4. Implementación

En el capítulo de implementación analizaremos las diferentes formas de completar el análisis y diseño orientados a objetos mediante lenguajes de programación y sistemas de bases de datos. Estudiaremos la implementación en lenguejes de programación orientados a objetos, al igual que en lenguajes de programación no orientados a objetos. En el tema de base de datos, estudiaremos la implementación en sistemas de bases de datos orientados a objetos al igual que en los sistemas de base de datos relacionales.

4.1 De Diseño a Implementación

4.1.1 Implementación usando lenguajes de programación

La implementación del diseño de los sistemas orientados a objetos es más fácil usando lenguajes de programación orientados a objetos. En general, no todos los lenguajes de programación implementan de la misma forma los diferentes conceptos de orientación a objetos. Mas aún, los diferentes lenguajes varían en el nivel de apoyo a los conceptos básicos. La gran mayoría de estos lenguajes de programación son capaces de expresar los siguientes tres aspectos:

• Estructuras de datos: definido de forma declarativa
• Flujo dinámico de control: este puede ser procedural (condiciones, ciclos, llamadas) o declarativo (reglas, restricciones, tablas). No existe apoyo a concurrencia en la mayoría de los lenguajes, excepto Ada. La concurrencia puede ser simulada usando corutinas, modelos de control, o manejadores de eventos. Multi-procesamiento y comunicación entre procesos son provistos por el sistema operativo.
• Transformaciones funcionales: son expresadas por medio de operadores primitivos y subrutinas.

4.1.2 Implementación usando sistemas de base de datos

Cuando la mayor preocupación es el acceso a datos almacenados, más que la implementación de operaciones de datos, una base de datos es el mecanismo más apropiado para la implementación. Los comandos de las bases de datos operan en conjuntos de datos, que pueden pertenecer a diferentes usuarios, y ser accesados de forma concurrente. Por otro lado, los lenguajes de programación operan por lo general de forma procedural en programas pertenencientes a un solo usuario y de forma secuencial. Las operaciones de la base de datos, aunque menos procedurales que en los lenguajes convencionales, son más procedurales que en los sistemas basados en reglas.

4.2 Estilo de Programación

4.2.1 Estilo Orientado a Objetos

Los buenos programas hacen más que simplemente satisfacer requisitos funcionales, deben seguir guías de diseño para que los programas sean correctos, extensibles, y corregidos facilmente. La

mayoría de las guías de programación convencionales también se aplican a programas orientados a objetos. Además, las facilidades particulares de los lenguajes orientados a objetos, como herencia, requieren nuevas guías.

Guías de programación:
1) Reuso
2) Extensibilidad
3) Robustez
4) Mega-programación

4.2.1.1 Reuso

El reuso de código reduce el tiempo del diseño, la codificación, y el costo del sistema al amortizar el esfuerzo sobre varios diseños. El reducir la cantidad de código también simplifica su entendimiento, aumentando la probabilidad de que el código sea correcto. El reuso es posible en lenguajes convencionales, pero los lenguajes orientados a objetos aumentan substancialmente las posibilidades de tal reuso, gracias a la modularidad de los sistemas.

4.2.1.1.1 Tipos de reuso

Hay dos tipos de reuso:
• compartir el código escrito recientemente dentro de un proyecto, y
• reuso en nuevos proyectos de código anteriormente escrito.
Guías similares se aplican a ambos tipos de reuso.

Compartir el código dentro de un proyecto es simplemente cuestión de descubrir secuencias de código redundante en el diseño y usar las facilidades del lenguaje de programación, como procedimientos o métodos. Este tipo de compartición de código siempre es bueno al producir programas más pequeños y correcciones más rápidas.

Reuso para el futuro requiere planeación y representa una inversión. No es muy probable que una clase aislada sirva para múltiples proyectos. Los programadores tienden a reusar subsistemas bien planificados, como tipos de datos abstractos, paquetes gráficos, y librerías de análisis numérico.

4.2.1.1.2 Reglas de Estilo para Reuso

• Mantener los métodos coherentes: los métodos son coherentes si ejecutan una sola función o un grupo de funciones muy relacionadas. Si se mezclan dos o más cosas no relacionadas, se debe repartir en métodos separados.

• Mantener los métodos pequeños: si los métodos son grandes, partirlos en métodos más pequeños. Un método que excede una o dos páginas es probablemente demasiado grande. Al partir un método en partes más pequeñas se podría reusar algunas partes aunque el método completo no se reuse.

