes orientados a objetos. Smalltalk no es solamente un lenguaje, sino un ambiente de desarrollo que incorpora algunas de las funciones de un sistema operativo. Para desarrollos un solo usuario, ofrece las mejores características tanto de lenguaje, como de ambiente. Smalltalk es deficiente en áreas en las cuales no se ha pretendido su uso, como proyectos de varias personas, y en su débil o no especificada capacidad para interactuar con Software externo o

dispositivos de Hardware. Smalltalk articula de una forma muy elegante las metas de extensibilidad y reutilización.

Todos los aspectos del lenguaje Smalltalk están disponibles a través de un interprete en línea y una utilería para el desplegado de información (browser) de clases. La sintaxis del lenguaje es sencilla. Las variables y los atributos no son tipificados. En Smalltalk todo es un objeto, incluso las clases. Las clases pueden ser agregadas, extendidas, probadas y depuradas interactivamente. El colector de basura libra al programador de la carga de administración de memoria.

¿Que proporciona Smalltalk al implementador? quizá la contribución más importante es su ambiente de desarrollo altamente interactivo, que evita los retrasos del ciclo de edición, compilación y ligado, lo cual es común en los lenguajes tradicionales basados en compiladores. El ambiente de Smalltalk permite un rápido desarrollo de programas. Otro de sus puntos fuertes es la biblioteca de clases, la cual fue diseñada para ser extendida y adaptada por medio de la agregación de subclases para satisfacer las necesidades de aplicación. Debido a que Smalltalk es un lenguaje no tipificado, los componentes de las bibliotecas se pueden combinar para crear rápidamente prototipos de la aplicación.

Otra importante contribución de Smalltalk es la arquitectura Modelo-Vista-Controlador (MVC) para diseñar interfases de usuario. La interfase del usuario se divide en el modelo de aplicación , un sinnúmero de diferentes vistas del modelo, y controladores que sincronizan los cambios al modelo y a sus vistas. MVC hace posible el concentrarse en lo esencial de la aplicación (modelo) y agregar la interfase de usuario (vistas y combinador) de manera independiente. La biblioteca de clases proporciona una versión estándar de cada uno de estos componentes, los cuales se pueden dividir en subclases y extenderse de forma incremental. Pueden existir muchos pares diferentes de vistas-controladores para cada modelo, y las vistas y controladores se pueden modificar extensamente sin afectar al modelo. No obstante, MVC es un sistema complejo y no resulta fácil de aprender.

Smalltalk es un lenguaje puro, orientado a objetos, con metadatos disponibles que son extendibles y modificables en tiempo de ejecución. La implementación del lenguaje como interprete, estrechamente integrado con

otras partes de su ambiente, brinda un soporte ideal para un rápido desarrollo incremental.

C++

C++ Es un lenguaje híbrido en el que algunas entidades son objetos y otras no lo son. C++ Es una extensión del lenguaje C, implementado no solamente para agregarle capacidades orientadas a objetos, sino también para reorientar algunas de las debilidades del lenguaje C. Muchas de las características agregadas están directamente hecha para la programación orientada a objetos, tales como la expansión de subrutinas, sobrecarga de funciones, y prototipos de funciones. Debido a su origen como una extensión del lenguaje C, se encuentra respaldado por muchas firmas. Su orientación como un lenguaje no propietario, y la disponibilidad de una gran variedad de compiladores, hacen parecer que se convertirá en el lenguaje orientado a objetos dominante en el mercado.

C++ Es un lenguaje fuertemente tipificado que fue desarrollado por Bjarne Stroustrup en los laboratorios de AT&T Bell. Originalmente fue implementado como un preprocesador que traduce C++ a lenguaje C estándar. Como preprocesador, C++ introdujo problemas para depuradores simbólicos, aunque ya se encuentran disponibles compiladores directos, así como también depuradores simbólicos que soportan objetos con herencia y cohesión dinámica. Los vendedores comerciales ofrecen implementaciones de C++ para una variedad de sistemas operativos.

A diferencia de otros lenguajes orientados a objetos, C++ no contiene una biblioteca de clases estándar como parte de su ambiente, aunque la versión estándar de AT&T incluye bibliotecas de Entrada/Salida, trabajo con corutinas, y aritmética compleja. Las bibliotecas de clases han sido implementadas por varios fabricantes. Desafortunadamente, debido a que C++ no proporciona parámetros para organización de bibliotecas, algunas bibliotecas pueden resultar incompatibles. El surgimiento de un consenso a favor de una fundación de bibliotecas de clases estándar sería una importante ventaja para C++.

C++ contiene facilidades para manejo de herencia y resolución de métodos en tiempo de ejecución, pero la estructura de datos de C++ no está automáticamente orientada hacia objetos. La resolución de métodos y la habilidad de introducir una operación en una subclase es posible solamente si la operación es declarada "virtual" en la super clase. De este modo, la necesidad de introducir un método debe anticiparse y escribirse en la definición de clase de origen. Desafortunadamente, quien escribe la clase tal vez no se espere que haya necesidad de definir subclases especializadas, o tal vez no conozca las operaciones que deberán ser redefinidas por una subclase. Esto significa que muchas veces la superclase debe ser modificada cuando una subclase es definida, lo cual presenta una seria restricción a la capacidad de reutilizar bibliotecas de clases a través de la creación de subclases, especialmente si el código fuente de la biblioteca no está disponible. (Por supuesto, todas las operaciones se podrían declarar "virtuales", a costa de una sobrecarga de memoria y llamadas a funciones).

La implementación de resoluciones de métodos en tiempo de ejecución es eficiente. Para cada clase se inicializa una estructura predefinido con apuntadores a cada uno de los métodos disponible para la clase. Cada objeto contiene un apuntador a la estructura del método de su clase. Al tiempo de ejecución, una operación virtual se resuelve cargando la estructura de método del objeto y seleccionando uno de los miembros para encontrar la dirección del método. C++ no soporta objetos descriptores en tiempo de ejecución que no sean apuntadores a estructuras de métodos. A partir de la versión 2.0, C++, soporta herencia múltiple.

C++ incluye facilidades de especificación de acceso a los atributos y operaciones de una clase. El acceso puede ser permitido a través de métodos de cualquier clase (públicos), restringido a métodos de subclases de una clase (protegido), o restringido a métodos directos de clase (privados).

Al igual que con C, la sintaxis de declaración en C++ es complicada y su gramática difícil de describir. C++ soporta operadores de sobrecarga: diversos métodos que comparten el mismo nombre pero cuyos argumentos pueden variar en cantidad o en tipo.

C++ soporta diversas estrategias de asignación de memoria para los objetos (asignados estáticamente por el compilador, basados en apilamiento, y

asignados en base al área de memoria disponible en tiempo de ejecución). El programador debe evitar mezclar objetos con diferentes tipos de manejo de memoria debido a que los conflictos de las referencias pueden causar fallas al momento de uso. Cada clase puede tener diversas funciones "constructor" y "converter", las cuales inicializan los objetos nuevos y que convierten los tipos de dato en los procesos de asignación y paso de argumentos.

En resumen, el C++ es un lenguaje complejo, maleable, que se caracteriza por su énfasis en la detección temprana de errores, diversas opciones de implementación, y eficiencia de ejecución.

EIFFEL

Eiffel es un lenguaje orientado a objetos fuertemente tipificado, escrito por Bertrand Meyer. Los programas consisten de conjuntos de declaraciones de clases que incluyen métodos. Soporta herencia múltiple, clases parametrizadas (genéricas), administración de memoria, y aserciones. Proporciona una biblioteca modesta de clases, que incluye listas, arboles, pilas, colas, archivos, cadenas, tablas hash, y árboles binarios. Para brindar portabilidad, el compilador traduce los programas fuente a lenguaje C.. Eiffel posee facilidades de ingeniería de Software para manejar encapsulamiento y control de acceso. En términos de sus capacidades técnicas, Eiffel es el mejor lenguaje orientado a objetos comercial, aunque esto también es argumentable.

El aspecto central del Eiffel es su declaración de clases, la cual permite listar atributos y operaciones. Eiffel proporciona un acceso uniforme a los atributos y operaciones abstrayéndolas en un solo concepto al que se denomina como "característica". Una declaración de clases de Eiffel puede incluir una lista de características exportadas, una lista de clases antecesoras, y una lista de declaraciones de características. Eiffel no trata las clases o las asociaciones como objetos de una primera clase.

Eiffel soporta el manejo de memoria a través de una corutina que detecta los objetos que ya no son referenciados y libera la memoria que se les había asignado. En tiempo de ejecución, Eiffel ejecuta la corutina cuando el espacio de memoria disponible resulta insuficiente. Se proporcionan varios mecanismos para controlar el manejo de memoria. La ejecución automática

de la corutina puede ser suprimida a través de un interruptor del compilador. Para sistemas operativos que no soportan memoria virtual, existe un interruptor del compilador de Eiffel que proporciona paginación automática de memoria.

CLOS

Clos (Common Lisp Object System) es una extensión orientada a objetos del lenguaje Common Lisp, objetos. Es resultado del trabajo de un grupo encargado de la estandarización de Lisp que trabajaba en el estándar de ANSI X3J13. Este grupo estudió las muchas extensiones orientadas a objetos de Lisp, pero en lugar de seleccionar uno de ellos como norma, el grupo decidió formular un nuevo lenguaje basado en las características de lenguajes que habían probado tener éxito. Debido a que Clos se puede implementar en Common Lisp, esto permite la experimentación de nuevos conceptos, mientras se establece un estándar común para Lisp.

Aunque originalmente se implementó como una modalidad híbrida, Clos está tan bien integrado con las características tradicionales del Common Lisp, que contiene la mayoría de las ventajas de un lenguaje "puro" orientado hacia objetos. Esto se debe a que cada dato de objeto, incluyendo los átomos y listas de Lisp, es miembro de una clase.