• Mantener los métodos consistentes: métodos similares deben usar los mismos nombres, condiciones, orden de argumentos, tipo de datos, valor de retorno, y condiciones de error. *Unix* ofrece ejemplos de inconsistencia en funciones. Por ejemplo, existen dos funciones inconsistentes para manipulación de cadenas, *puts* y *fputs*. La función *puts* escribe una cadena a la salida estándar, seguida de una carácter de nueva línea; mientras que *fputs* escribe una cadena a un archivo particular, sin el carácter de nueva línea. Se debe evitar este tipo de inconsistencias.

• Separar la política de la implementación: métodos "políticos" toman decisiones, varían los argumentos y juntan contextos globales, cambiando control entre métodos de implementación. Métodos políticos deben verificar estados y errores, y no debieran calcular o implementar directamente algoritmos complejos. Métodos políticos dependen usualmente de la aplicación, pero son simples de entender y de escribir. Métodos de implementación deben ejecutar operaciones detalladas, sin decidir cuando o por que se hacen. Si el método de implementación encuentra errores, debe devolver solo un estado pero no debe tomar ninguna acción particular. Los métodos de implementación especifican la computación sobre argumentos completos y a veces contienen algoritmos complicados. Los métodos de implementación no accesan contexto global, no toman decisiones, no contienen valores por omisión, o cambian el flujo de control, pudiendo ser reusados en otros contextos. No se debe combinar política e implementación en un sólo método. Por ejemplo, un método para escalar una ventana por 2 es un método político ya que implica no solamente el escalamiento de la ventana, sino también la proporción del escalamiento.

• Proveer cobertura uniforme: si las condiciones de entrada pueden ocurrir en varias combinaciones, se deben escribir métodos para todas las combinaciones, no simplemente las que se necesiten en ese momento. Por ejemplo, si se escribe un método para obtener el último elemento de una lista, también es conveniente escribir uno para obtener el primero.

• Ampliar el método lo máximo posible: se trata de generalizar tipos de argumentos, precondiciones y restricciones, suposiciones sobre el comportamiento del método, y el contexto en el cual el método opera. Se deben tomar acciones en el caso de valores vacíos, extremos, o fuera de rango. Usualmente un método puede hacerse general con un poco más de código.

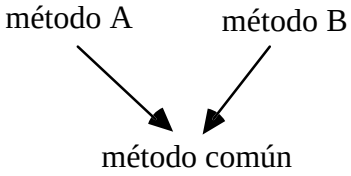
• Evitar información global: minimizar las referencias externas, que requieren uso de contexto. Frecuentemente se puede pasar la información como argumento. De lo contrario, guardar la información global como parte del objeto para que otros métodos la accesen de forma uniforme.

• Evitar modos: las funciones que cambian su comportamiento drásticamente dependiendo del contexto son difíciles de reusar. Cambiar por funciones sin modos. Por ejemplo, una aplicación de procesamiento de texto requiere operaciones de insertar y reemplazar. Un enfoque es tener un modo para *insertar* o *reemplazar*, y luego usar una operación de *escribir* para insertar o reemplazar texto dependiendo del modo actual. Un enfoque sin modos usa dos operaciones, *insertar* y *reemplazar*, que hacen la misma operación sin depender del modo. El peligro de modos es que un objeto dejado en un modo en una parte de la aplicación puede afectar más tarde una operación en la aplicación.

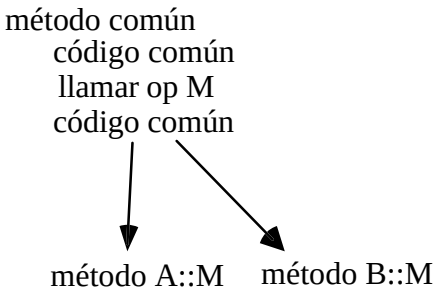
4.2.1.1.3 Usando herencia

Las guías anteriores mejoran la oportunidad de heredar código compartido. Algunas veces, los métodos en clases diferentes son similares pero no lo suficiente para representarse con un solo método heredado. Hay varias técnicas para partir métodos y heredar el código.

• subrutinas: es el enfoque más sencillo, con un llamado a una subrutina. Se pasa el código común a un método que es luego llamado por cada método. El método común es asignado a una clase ancestra.