El ambiente de programación de Common Lisp y Clos consiste en un interprete, el cual permite que el código se pueda compilar en una forma tal que puede ejecutarse con mayor eficiencia. Las facilidades de Depuración bajo Common Lisp dependen de las implementaciones específicas, pero generalmente son excelentes.

Clos proporciona poderosas y flexibles capacidades de manejo de herencia. Soporta la herencia múltiple, y posee reglas para resolver la ambigüedad que resulte de las características heredadas con el mismo nombre.

Las operaciones polimórficas que requieren un dinámico método de resolución pueden implementarse como métodos genéricos. Todos los argumentos a un método genérico son explícitos; No existe una variable especial como "self" en Smalltalk. A diferencia de la mayoría de lenguajes

orientados hacia objetos, Clos utiliza argumentos múltiples para resolver métodos, permitiendo la implementación directa de operaciones que exhiben "polimorfismo múltiple" en los que un método específico para una operación depende de más de uno de sus argumentos (por ejemplo, operaciones aritméticas binarias).

Clos proporciona una rica colección de metadatos a que pueden ser accesados y actualizados en tiempo de ejecución. Las nuevas clases pueden ser definidas, y los métodos se pueden agregar a las clases en forma dinámica. Estas características son una parte estándar del lenguaje. A diferencia de la mayoría de lenguajes, los que están basados en Lisp permiten la construcción de nuevos procedimientos durante el tiempo ejecución.

Al igual que Common Lisp, Clos es un lenguaje débilmente tipificado. Se proporcionan tipos nativos, y las clases se comportan como tipos, pero no hay ningún requerimiento para declarar el tipo de una variable, y no existe verificación que asegurarse que el uso de un objeto sea consistente con su declaración. Las declaraciones opcionales pueden ser usadas por el compilador para optimizar el acceso, pero esta optimización depende de la implementación.

El concepto de encapsulación no es forzoso en CLOS. El programador es responsable de definir y documentar la interfase pública de cada clase, ya que no existe nada que prevenga que el código de una clase sea accesado directamente desde otras clases

COMPARACIÓN DE LOS LENGUAJES ORIENTADOS A OBJETOS.

	C++ 2.0	*Smalltalk 80*	*Clos*	*Eiffel*	*Objective C*
Integración de Clases con tipos primitivos	Híbrido	Puro	Integrado	Integrado	Híbrido
Fuerte Verificación de tipos	SI	NO	NO	SI	SI
Capacidad de restringir el acceso a atributos: Control de acceso a clientes					
Control de acceso desde los clientes	SI	SI	NO	SI	SI
Control de acceso desde las subclases	SI	NO	NO	SI	NO
Biblioteca de clases standard	NO	SI	NO	SI	SI
Clases parametrizadas	A Futuro	No aplicable	No aplicable	SI	NO
Herencia múltiple	SI	NO	SI	SI	NO
Alcance de los nombres de clases (paquetes)	NO	NO	SI	NO	NO
Modelo de Mensajes	SI	NO	SI	SI	NO
A un solo objeto destino	SI	SI	NO	SI	SI
Unión dinámica de argumentos múltiples	NO	NO	SI	NO	NO
Combinación de métodos					
SUPER concepto	NO	SI	SI	SI	SI
Antes y después de los métodos	NO	NO	SI	NO	NO

Aserciones y Restricciones	NO	NO	NO	SI	NO
Metadatos en tiempo de ejecución	NO	SI	SI	NO	SI
Recolección de basura	NO	SI	SI	SI	NO
Eficiencia: Unión Estática si es posible	SI	NO	NO	SI	SI

Aplicación de la metodología

Planteamiento del problema

En la actualidad, la mayoría de las empresas han atendido en menor o mayor grado, la necesidad de contar con sistemas de información confiables y eficientes. Sin embargo, consideramos que si bien este tipo de sistemas cumplen con el objetivo de proporcionar información confiable y oportuna, adolecen todavía de la capacidad de integración y sobre todo, de la capacidad de facilitar el análisis de la información que generan, y con ello fortalecer la toma de decisiones.

La labor del tomador de decisiones, generalmente está respaldada por una mezcla de recursos humanos y técnicos. Una eficiente coordinación de estos recursos, se refleja necesariamente en la calidad de las decisiones. Sin embargo, dicha coordinación se torna entonces, en una tarea crítica y costosa en la mayoría de los casos.

El proceso de toma de decisiones se origina en los sistemas de producción de una empresa, los cuales están relacionados con el giro de la misma. Posteriormente la información generada a partir del registro de operaciones en los sistemas de producción, puede llegar directamente al tomador de decisiones, o ser resumida y organizada de tal forma que su análisis sea más fácil de entender, y se presente incluso, en forma de gráficas. Esta labor es realizada por personal dedicado a dicha actividad y procesada en herramientas que generalmente no tienen relación directa con los sistemas de producción.

Otro obstáculo que se presenta en la toma de decisiones, es la dificultad de contar con un panorama general de la información de interés para la empresa. En ocasiones una decisión no tiene buenos resultados por haber descuidado factores de los cuales no se contaba con información suficiente.

Por último, el proceso de análisis y evaluación de la información por parte del tomador de decisiones, continua siendo un proceso basado en su experiencia

y capacidad. De hecho, el esfuerzo para llevarlo a cabo sigue siendo considerable, y se cuentan con pocas herramientas de apoyo.

Nuestro planteamiento del problema consiste en desarrollar un sistema de información gráfico de apoyo a la toma de decisiones que cumpla con las siguientes características y ventajas :

- Que facilite el análisis de información a través de una interfase gráfica que sea amigable y de uso sencillo para el tomador de decisiones.

- Que permita conocer el status global de la empresa, y a la vez, que cuente con la posibilidad de mostrar niveles más detallados del comportamiento de la información.

- Que posea la flexibilidad para adaptarse a cualquier tamaño de empresa

- Que mediante su uso, reduzca costos de personal dedicado a la síntesis, organización y presentación de información directiva.

- Que beneficie al tomador de decisiones, permitiéndole contar con mayor información en menor tiempo.

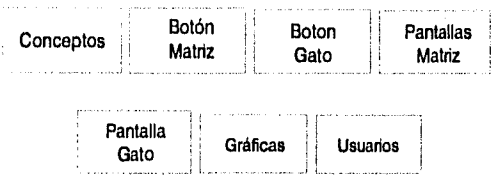
- Que la información que presente al tomador de decisiones, se encuentra ya evaluada de acuerdo a los criterios propios de la empresa.

- Que favorezca la estandarización de información corporativa.

A partir de lo anterior se llego a la decisión de realizar un sistema que tendría pantallas al estilo de los denominados tableros de control en los que por medio de botones de colores se representa la situación de las diferentes áreas que conforman una empresa, esta situación es resultado de una evaluación entre dos "conceptos" de información. Por lo que el sistema deberá tener pantallas que muestren estos cruces de conceptos y otros que sirvan para mostrar estados y para realizar la navegación entre niveles de detalle de la información.

Identificación de las clases de objetos:

Las clases identificadas para la aplicación STD son las siguientes:



Diccionario de datos

Conceptos:

Un concepto es cualquier tema o elemento de información que interesa a una organización y que tiene un comportamiento variable en el tiempo; como ejemplos de conceptos tenemos los siguientes: las ventas del mes, el número de empleados de la empresa, el costo mensual de producción, el nivel de inventarios, etc.

Ahora bien, un concepto analizado de manera aislada no proporciona información valiosa a menos que se relacione (compare) con el valor de otro concepto. De esta manera tenemos que las ventas de una empresa no nos dicen nada por sí solas, pero si las relacionamos con el presupuesto de ventas o el costo de ventas o bien contra las ventas de un periodo anterior, podremos observar la tendencia de las ventas con respecto a algún otro concepto y de tal relación podemos deducir entonces sí es que las ventas se están desarrollando positivamente, en un rango tolerable, o bien en forma negativa.

Botón matriz:

Es un control gráfico en cuya definición se relacionan dos conceptos de acuerdo a un criterio de evaluación. Este criterio es el que determina el color que tomará el botón; los posibles colores son: Verde, amarillo, rojo o gris.

Verde	El cruce de los conceptos representado por ese botón se encuentra en condiciones sobresalientes para la empresa
Amarillo	La relación entre los conceptos del renglón y la columna, que el botón representa, se encuentra en condiciones tolerables.
Rojo	El cruce de conceptos representado en el botón, se encuentra en una situación crítica
Gris	El cruce de datos programado en el botón no pudo ser determinado debido a que no existían los datos necesarios en el sistema (falta de captura ó definición errónea).

Botón Gato:

Es un control gráfico contenido en las pantallas "Gato" que muestra el estado de la siguiente pantalla y además sirve para mostrar esta.

Pantallas Matriz:

La pantalla matriz es aquella en la que se presentan los cruces entre los conceptos de información definidos en el sistema. Tales cruces son representados por un botón matriz que puede adquirir diferentes colores dependiendo de la calificación de dicha relación. Los botones se distribuyen en la pantalla de acuerdo a una matriz de 10 por 10 lo que nos da la posibilidad de tener por pantalla hasta 100 relaciones calificadas.

Pantallas gato

Las pantallas gato sirven para agrupar botones gato, se pueden mostrar hasta 36 botones gato por cada pantalla, agrupados en nueve grupos de 4 botones cada una. Las pantallas gato permiten la navegación entre los diferentes niveles de detalle de información contenidos en el sistema.

Gráfica

Una gráfica es la representación pictórica de los últimos 13 datos correspondientes a los conceptos que se interceptaban en el botón matriz que llamó a la gráfica.

Usuarios

Denominaremos a los usuarios como toda aquella persona autorizada para el uso del sistema STD.

MODELO DE OBJETOS INICIAL

El sistema STD se ha dividido en dos grandes módulos:

Uno llamado de administración y otro de consulta. Esta división se definió considerando las diferentes operaciones realizadas por los también diferentes usuarios del sistema. Algunos usuarios solo consultan la información y no se preocupan por actualizar los datos o por definir las consultas; mientras que otros estarán encargados de realizar éstas últimas operaciones para ellos.

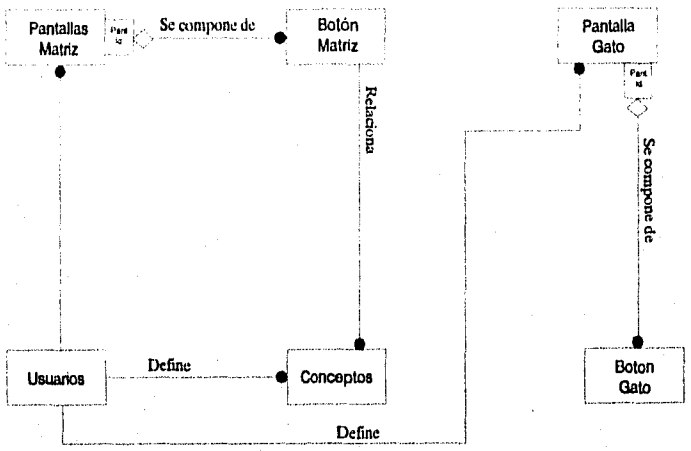
Se han identificado las mismas clases para ambos módulos, aunque existen diferencias básicas como las siguientes:

El módulo de administración funge como generador de la información que será accesada por el módulo de consulta. Éste módulo actualiza información mientras que el módulo de consulta solo los lee.

La forma en que se relacionan las clases es distinta en ambos módulos por lo que el análisis del sistema resulta más sencillo separando los módulos.

No resulta lógico incluir las vistas a las clases desde el módulo de consulta cuando estas son analizadas desde el punto de vista de administración.

Diagrama de objetos inicial (Administración)

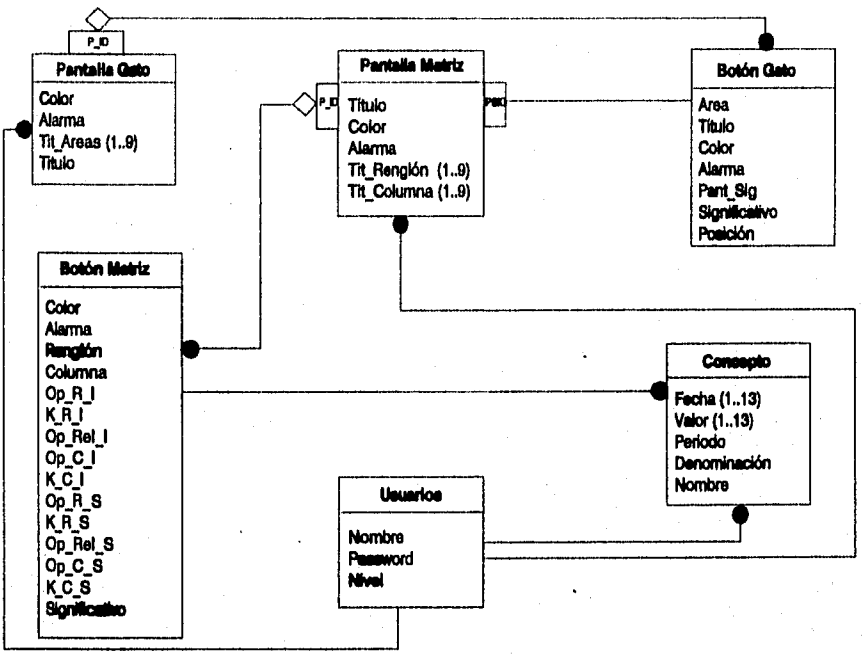


Lista de asociaciones identificadas

El usuario define los conceptos de información.
El usuario define las pantallas matriz.
El usuario define pantallas gato.
Las pantallas gato están compuestas de botones gato.
Las pantallas matriz están compuestas de botones matriz.
Los botones matriz relacionan conceptos de información.

IDENTIFICACIÓN DE ATRIBUTOS

MODELO DE OBJETOS CON ATRIBUTOS (adm)

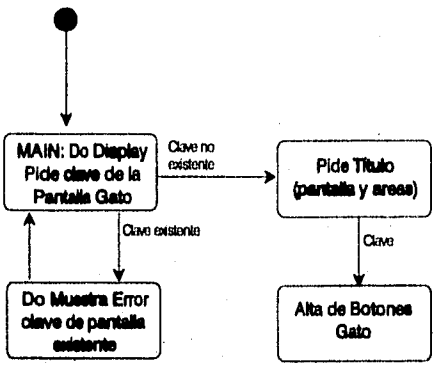


Refinamiento con herencia

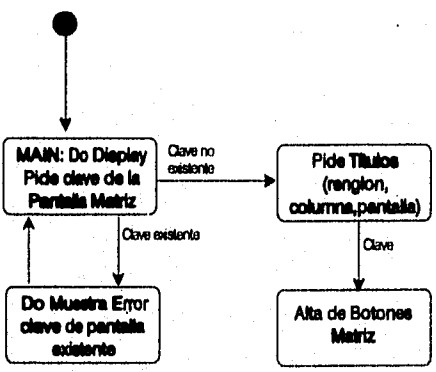
Debido a que el comportamiento esperado de los objetos del sistema es muy diferente entre ellos, se optó por no utilizar herencia, basándonos no en los atributos, sino en el comportamiento.

Pareciera conveniente agrupar las clases pantallas gato y pantallas matriz en una superclase llamada simplemente pantallas. Sin embargo si examinamos los atributos de cada clase, encontramos que el único atributo común es el de identificador, con lo cual no se aprovechan las ventajas de la herencia. Esta misma situación se repite en el caso de los botones gato y botones matriz.

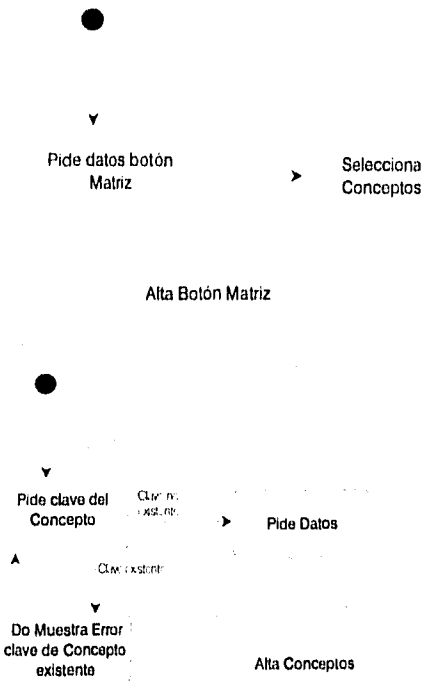
MODELO DINÁMICO

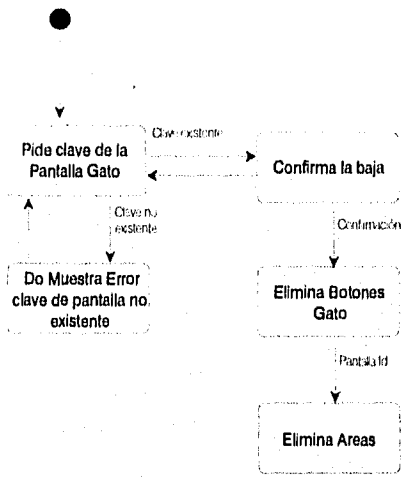


Alta Pantallas Gato

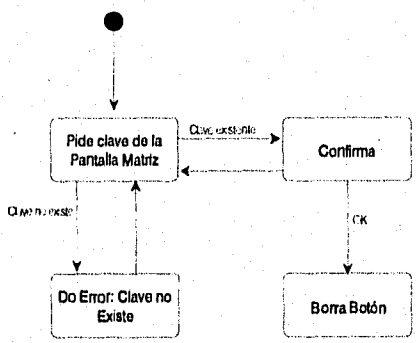


Alta Pantallas Matriz

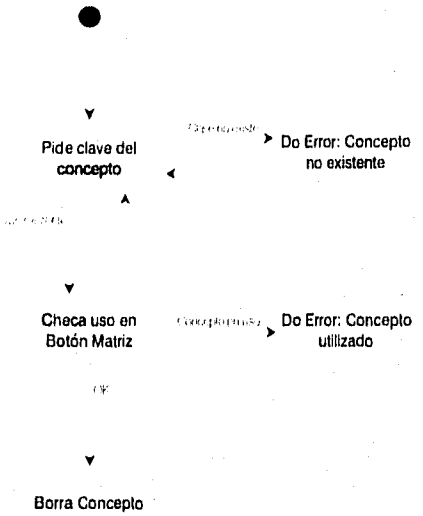

Pide datos botón Matriz
Selecciona Conceptos
Alta Botón Matriz
Pide clave del Concepto
Pide Datos
Do Muestra Error clave de Concepto existente
Alta Conceptos