• distribuir: a veces la mejor forma para incrementar el reuso de código entre clases similares es distribuir las diferencias entre los métodos de las diferentes clases, dejando el resto del código como un método compartido. Este enfoque es efectivo cuando las diferencias entre los métodos son pequeñas y las similitudes son grandes. La parte común se lleva a un nuevo método el cual llama la operación, implementada por un método diferente, omitiendo las diferencias de código en cada subclase. A veces una clase abstracta debe ser añadida para guardar el método común (para todas las subclases.).



• delegación: el uso de herencia podría incrementar el reuso de código dentro de un programa, aún cuando una verdadera relación superclase/subclase no existe. Se debe evitar utilizar *herencia de implementación*, y en su lugar se debe usar delegación. La herencia debe ser usada sólo si la relación de generalización es semánticamente válida. Herencia significa que cada instancia de una subclase es verdaderamente una instancia de la superclase, y por lo tanto todos los atributos y operaciones de la superclase se deben aplicar a la subclase. El uso impropio de herencia hace que los programas sean difíciles de mantener y extender. Delegación provee un buen mecanismo para lograr el reuso deseado de código. La operación se guarda en la clase deseada y mandada a otra clase para su verdadera ejecución. Ya que cada operación debe ser enviada de forma explícita, los efectos secundarios inesperados son raros que ocurran.

• código externo encapsulado: a veces se desea reusar código que ha sido desarrollado para una aplicación con diferentes convenciones de interfaz. En lugar de insertar una llamada directa al código externo, es más seguro encapsular su comportamiento dentro de una operación o clase.

De esta forma la rutina o paquete externo puede ser cambiado o reemplazado, y solo se necesitaría cambiar el código en un sólo lugar.

Por ejemplo: se tiene una aplicación de análisis numérico, pero sabiendo que un buen software para invertir matrices ya existe, no se desea re-implementar el algoritmo en un lenguaje orientado a objetos. La clase matriz podría ser escrita para encapsular la funcionalidad provista por el paquete de subrutina externa.

4.2.1.2 Extensibilidad

La mayoría de los sistemas son extendidos de formas que los desarrolladores originales no tenían previsto. Las guías para reuso mejoran también la extensibilidad. Los siguientes principios amplían la extensibilidad:

• Encapsular clases: Una clase es encapsulada si su estructura interna está oculta de las otras clases. Sólo los métodos de la clase deben accesar su implementación.

• Estructurar datos ocultos: No exportar estructuras de datos desde un método. Las estructuras de datos internas son específicas al algoritmo del método. Si se exporta se limita la flexibilidad de cambiar el algoritmo más tarde.

• Evitar atravesar múltiples ligas o métodos: Un método debe tener un conocimiento limitado del modelo de objeto. Un método debe ser capaz de atravesar ligas entre un objeto y sus vecinos, y debe ser capaz de llamar operaciones en ellos. Por otro lado, no debiera atravesar una segunda liga desde el vecino a una tercera clase porque la segunda liga no es directamente visible al primer método. En su lugar, se debe llamar una operación en el objeto vecino para atravesar la operación. Si la red de asociaciones cambia, el método de la operación puede ser reescrito sin cambiar la llamada.

• Evitar oraciones de casos (*case* statements) sobre tipos de objetos: Usar métodos en lugar de oraciones de casos usados para seleccionar el comportamiento basado en el tipo del objeto. Despachar operaciones basadas en tipo de objetos es la razón de ser de los métodos.

• Distinguir entre operaciones privadas y públicas: Operaciones públicas son visibles fuera de la clase y tienen interfaces públicas. Cuando una operación pública es usada por otras clases, es costoso cambiar la interfaz, por lo cual las operaciones públicas deben ser definidas con cuidado. Las operaciones privadas son internas a la clase y ayudan a implementar operaciones públicas. Las operaciones privadas pueden ser removidas o su interfaz cambiada para modificar la implementación de la clase con impacto limitado a los otros métodos de la clase. Los atributos y asociaciones deben ser clasificadas como privadas o públicas, y los atributos y asociaciones públicas deben ser clasificadas para sólo leer o también escribir fuera de la clase que es la dueña.

4.2.1.3 Robustez

Se debe tratar de ser eficiente al escribir métodos pero no a expensas de la robustez. Un método es robusto si no falla aunque reciba parámetros impropios. Robustez contra errores internos puede ser balanceada contra el rendimiento.