Baja pantallas Gato

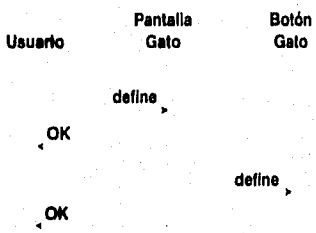


Baja pantallas Matriz

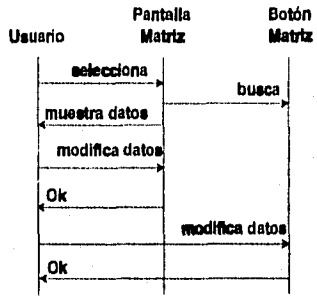


Borrado de Conceptos

Rastreo de Eventos (adm) alta a pantallas gato

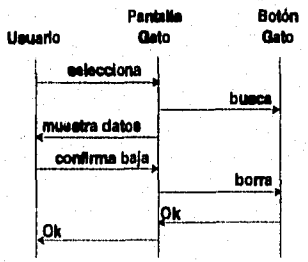


Rastreo de Eventos (adm) cambio a pantallas matriz

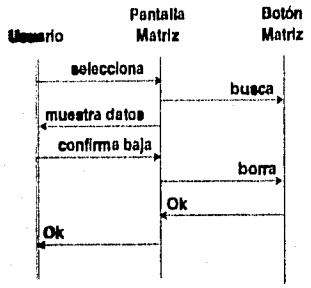


Rastreo de Eventos (adm) cambio a conceptos

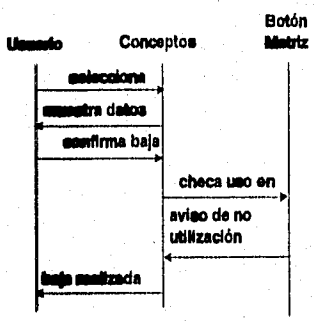
Rastreo de Eventos (adm) borra pantallas gato



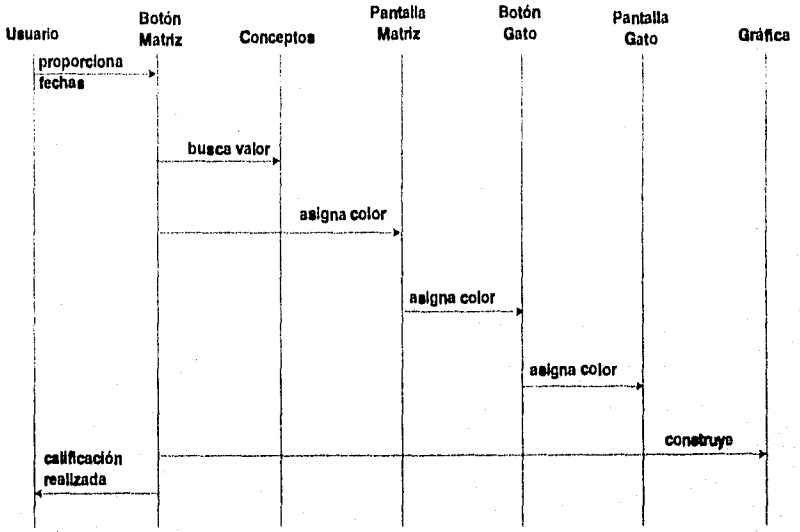
Rastreo de Eventos (adm) borra pantallas matriz



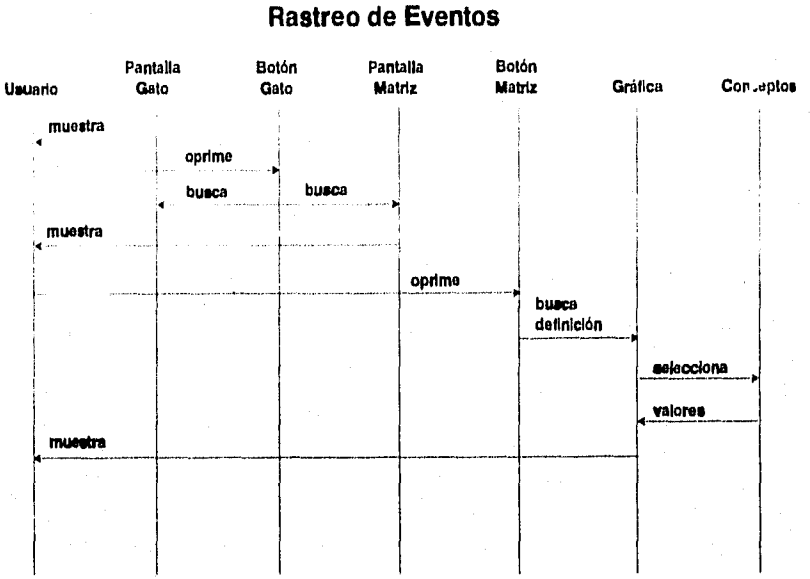
Rastreo de Eventos (adm) borra conceptos



Rastreo de Eventos (adm) calificación



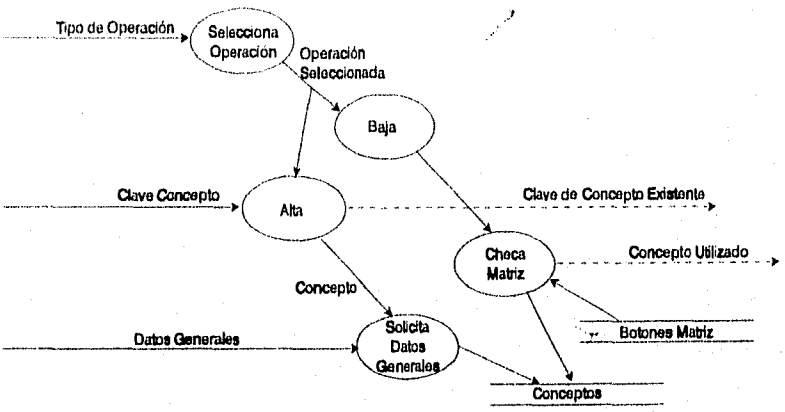
CONSULTA



El sistema muestra la primer pantalla gato.
El usuario oprime un botón de la pantalla gato.
El botón busca la pantalla siguiente que puede ser gato o matriz y la muestra al usuario.
Si la pantalla mostrada es una matriz el usuario puede oprimir algún botón matriz
el botón matriz busca la definición de la gráfica correspondiente.
La gráfica selecciona los conceptos relacionados en ella.
Los conceptos proveen los valores a graficar.
La gráfica es mostrada al usuario.

MODELO FUNCIONAL

DFD de Conceptos



CONCEPTOS

Selecciona Operación

Determinar el Botón accionado por el usuario
Si el Botón accionado es Agregar
    Ejecuta Función Alta
Si el Botón accionado es Borrar
    Ejecuta Función Baja
Si el Botón accionado es Buscar
    Solicita Clave del Concepto
    Localiza la clave capturada en el archivo de Conceptos
    Si no existe la clave del Concepto
        Despliega mensaje de "Registro no Encontrado"
    Si existe la clave
        Despliega los datos del concepto

Alta

Solicitar la Clave del Concepto
Localizar la clave del concepto en la tabla de Conceptos
Si la clave del concepto existe
    Desplegar mensaje de "Concepto ya registrado"
Si la clave del concepto no existe
    Ejecuta función Solicita Datos Generales

Solicita Datos Generales

Solicita Descripción del Concepto
Solicita la Periodicidad del Concepto
Solicita la Denominación del Concepto
Actualiza la información en el archivo de Conceptos

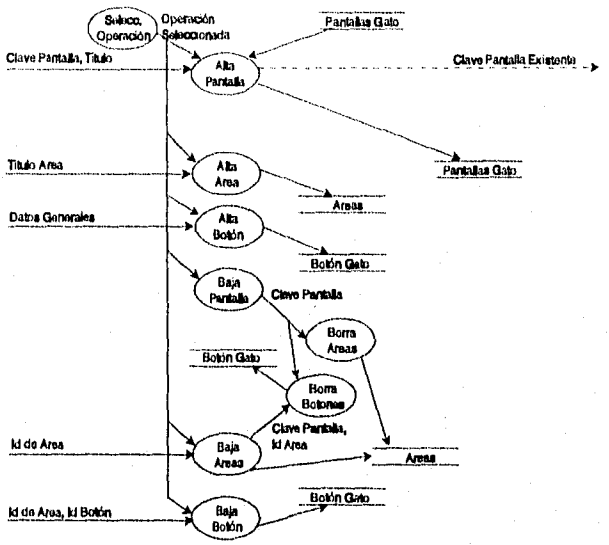
Baja

Solicita Confirmación de la Baja
Si la Baja es confirmada
    Ejecuta la función Checa Matriz

Checa Matriz

Localiza concepto en el archivo de Botones Matriz
Si existe el Concepto
    Despliega mensaje de Error "Concepto Utilizado, No se puede Borrar"
Si no existe el Concepto
    Borra el concepto del archivo de Conceptos

DFD de Pantallas Gato



PANTALLAS GATO

Selecciona Operación

Determinar el botón accionado por el usuario
Si el botón accionado es agregar
    Ejecuta la función alta pantalla gato
Si el botón accionado es borrar
    Ejecuta la función baja pantalla gato
Si el botón accionado es buscar
    Solicita clave de la pantalla gato
    Localiza la clave capturada en la tabla de pantallas gato
    Si no existe clave
        Despliega mensaje de "Registro No Encontrado"
    Si existe la clave
        Despliega los datos de pantalla gato

Determina las áreas de la pantalla gato
Localiza el identificador del área en la tabla de áreas
Muestra títulos de áreas en pantalla gato
Determina los botones gato de cada área y pantalla gato
Localiza los botones gato en la tabla de botones gato
Muestra botones gato por área en la pantalla gato

Alta Pantalla Gato

Solicita la clave de la pantalla gato
Determina la clave en la tabla de pantallas gato
Si ya existe la clave
    Muestra mensaje "Pantalla Ya Existe"
Si no existe la clave
    Solicita título de la pantalla gato
    Actualiza la información en la tabla de pantallas gato