• Protección contra errores: el software debe estar protegido contra entradas incorrectas del usuario, lo cual nunca debería causar una caída del sistema. Cualquier método que acepta una entrada del usuario debe validar la entrada que pueda causar problemas. El diseñador de métodos debe considerar dos tipos de condiciones de error:

(1) Errores de aplicación (usuario) que son identificados durante análisis.

Por Ejemplo: La aplicación del ATM debe reportar o procesar errores sobre la lectura de tarjetas en el ATM y líneas de comunicación. La respuesta a estos errores es parte del análisis.

(2) Errores de bajo nivel del sistema se ocupan de aspectos de programación del método, incluyendo errores del sistema operativo, como errores de asignación de memoria, o errores de archivos de entrada y salida, y fallas de hardware.

El programa se debe proteger contra estos errores y dar buena información de diagnósticos aunque una falla fatal ocurra. Durante el desarrollo es a veces bueno insertar oraciones internas en el código para descubrir fallas, aunque luego sean removidas durante la producción. Un lenguaje orientado a objetos de tipos fuertes (*strongly-typed*) provee mayor protección contra errores de tipo.

• Optimizar después de correr el programa: No optimizar el programa hasta que este funcione. A menudo los programadores dedican demasiado esfuerzo a mejorar partes del código que ejecutan poco frecuente. Se busca medir el rendimiento del programa antes de optimizarlo. Las operaciones pueden implementarse en más de una forma. Se debe analizar las alternativas según se relacionan a memoria, velocidad, y simplicidad de implementación. También es necesario evitar optimizar más de lo necesario, ya que la optimización compromete la extensibilidad, reuso y comprensión. Si los métodos están encapsulados de forma apropiada, pueden ser reemplazados con versiones optimizadas sin afectar el resto del programa.

• Validar argumentos: Las operaciones externas disponibles a los usuarios de la clase deben tener los argumentos revisados rigurosamente para evitar fallas. Los métodos internos pueden asumir que los argumentos son válidos por razones de eficiencia. Los métodos públicos deben tener más cuidado al revisar la validez de sus argumentos, ya que los usuarios externos están más propensos a violar las restricciones en los argumentos. Los métodos internos o privados pueden asumir precondiciones ya que el programador tiene mayor control y se basa en métodos públicos para revisar los posibles errores. No se deben incluir argumentos que no puedan ser validados.

Ejempolo: la función *scanf* en *Unix* lee una línea de entrada del *buffer* interno sin revisar el tamaño del buffer. Este problema ha sido aprovechado para escribir programas de virus que fuerzan un sobreflujo en el buffer del software del sistema, ya que no valida sus argumentos.

• Evitar límites predefinidos: En lo posible se debe usar una asignación de memoria dinámica para crear estructuras de datos que no tengn límites predefinidos. Durante el diseño es difícil predecir la capacidad máxima esperada para la estructura de datos en la aplicación.

• Instrumentar el programa para monitoreo de rendimiento y búsqueda de errores: Al igual que el diseñador de un circuito de hardware instrumenta la tarjeta integrada con puntos de verificación, se debe instrumentar el código para búsqueda de errores, estadísticas, y rendimiento. El nivel de búsqueda de errores que se debe construir depende del ambiente de programación presente en el lenguaje. Si el lenguaje de implementación no lo proporciona se pueden añadir métodos de impresión para cada clase. También se pueden añadir mensajes de entrada y salida a los métodos, imprimiendo selectivamente valores de entrada y salida. Algunos sistemas operativos incluyen herramientas para analizar la ejecución del programa.

Ejemplo: Smalltalk ofrece varias herramientas para tales propósitos.

4.2.1.4 Mega-Programación

Mega-programación se refiere a escribir programas grandes y complejos con un equipo de programadores. La comunicación humana es crítica en tales proyectos y requiere prácticas de ingeniería de software apropiadas:

• No comenzar prematuramente a programar: Es importante primero completar el proceso de pensamiento genérico, primero diseñar y luego implementar.

• Mantener los métodos claros: Un método es claro si alguien fuera del creador del método es capaz de entender el código. El mantener los métodos pequeños y coherentes ayuda a lograrlo.