Alta Áreas

Solicita el título del área
Actualiza la información en la tabla de áreas

Alta Botón Gato

Solicita título del botón gato
Pregunta si el botón gato es significativo
Pregunta si la pantalla siguiente es pantalla matriz
Solicita identificador de pantalla siguiente
Actualiza la información en la tabla de botones gato

Baja Pantalla Gato

Verifica "Confirme Baja"
Si la respuesta es afirmativa
    Ejecuta borra áreas
    Borra la pantalla gato de la tabla pantallas gato

Borra Áreas

Determina las áreas de la pantalla gato

Localiza las áreas en la tabla de áreas
Ejecuta borra botón gato
Borra las áreas de la tabla áreas

Borra Botón Gato

Determina los botones gato del área
Localiza los botones gato en la tabla botón gato
Borra los botones gato de la tabla botón gato

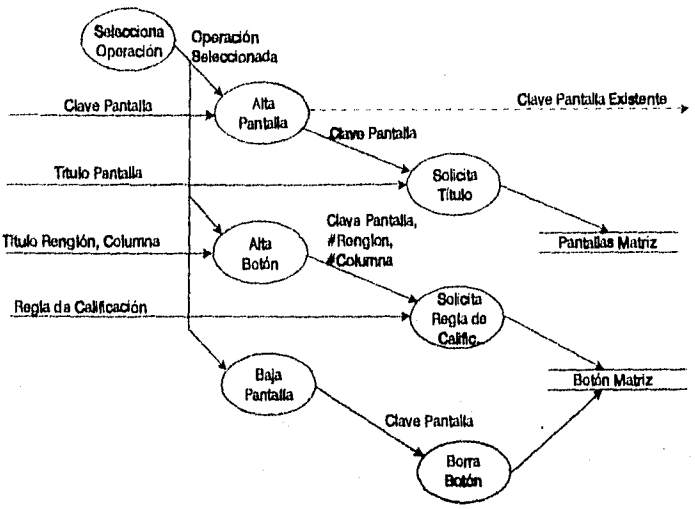
Baja Áreas

Solicita título área
Localiza área en tabla áreas
Verifica "Confirme Baja"
Si la respuesta es afirmativa
    Ejecuta borra botón gato
    Borra el área de la tabla áreas

Baja Botón Gato

Solicita identificador del botón gato
Localiza el botón gato en la tabla de botones gato
Verifica "Confirme Baja"
Si la respuesta es afirmativa
    Borra el botón gato de la tabla botones gato

DFD de Pantallas Matriz



PANTALLAS MATRIZ

Selecciona Operación

Determina el botón accionado por el usuario
Si el botón accionado es agregar
    Ejecuta la función alta pantalla matriz
Si el botón accionado es borrar
    Ejecuta la función baja pantalla matriz
Si el botón accionado es buscar
    Solicita la clave de la pantalla matriz
    Localiza la clave capturada en la tabla de pantallas matriz
    Si no existe la clave
        Despliega mensaje "Registro No Encontrado"
    Si existe la clave
        Despliega los datos de la pantalla matriz
        Localiza los botones matriz en la tabla botones matriz
        Muestra los botones matriz en la pantalla matriz

Alta Pantalla Matriz

Solicita la clave de la pantalla matriz
Determina la clave en la tabla de pantallas matriz
Si ya existe la clave
    Muestra mensaje "Pantalla Ya Existe"
Si no existe la clave
    Solicita título de la pantalla matriz
    Solicita título para columnas
    Solicita título para renglones
    Actualiza la información en la tabla de pantallas matriz

Alta Botón Matriz

Ejecuta solicita regla de calificación

Solicita Regla de Calificación

Solicita concepto renglón del límite inferior
Localiza el identificador de concepto en la tabla conceptos
Si no existe la clave
    Despliega mensaje "Concepto No encontrado"
Si existe la clave
    Solicita el operador renglón del límite inferior
    Solicita la constante renglón del límite inferior
    Solicita el operador de relación del límite inferior
    Solicita concepto columna del límite inferior
    Localiza el identificador de concepto en la tabla conceptos
    Si no existe la clave
        Despliega mensaje "Concepto No encontrado"
    Si existe la clave
        Solicita el operador columna del límite inferior
        Solicita la constante columna del límite inferior

Solicita concepto renglón del límite superior
Localiza el identificador de concepto en la tabla conceptos
Si no existe la clave

Despliega mensaje "Concepto No encontrado"
Si existe la clave
Solicita el operador renglón del límite superior
Solicita la constante renglón del límite superior
Solicita el operador de relación del límite superior
Solicita concepto columna del límite superior
Localiza el identificador de concepto en la tabla conceptos
Si no existe la clave
Despliega mensaje "Concepto No encontrado"
Si existe la clave
Solicita el operador columna del límite superior
Solicita la constante columna del límite superior
Pregunta si el botón matriz refleja su calificación en pantalla anterior
Actualiza la información en la tabla de botones matriz

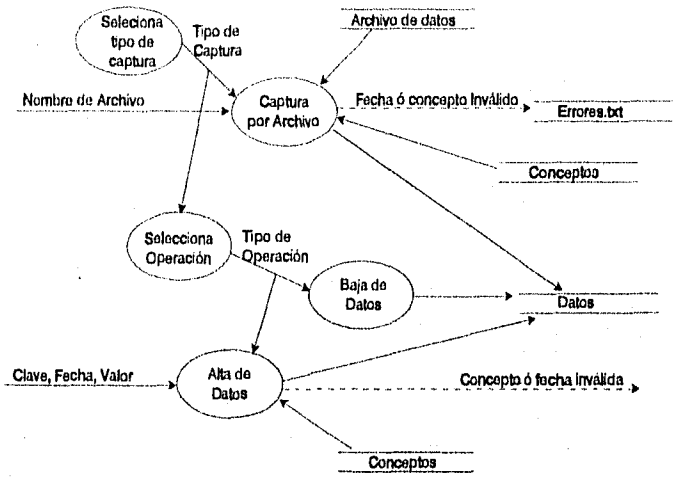
Baja Pantalla Matriz

Verifica "Confirme Baja"
Si la respuesta es afirmativa
Ejecuta borra botón matriz
Borra la clave de la tabla pantallas matriz

Borra Botón Matriz

Determina los botones matriz de la pantalla matriz
Localiza los botones matriz en la tabla botones matriz
Borra los botones matriz de la tabla botones matriz

DFD de Captura de Datos



CAPTURA

Selecciona Tipo de Captura

Determinar el Botón accionado por el usuario.
Si el Botón accionado es Captura por Archivo
    Ejecuta la función Captura por Archivo
Si el Botón accionado es Captura Manual
    Ejecuta la función Captura Manual

Captura por Archivo

Solicita el Nombre del Archivo
Valida que el archivo se encuentre
Si el archivo no se encuentra
    Despliega mensaje de error "Archivo no encontrado"

```
Si el Archivo se encuentra
    Ejecuta:
    Mientras no ocurra fin de archivo
        Lee concepto del Archivo de Datos
        Valida que el concepto existe en el archivo de Conceptos
        Si el concepto no existe
            Graba error en el archivo de Errores
            Continua con el Sig. registro del archivo de datos
        Si el concepto Existe
            Valida que la periodicidad del concepto sea correcta
            Si la periodicidad no es correcta
                Graba error en el archivo de errores
                Continua con el siguiente registro del arch.de
                datos
            Si la periodicidad es correcta
                Actualiza el Archivo de Datos
                Continua con el siguiente registro del arch.de
                datos
    Fin
```

Captura Manual

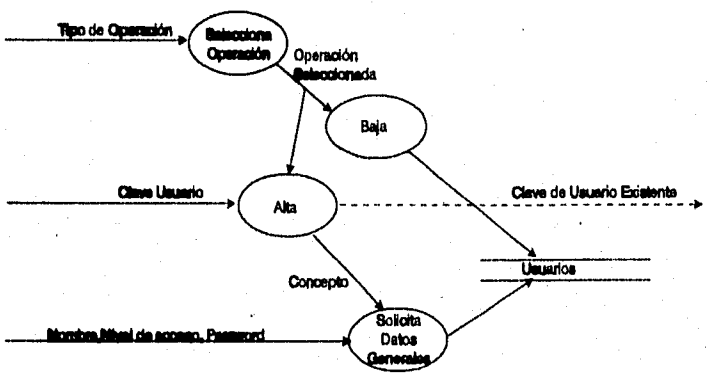
```
Solicita clave de concepto
Valida que el concepto exista en el archivo de conceptos
Si el concepto no existe
    Despliega mensaje de error "Concepto no registrado"
Si el concepto existe
    Solicita fecha del concepto
    Valida que la fecha capturada corresponda con la periodicidad
    del concepto
    Si le periodicidad es invalida
        Despliega mensaje de error "Fecha inválida"
        Solicita nuevamente la fecha del concepto
    Si la periodicidad es válida
```

Solicita Importe del Concepto
Actualiza archivo de Datos

Baja de Datos

Solicita la Clave del Concepto
Solicita la Fecha del Concepto
Solicita la Confirmación de la Baja
Si se Confirma la Baja
    Borra el registro del Archivo de Datos.

DFD de Usuarios



Selecciona Operación

Determina el botón accionado por el usuario
Si el botón accionado es alta
    Ejecuta la función alta usuario
Si el botón accionado es borrar
    Ejecuta la función baja usuario

Alta usuarios

Solicita la clave del usuario
Determina la clave en la tabla de usuarios
Si ya existe la clave
    Muestra mensaje "usuario Ya Existe"
Si no existe la clave
    Solicita nombre del usuario
    Solicita password
    Solicita nivel de acceso

Baja usuarios

Verifica "Confirme Baja"
Si la respuesta es afirmativa
    Borra la clave de la tabla de usuarios

OBTENCIÓN DE OPERACIONES

Las operaciones identificadas para cada una de las clases son las siguientes :

Pantallas Gato

Alta
Baja
Cambio

Pantallas Matriz

Alta
Baja
Cambio

Botones Matriz

Alta
Baja
Cambio

Botones Gato

Alta
Baja
Cambio

Conceptos

Alta
Baja
Cambio
Captura de Datos

Usuarios

Alta
Baja
Cambio

Gráficas

Actualización

La operación denominada 'Calificación' se separa de las anteriores, dado que por su complejidad, no está ligada directamente a una clase de objetos en particular.