• Hacer métodos legibles: Nombres de variables con sentido incrementan la legibilidad de los métodos. Se debe agregar algunos caracteres adicionales para no tener que descifrar luego los nombres de las variables. Usar variables temporales en lugar de expresiones demasiado anidadas. No usar la misma variable temporal para dos propósitos diferentes dentro de un método, aunque su uso no se traslape.

• Usar exactamente los mismos nombres como en el modelo de objetos: Los nombres usados dentro del programa deben corresponder a los encontrados en el modelo de objetos. Un programa puede necesitar introducir nombres adicionales por razones de implementación, lo cual está bien, pero se debe preservar la uniformidad, la cual mejora el entendimiento y la documentación.

• Escoger nombres con cuidado: Asegurar que los nombres describen con exactitud las operaciones, clases, y atributos que etiquetan. Seguir un patrón uniforme en el momento de asignar los nombres. No usar el mismo nombre para operaciones semánticamente diferentes, e.g. *invertir-matriz*, *invertir-figura*.

• Usar guías de programación: Equipos de proyecto deben usar guías de programación existentes en la organización. Si no existen, el equipo de software deben crear sus propias guías para decidir

aspectos como la forma de los nombres de las variables, estilo de programación, métodos de documentación, y documentación en línea.

• Empacar en módulos: Agrupar en un módulo las clases con funciones similares.

• Documentar clases y métodos: La documentación de un método describe su propósito, función, contexto, entradas y salidas al igual que cualquier suposición y precondición sobre el estado del objeto. Se debe describir el algoritmo, incluyendo por qué fué escogido. Comentarios internos dentro del método deben describir pasos mayores.

• Publicar la especificación: La especificación es un contrato entre el productor y el consumidor de la clase. Cuando una especificación ha sido escrita, el productor no puede romper el contrato, ya que hacerlo afectaría al consumidor. La especificación solo contiene declaraciones. Algunos lenguajes como Ada y C++, apoyan la separación de la especificación del código de su implementación. Descripciones en línea de la clase y sus características ayudan a promover el uso correcto de la clase. Cada operación también debe ser documentada por separado, dando su origen, significado general, diciendo que es lo que debe hacer cada subclase para implementar la operación, y qué métodos relacionados son necesarios para la subclase.

Ejemplo de una especificación parcial:

Descripción de Clase
Nombre de Clase: Círculo
Versión: 1.0
Descripción: Elipse cuyos ejes mayor y menor son iguales
Super Clases: Elipse
Características:
    Atributos Públicos:
        centro: Punto - lugar de su centro
        radio: Real - su radio
    Métodos Públicos:
        dibujar(Ventana) - dibujar un círculo en la ventana
            intersectarLínea(Línea): Conjunto de Puntos - encontrar la intersección de
        una línea y un círculo, retornar un conjunto de 0-2 puntos
            área(): Real - calcular área del círculo
            perímetro(): Real - calcular circunferencia del círculo
    Métodos Privados: ninguno

Descripción de Método:
Método Círculo::intersectarLínea(línea:Línea): Conjunto de Puntos
    Descripción: Dado un círculo y una línea, encontrar la intersección, retornar un conjunto de 0-2 puntos de intersección. Si la línea es tangente al círculo, el conjunto contiene un solo punto.
    Entradas:
        self:Círculo - círculo que va ser intersectado por la línea
        línea:Línea - línea que va ser intersectada por el círculo
    Retorna:
        Un conjunto de puntos de intersección. Puede contener 0, 1, o 2 puntos.
    Efectos Secundarios: ninguno
    Errores: Si la figura no es intersectada, retornar un conjunto vacío.
        Si la línea es tangente al círculo, retornar un punto tangente.
        Si el radio del círculo es 0, retornar un solo punto si el punto está en la línea.

Descripción de Operación:
Operación intersectarLínea(línea:Línea): Conjunto de Puntos
Clase Original: FiguraGeométrica
Descripción: Retornar un conjunto de puntos de intersección entre el objeto geométrico y la línea.
El conjunto puede contener 0, 1, o mas puntos. Cada punto tangente aparece una sola vez. Si la línea es colineal con un segmento de línea en la figura, solo los dos puntos extremos del segmento son incluidos.
Status: Operación abstracta en la clase original, debe ser sobrescrita.
Entradas:
self:FiguraGeométrica - figura que va ser intersectado por la línea
línea:Línea - línea que va ser intersectada por el círculo
Retorna:
Un conjunto de puntos de intersección. Puede contener 0 o mas puntos.
Efectos Secundarios: ninguno
Errores: Si la figura no es intersectada, retornar un conjunto vacío
Si la línea es colineal con un segmento de línea en la figura, el conjunto incluye solamente los puntos en el extremo del segmento.
Si la figura es un área, entonces su borde es usado.

4.3 Lenguajes Orientados a Objetos

En general, un lenguaje orientado a objetos apoya el concepto de objetos (datos + operaciones), encapsulamiento, paso de mensajes, polimorfismo, y herencia.

4.3.1 Traducir el Diseño a una Implementación

Los tres modelos del OMT contribuyen al desarrollo de código:
• El modelo de objeto contiene la mayoría de la estructura declarativa: especificación de clases, atributos, jerarquía de herencia y asociaciones.
• El modelo dinámico especifica la estrategia de alto control: procedural, evento, o multi-tarea.
• El modelo funcional captura la funcionalidad de los objetos que deben incorporarse en los métodos.

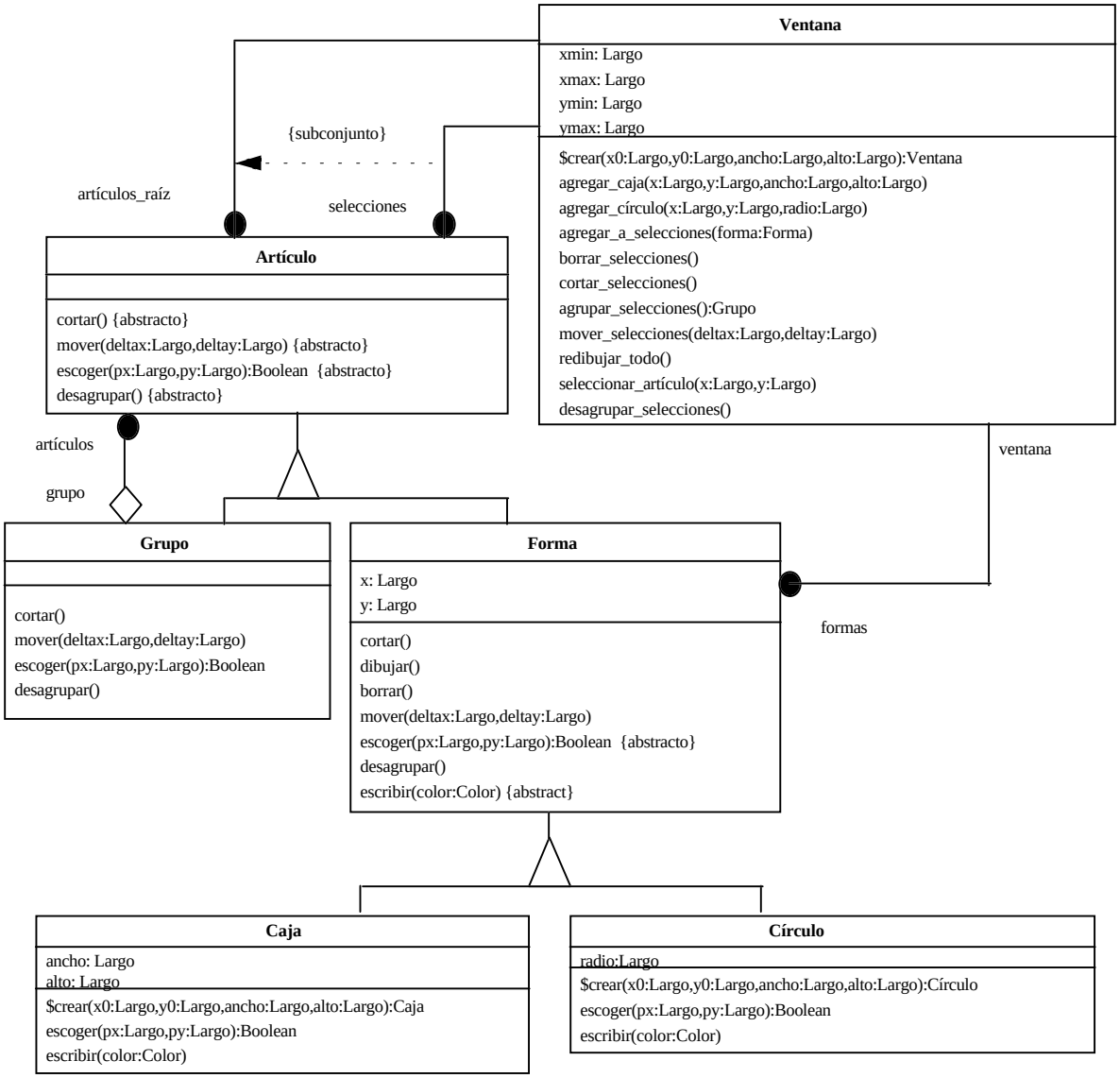


Figura 4.1. Diagrama de clases para un Editor Gráfico

Consideraciones:
1) Definir las clases
2) Crear objetos
3) Llamar operaciones
4) Usar herencia
5) Implementar asociaciones

Existen varios lenguajes orientados a objetos, e.g. C++, Smalltalk, Eiffel. Usaremos C++ para los ejemplos presentados en esta sección.

Ejemplo Editor Gráfico: El *Editor Gráfico* se muestra en la Figura 4.1. El editor permite *grupos* recurrentes de *formas* que se construyan de *cajas* y *círculos*. Una *ventana* contiene un *grupo* de *formas*. Los *grupos* pueden construirse de *formas* o *grupos* más pequeños. *Artículos* (*grupos* o

formas) que no son parte de un *grupo,* son *artículos-raíz* en la *ventana* y se pueden manipular. Un *artículo-raíz* puede ser seleccionado al escoger una de sus *formas* con el ratón. Una *forma* se escoge si el cursor lo apunta. Los *artículos* seleccionados se pueden *agrupar*, *borrar*, o *mover*. Existen comandos para *borrar* la selección o *crear* nuevas *formas*. Las *formas* se borran al ser sobrescritas con el color de otras.

4.3.1.1 Definir las Clases

En C++ se declaran las clases de objetos usando métodos públicos o privados. Los atributos y métodos son miembros de la clase. Un método no puede tener el mismo nombre que un atributo. El constructor es el método para crear el objeto, y debe tener el mismo nombre que la clase. El destructor es el método que se llama antes de la destrucción del objeto, y debe tener el mismo nombre de la clases con el prefijo "~". (La definición de *Largo* puede ser hecha usando un *typedef* en C++ para *int* o *float*.)