DISEÑO DE ALGORITMOS PARA CADA OPERACIÓN

A continuación se describen los algoritmos para cada una de las operaciones identificadas.

Alta de Pantallas Gato

Solicita la clave de la pantalla gato
Valida la existencia de la clave en el archivo de pantallas gato
Si ya existe la clave
    Muestra mensaje "Pantalla Ya Existe"
Si no existe la clave
    Solicita título de la pantalla gato
    Solicita los títulos de las áreas de la pantalla
    Para cada una de las áreas capturadas almacena en el archivo de áreas los datos de : no. de pantalla, número de área y título de área
    Ejecuta la operación "Alta de Botones Gato"
    Almacena en el archivo de pantallas gato, los datos de : número de pantalla y título de pantalla

Bajas de Pantallas Gato

Solicita No. De pantalla gato a Eliminar
Verifica existencia del no. de pantalla en el archivo de pantallas gato
Si no existe
    Muestra mensaje "Pantalla no Existe"
Si existe
    Muestra los datos de : título de pantalla, títulos de áreas y botones gato pertenecientes a dicha pantalla.
    Solicita confirmación de Baja
    Confirmación = "SI" ?
        Borra del archivo de botones gato aquellos cuyo atributo de no. de pantalla sea igual al no. de pantalla indicado
        Borra del archivo de áreas aquellos cuyo atributo de no. de pantalla sea igual al no. de pantalla indicado
        Borra del archivo de pantallas gato el número de pantalla indicado
        Limpia la pantalla
    Confirmación = "NO"

Cancela la operación

Cambios de Pantallas Gato.

Solicita No. de pantalla gato a Modificar
Verifica existencia del no. de pantalla en el archivo de pantallas gato
Si no existe
Muestra mensaje "Pantalla no Existe"
Si existe
Muestra los datos de : título de pantalla, títulos de áreas y botones gato pertenecientes a dicha pantalla.
Solicita los nuevos datos, excepto el campo de número de pantalla.
Actualiza el archivo de áreas con los nuevos datos
Actualiza el archivo de botones gato con los nuevos datos
Actualiza el archivo de pantallas gato con los nuevos datos.

Alta de Pantallas Matriz

Solicita No. De pantalla matriz
Valida la existencia de la clave en el archivo de pantallas matriz
Si ya existe la clave
Muestra mensaje "Pantalla Ya Existe"
Si no existe la clave
Solicita título de la pantalla matriz
Solicita los títulos de los renglones y columnas de la pantalla
Para cada uno de los renglones capturados almacena en el archivo de renglón matriz los datos de : no. de pantalla, número de renglón y título de renglón
Para cada una de las columnas capturadas almacena en el archivo de columna matriz los datos de : no. de pantalla, número de columna y título de columna
Ejecuta la operación "Alta de Botones Matriz"
Almacena en el archivo de pantallas matriz, los datos de : número de pantalla y título de pantalla

Bajas de Pantallas Matriz

Solicita No. De pantalla matriz a Eliminar
Verifica existencia del no. de pantalla en el archivo de pantallas matriz
Si no existe
Muestra mensaje "Pantalla no Existe"
Si existe
Muestra los datos de : título de pantalla, títulos de renglón y columna, y los botones matriz pertenecientes a dicha pantalla.
Solicita confirmación de Baja
Confirmación = "SI" ?
Borra del archivo de botones matriz aquellos cuyo atributo de no. de pantalla sea igual al no. de pantalla indicado
Borra del archivo de renglón matriz aquellos cuyo atributo de no. de pantalla sea igual al no. de pantalla indicado
Borra del archivo de columna matriz aquellas cuyo atributo de no. de pantalla sea igual al no. de pantalla indicado
Borra del archivo de pantallas matriz el número de pantalla indicado
Limpia la pantalla
Confirmación = "NO"
Cancela la operación

Cambios de Pantallas Matriz.

Solicita No. de pantalla matriz a Modificar
Verifica existencia del no. de pantalla en el archivo de pantallas matriz
Si no existe
Muestra mensaje "Pantalla no Existe"
Si existe
Muestra los datos de : título de pantalla, títulos de renglón y columna, y botones matriz pertenecientes a dicha pantalla.
Solicita los nuevos datos, excepto el campo de número de pantalla.
Actualiza el archivo de renglón matriz con los nuevos datos

Actualiza el archivo de columna matriz con los nuevos datos
Actualiza el archivo de botones matriz con los nuevos datos
Actualiza el archivo de pantallas matriz con los nuevos datos.

Alta y cambio de Botones Gato

Solicita para cada uno de los botones elegidos por el usuario, los datos de:
Título del botón.
N° de pantalla siguiente
Tipo de la pantalla siguiente (Matriz o Gato)
La indicación de si el botón es significativo (Si/No)
Actualiza para cada uno de los botones seleccionados por el usuario, los datos en el archivo de botones Gato.

Baja de Botones Gato

Para cada uno de los botones elegidos por el usuario
Pide confirmación de la baja
si se confirma la baja
    Elimina del archivo de botones Gato los botones seleccionados
si la baja no es confirmada
    cancela la operación

Alta y cambio de Botones Matriz

Solicita para cada uno de los botones elegidos por el usuario, los datos de:
Las claves de los dos conceptos que se cruzarán en el botón Matriz
Las reglas de evaluación para los límites inferior y superior de la relación
La indicación de si el botón es significativo (Si/No)
Actualiza para cada uno de los botones seleccionados por el usuario, los datos en el archivo de botones Matriz.

Baja de Botones Matriz

Para cada uno de los botones elegidos por el usuario
Pide confirmación de la baja
si se confirma la baja
    Elimina del archivo de botones Matriz los botones seleccionados
si la baja no es confirmada
    cancela la operación

Alta de Conceptos

Solicita clave del concepto
Valida la existencia de la clave en el archivo de conceptos
Si ya existe la clave
    Muestra mensaje "Concepto Ya Existe"
Si no existe la clave
    Solicita título del concepto
    Solicita periodicidad de los datos del concepto
    Solicita denominación del concepto
    Almacena en el archivo de conceptos los datos capturados

Baja de Conceptos

Solicita clave del concepto
Valida la existencia de la clave en el archivo de conceptos
Si no existe la clave
    Muestra mensaje "Concepto no Existe"
Si existe la clave
    Muestra los datos del concepto
    Verifica que la clave del concepto no exista en el archivo de
    Botones matriz
    Si existe en el archivo de botones matriz
        Manda mensaje "Concepto Utilizado"
        cancela la operación
    si no existe en botones matriz

Pide confirmación de la baja
si se confirma la baja
Elimina del archivo de conceptos la clave seleccionada
si no se confirma la baja
cancela la operación

Cambio de Conceptos

Solicita clave del concepto
Valida la existencia de la clave en el archivo de conceptos
Si no existe la clave
Muestra mensaje "Concepto no Existe"
Si existe la clave
Muestra los datos del concepto
Solicita los nuevos datos del concepto
actualiza el archivo de conceptos con los nuevos datos

Captura de datos de Conceptos

Solicita el tipo de captura (Manual o por archivo)
si se eligió manual
Solicita la clave del concepto
Verifica la existencia del concepto en el archivo de conceptos
si existe
solicita la fecha y el valor
agrega en el archivo de datos los valores capturados
si no existe
Muestra mensaje "Concepto no existente"
cancela la operación
si es por archivo
solicita el nombre del archivo a importar
para cada una de las líneas del archivo
valida la existencia de la clave de concepto en el archivo de conceptos
si existe

valida la fecha de acuerdo con la periodicidad del concepto
si es correcta
agrega los valores leídos al archivo de datos
si no es correcta
manda la línea leída a un archivo de errores
si no existe
manda la línea leída a un archivo de errores

Alta de Usuarios

Solicita clave del usuario
Valida la existencia de la clave en el archivo de usuarios
Si ya existe la clave
Muestra mensaje "Usuario Ya Existe"
Si no existe la clave
Solicita nombre del usuario
Solicita el password de usuario
Solicita el nivel del usuario (Consulta o administración)
Agrega los datos capturados al archivo de usuarios

Baja de Usuarios

Solicita clave del usuario
Valida la existencia de la clave en el archivo de usuarios
Si no existe la clave
Muestra mensaje "Usuario no Existe"
Si existe la clave
muestra los datos del usuario
confirma la baja del registro
si se confirma la baja
elimina del archivo de usuarios la clave proporcionada
si no se confirma la baja
cancela la operación

Cambio de usuarios

Solicita clave del usuario
Valida la existencia de la clave en el archivo de usuarios
Si no existe la clave
    Muestra mensaje "Usuario no Existe"
Si existe la clave
    muestra los datos del usuario
    solicita los nuevos datos del usuario
    Actualiza el archivo de usuarios con los nuevos datos

Calificación

Solicita las fechas de calificación para todas las periodicidades de conceptos definidas en el sistema
Para todos los botones matriz existentes
Busca en el archivo de datos los valores correspondientes a los conceptos definidos en la regla de evaluación a la fecha determinada
si algún valor no se encontró
    modifica el campo color del archivo de botones matriz con el color GRIS
si todos los valores se encontraron
    evalúa la fórmula del límite inferior
    si la expresión es verdadera
        modifica el campo color del archivo de botones matriz con el valor ROJO
    si la expresión es falsa
        evalúa la fórmula del límite superior
        si la expresión es verdadera
            modifica el campo color del archivo de botones matriz con el valor VERDE
        si la expresión es falsa
            modifica el campo color del archivo de botones matriz con el valor AMARILLO
modifica el campo color del registro correspondiente al botón en archivo de pantallas matriz con el peor de los casos de aquellos botones que tenían marcado el campo de significativo (el peor de los casos es el botón cuyo color tiene mas alta prioridad de acuerdo a la siguiente tabla:

VERDE = 0, GRIS = 1, AMARILLO = 2 y ROJO = 3)
modifica el campo alarma del registro correspondiente al botón en archivo de pantallas matriz con el peor de los casos de aquellos botones que no tenían marcado el campo de significativo (siguiendo el criterio del punto anterior)
Para todas aquellos botones Gato cuyo campo pantalla siguiente = MATRIZ
modifica el archivo de botones gato con el color y la alarma de la pantalla matriz correspondiente
modifica el campo color del registro correspondiente al botón en archivo de pantallas Gato con el peor de los casos de aquellos botones que tenían marcado el campo de significativo (siguiendo el criterio definido)
modifica el campo alarma del registro correspondiente al botón en archivo de pantallas Gato con el peor de los casos de aquellos botones que no tenían marcado el campo de significativo (siguiendo el criterio definido)
Para todos los botones Gato cuyo campo pantalla siguiente = GATO y hasta que todos los botones tengan color y alarma
modifica el archivo de botones gato con el color y la alarma de la pantalla gato correspondiente
modifica el campo color del registro correspondiente al botón en archivo de pantallas Gato con el peor de los casos de aquellos botones que tenían marcado el campo de significativo (siguiendo el criterio definido)
modifica el campo alarma del registro correspondiente al botón en archivo de pantallas Gato con el peor de los casos de aquellos botones que no tenían marcado el campo de significativo (siguiendo el criterio definido)

Armado de Gráficas

Para cada Botón Matriz registrado en el archivo
Busca en el archivo de datos los valores correspondientes a las últimas 13 fechas a partir de la fecha definida en la calificación de cada uno de los dos conceptos que se utilizan en el botón matriz

actualiza el archivo de gráficas con los 26 valores obtenidos, la periodicidad, denominación, titulo de los conceptos y datos de identificación del botón matriz

Áreas

Pantalla_id	text	4	Área a la que pertenece el área
Area_id	byte	1	N° de área
Titulo área	text	30	Titulo del área

Botón Gato

Pantalla_id	text	4	Pantalla a la que pertenece el botón
Área id	byte		Área en la que se ubica el botón
Boton_id	byte		N° de botón
Botón titulo	text	10	Título del botón
color	byte		último status del botón
alarma	byte		Color de alarma del botón
pantalla sig	Text	4	N° de pantalla a la que llama el botón
Tipo siguiente	byte		Tipo de la pantalla siguiente (matriz o Gato)
significativo	byte		El botón acarrea color a pantalla anterior o no

Botón Matriz

Pantalla id	text	4	Pantalla a la que pertenece el botón
renglón	byte		N° de renglón en que se ubica el botón
columna	byte		N° de columna en que se ubica el botón
color	byte		Ultimo status del botón
concepto renglón	text	20	Uno de los 2 conceptos que se relacionan en el botón (Lim. Inferior)
operación renglón	text	2	operador que actuará sobre el valor del concepto renglón. (Lim. Inferior)
constante renglón	single		constante numérica que afectará el valor del concepto renglón

			(Lim. Inferior)
operación de relación	text	2	operador relacional que evalúa los resultados el Lim. inferior
concepto columna	text	20	Uno de los 2 conceptos que se relacionan en el botón (Lim. Inferior)
operación columna	text	2	operador que actuará sobre el valor del concepto columna. (Lim. Inferior)
constante columna	single		constante numérica que afectará el valor del concepto columna (Lim. Inferior)
operación renglón 1	text	2	operador que actuará sobre el valor del concepto renglón. (Lim. Superior)
constante renglón 1	single		constante numérica que afectará el valor del concepto renglón (Lim. Superior)
operación de relación 1	text	2	operador relacional que evalúa los resultados el Lim. Superior
operación columna 1	text	2	operador que actuará sobre el valor del concepto columna. (Lim. Superior)
constante columna 1	single		constante numérica que afectará el valor del concepto columna (Lim. Superior)
significativo	byte		Determina si el botón acarrea su color a la pantalla anterior
Columna Matriz			
pantalla id	text	4	Pantalla a la que pertenece la columna
numero columna	byte		N° de la columna en la pantalla
titulo	text	20	Título de la columna
Conceptos			
llave	text	20	Clave del concepto
descripción	text	40	Descripción del concepto

	periodo	byte		Periodicidad del concepto (diaria, semanal, quincenal, mensual, trimestral y semestral)
	denominación	text	10	unidad de medida del concepto
Datos				
	concepto	text	20	Clave de concepto a la que pertenece el valor
	fecha	text	8	Fecha del dato
	valor	double		importe del dato
Fechas				
	diaria	text	8	última fecha de calificación de los datos con periodicidad diaria
	semanal	text	8	última fecha de calificación de los datos con periodicidad semanal
	quincenal	text	8	última fecha de calificación de los datos con periodicidad quincenal
	mensual	text	8	última fecha de calificación de los datos con periodicidad mensual
	trimestral	text	8	última fecha de calificación de los datos con periodicidad trimestral
	semestral	text	8	última fecha de calificación de los datos con periodicidad semestral
Gato				
	pantalla id	text	4	N° de pantalla
	titulo	text	40	Título de la pantalla
	color	text	1	Peor de los casos de los botones significativos.
	alarma	text	1	peor de los casos de los botones no significativos
Matriz				
	pantalla id	text	4	N° de pantalla

titulo	text	40	Título de la pantalla
color	text	1	Peor de los casos de los botones matriz pertenecientes a la pantalla significativos
alarma	text	1	Peor de los casos de los botones matriz pertenecientes a la pantalla no significativos
renMatriz			
pantalla id	text	4	N° de pantalla a la que pertenece el renglón
numero renglón	byte		N° del renglón en la pantalla matriz
titulo	text	20	titulo del renglón
Usuarios			
usuario	text	8	Clave del usuario
password	text	6	Password del usuario
nivel	text	1	Nivel de acceso del usuario (consulta o administración)
nombre	text	40	Nombre del usuario

CONSULTA

Diagrama de Estados

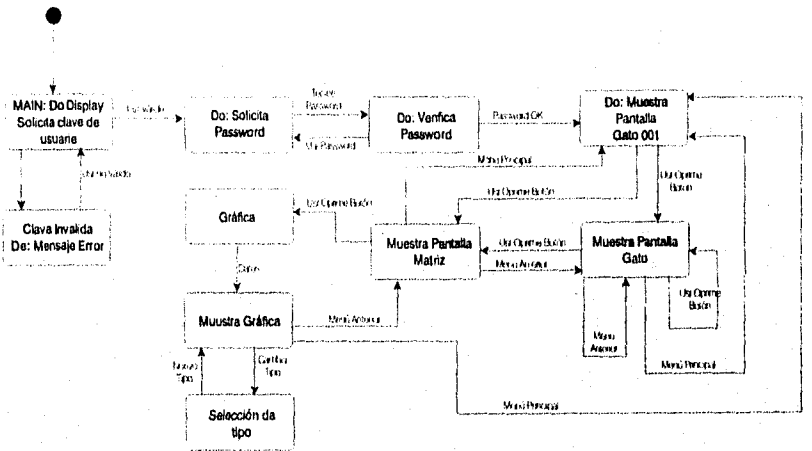
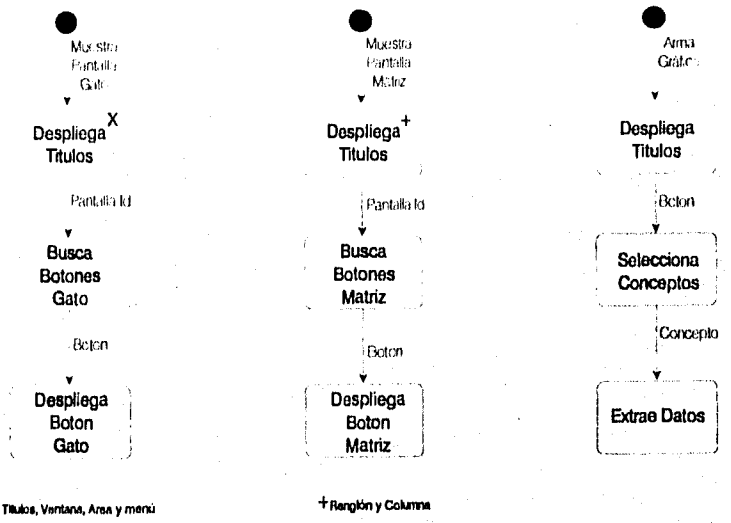


Diagrama de Estados



OBTENCIÓN DE OPERACIONES (Módulo de consulta)

Las operaciones identificadas para cada una de las clases son las siguientes :

Pantallas Gato

- Consulta

Pantallas Matriz

- Consulta

Botón Gato

- Oprime

Botón Matriz

- Oprime

Gráfica

- Consulta

Usuarios

Acceso al sistema

Consulta de Pantallas gato

Busca el archivo de Pantallas Gato, el número de pantalla pasado como parámetro
si no existe
Muestra mensaje de error "Pantalla no definida"
Cancela la operación
si existe
Busca en el archivo de áreas todas aquellas que tengan en el campo N° de pantalla el número pasado como parámetro
Busca en el archivo de Botón Gato todos aquellos que tengan en el campo N° de pantalla el número pasado como parámetro
muestra la pantalla

Consulta de Pantallas Matriz

Busca el archivo de Pantallas Matriz, el número de pantalla pasado como parámetro
si no existe
Muestra mensaje de error "Pantalla no definida"
Cancela la operación
si existe
Busca en el archivo de renglón matriz todos aquellos que tengan en el campo N° de pantalla el número pasado como parámetro

Busca en el archivo de columna matriz todas aquellas que tengan
en el campo N° de pantalla el número pasado como parámetro
Busca en el archivo de Botón matriz todos aquellos que tengan
en el campo N° de pantalla el número pasado como parámetro
muestra la pantalla

Oprime botón Gato
De acuerdo al campo tipo de pantalla siguiente
si la pantalla siguiente es de tipo matriz
ejecuta consulta pantalla matriz pasando como parámetro el valor
del campo N° de pantalla siguiente
Si la pantalla siguiente es de tipo Gato
ejecuta consulta de pantalla gato pasando como parámetro el
valor del campo N° de pantalla siguiente

Oprime botón Matriz
ejecuta la función consulta gráfica pasando como parámetro el número
de pantalla, el renglón y la columna en que se encuentra el botón matriz

Consulta Gráfica

Busca en el archivo de gráficas, el N° de pantalla, renglón y columna
pasados como parámetro
si no se encuentra
Muestra mensaje "Error en el proceso de calificación"
cancela la operación
si se encuentra
arma una gráfica de barras con los datos encontrados en el
registro

Acceso de usuarios al sistema
Permite que el usuario capture su clave
busca en el archivo de usuarios, la clave proporcionada por el usuario
si no se encuentra
muestra mensaje de error "Usuario no registrado"
cancela la operación
si se encuentra
permite que el usuario capture su password

valida el password proporcionado contra el del archivo de usuarios
si no es correcto
    Muestra mensaje de error "password incorrecto"
    cancela la operación
si es correcto
    ejecuta la operación consulta pantalla gato, mandando como parámetro la pantalla N° "1"

BIBLIOGRAFÍA

Planeación estratégica
Lo que todo director debe saber
George A. Steiner
CECSA
1992

Object-Oriented Analysis
Peter Coad/ Edward Yourdon
Yourdon Press
1990

Concepts of Object Oriented Programing
David N. Smith
McGraw Hill
1991

Designing Object Oriented Software
Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener
Prentice Hall
1990

Object Oriented Systems Analysis
Modeling the wold in data
Sally Shlaer, Stephen J. Mellor
Yourdon Press
1988

Object Oriented SoftWare Construction
Bertrand Meyer
Prentice Hall
1988

Structured Analisys and system specification
Tom DeMarco
Prentice Hall
1979

On undestanding types, data abstraction, and polymorphism
Luca Cardelli, Peter Wegner
ACM Computing Surveys

Object Oriented design
Grady Booch
Benjamin/Cummings
1991

Object Oriented Concepts, Databases, and aplications
Wom Kim, Frederick H. Lochovky
ACM Press
1988

Ingeniería del Software, un enfoque práctico.
Roger S. Pressman
Mc Graw Hill
1988

Ingeniería de Software
Richard Fairley
Mc Graw Hill
1989.

Smalltalk-80: The language and its implementation.
Adele Goldberg, David Robson
Addison Wesley
1983

Introducción a los sistemas de bases de datos
C.J. Date
Addison Wesley Iberoamericana
1986.

Análisis estructurado moderno
Edward Yourdon
Prentice Hall
1993

Técnicas de Bases de Datos
Estructuración en diseño y administración.
Shakuntala Atre
Ed. Trillas
1988.

An introduction to object-oriented programing
Timothy Budd
Addison Wesley
1991

Object oriented programing with turbo c++
Keith Weiskamp, Loren Heiny, Bryan Flaming
John Wiley & sons, inc.
1991

Object Oriented analysis and design
James Martin & James J. Odell
Prentice Hall
1991

Object Oriented Software
Ann L. Winblad, Samuel D. Edwards, David R. King
Addison Wesley
1990

Object oriented Software engineering with c++
Darrel Ince
McGraw Hill
1991

Object Oriented Design with aplications
Grady Booch
Benjamin/ Cummings
1991

Object Oriented Programming in Common LISP
Sonya E. Keene
Addison Wesley
1989

Rapid prototyping for object oriented systems
Mark Mullin
Addison Wesley
1990

Eiffel, The language
Bertrand Meyer
Prentice Hall
1992

ÍNDICE ANALÍTICO

—A—

abstracción, 14, 16, 18, 20, 26, 49, 54, 68, 79, 87, 98
análisis, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 22, 23, 25, 26, 27, 29, 32, 33, 43, 44, 45, 46, 47, 51, 53, 65, 67, 69, 70, 71, 73, 77, 79, 83, 84, 85, 86, 87, 88, 89, 96, 103, 116, 117, 121
arquitectura, 16, 43, 45, 47, 49, 60, 61, 68, 89, 102, 108
asociación, 25, 26, 28, 31, 32, 80, 81, 84, 90
asociaciones, 23, 24, 26, 27, 29, 31, 32, 43, 45, 48, 53, 67, 68, 73, 80, 83, 84, 86, 87, 89, 90, 111, 122
atributos, 14, 19, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 39, 43, 64, 67, 73, 74, 75, 78, 79, 80, 81, 82, 83, 86, 87, 90, 99, 101, 102, 103, 105, 108, 110, 111, 114, 123

—C—

capas, 40, 41, 48, 49, 50
clase, 14, 17, 19, 20, 25, 26, 28, 29, 30, 31, 35, 36, 37, 54, 70, 72, 74, 78, 79, 81, 82, 83, 84, 87, 89, 99, 100, 101, 102, 103, 104, 105, 106, 110, 111, 112, 113, 123, 145
Clasificación, 13, 14
compartición, 19

—D—

DBMS, 54, 55
diagrama de estado, 18, 32, 33, 35, 36, 37, 76, 87
diagramas de estado, 18, 22, 33, 36, 37, 38, 39
diagramas de flujo de datos, 18, 22, 40, 41, 61, 63, 93
diagramas de objetos, 18, 22, 35
diccionario de datos, 24, 86
diseño, 11, 13, 15, 16, 17, 18, 26, 45, 46, 47, 48, 51, 53, 58, 59, 60, 63, 65, 67, 68, 69, 71, 72, 73, 75, 77, 79, 80, 82, 83, 84, 85, 86, 88, 89, 91, 92, 94, 96, 97, 99, 105, 166

—E—

encapsulación, 19, 20, 102, 103, 113
eventos, 18, 22, 32, 33, 34, 35, 36, 37, 40, 43, 48, 51, 56, 57, 58, 62, 63, 64, 68, 69, 75, 76, 77, 87, 91

—H—

Herencia, 13, 14, 19, 20, 23, 24, 29, 30, 44, 68, 77, 78, 79, 86, 89, 90, 99, 100, 103, 109, 110, 111, 112, 114, 123

—I—

Identidad, 13, 20

implementación, 13, 14, 15, 16, 17, 18, 19, 24, 25, 27, 29, 30, 33, 42, 45, 46, 47, 49, 51, 56, 58, 61, 67, 68, 70, 71, 72, 73, 75, 79, 80, 81, 82, 83, 84, 85, 86, 89, 90, 96, 99, 102, 103, 104, 108, 110, 111, 113

instancia, 14, 79, 81, 106

Interfase interactiva, 60, 62, 63

—M—

Manejador de transacciones, 60

máquinas de estado, 75, 77

método, 14, 45, 81, 83, 100, 106, 110, 112

Modelo, 13, 16, 17, 18, 22, 23, 24, 25, 26, 29, 30, 31, 32, 33, 34, 38, 39, 40, 41, 42, 43, 44, 45, 51, 53, 54, 57, 60, 61, 62, 63, 64, 65, 67, 68, 69, 70, 71, 72, 73, 74, 75, 77, 79, 80, 82, 83, 84, 85, 86, 87, 88, 89, 90, 93, 94, 97, 108, 114

modelo de objetos, 18, 23, 30, 60, 61, 62, 67, 69, 77, 87, 94

modelo dinámico, 17, 18, 43, 61, 65, 68, 69, 75, 77, 94

modelo funcional, 17, 39, 43, 53, 61, 62, 65, 69, 87

módulos, 24, 32, 68, 82, 84, 87, 90, 91, 103, 105, 121

—O—

Objeto, 13

ocultamiento de información, 19, 49, 82

operaciones, 8, 14, 19, 21, 24, 26, 29, 31, 34, 41, 43, 44, 48, 49, 54, 55, 57, 58, 61, 62, 63, 64, 65, 67, 68, 69, 70, 71, 72, 75, 77, 78, 79, 80, 83, 84, 87, 89, 90, 92, 99, 100, 103, 104, 105, 110, 111, 112, 116, 121, 144, 145, 160

Orientado a Objetos, 13, 99

—P—

particiones, 48, 50

Polimorfismo, 13, 14, 20, 113

prototipo, 77, 80

—R—

requerimientns, 9, 20, 22, 23, 33, 45, 84, 86, 94

—S—

seudocódigo, 39, 42, 44, 97
Simulación dinámica, 60, 63, 64
sistema de tiempo real, 64
Sistema en tiempo real, 60
subclase, 14, 79, 100, 103, 110
subsistemas, 16, 47, 48, 49, 50, 51, 52, 53, 54, 56, 59, 68, 88
superclase, 14, 15, 29, 30, 79, 103, 110, 123

—T—

Transformación continua, 60
transformación continua, 61
Transformación en lote, 60