```
class Ventana
{
private:
        Largo xmin;
        Largo ymin;
        Largo xmax;
        Largo ymax;
        void agregar_a_selección(Forma* forma);
public:
        Ventana(Largo x0, Largo y0, Largo ancho, Largo alto);   // Constructor
        ~Ventana();                                                    // Destructor
        void agregar_caja(Largo x, Largo y, Largo ancho, Largo alto);
        void agregar_círculo(Largo x, Largo y, Largo radio);
        void borrar_selecciones();
        void cortar_selecciones();
        Grupo* agrupar_selecciones();
        void mover_selecciones(Largo deltax, Largo deltay);
        void redibujar_todo();
        void seleccionar_artículo(Largo x, Largo y);
        void agrupar_selecciones();
};
```

4.3.1.2 Crear Objetos

Los lenguajes orientados a objetos, por lo general, instancian nuevos objetos de una de dos formas. Algunos lenguajes como Smalltalk, tienen clases que son objetos de por sí. En estos lenguajes, una operación aplicada al objeto *clase* instancia un nuevo objeto. Otros lenguajes, como C++ o Eiffel, no tienen objetos *clase*. Estos lenguajes tiene operaciones especiales para crear nuevos objetos. Cuando un nuevo objeto es creado, el lenguaje debe asignar almacenamiento para guardar los valores de sus atributos y debe asignar un identificador de objeto único, que puede ser la dirección del bloque de almacenado o un índice en una tabla.

Lenguajes diferentes usan uno de dos estilos para destruir objetos que ya no son necesarios. En algunos lenguajes, como en C++, los objetos son destruidos por una operación explícita. El

programador debe tener cuidado que no queden referencias a objetos destruidos, o resultarían en errores de memoria. Otros lenguajes, como Smalltalk, evitan la posibilidad de errores en el manejo de memoria de forma explícita, para lo cual utilizan un recolector de basura automática que destruye los objetos que son inaccesibles.

C++ no tiene objetos clase u operaciones de clase para crear instancias de una clase. En su lugar, tiene una operación especial, *constructor*, para inicializar nuevas instancias:

```
Ventana::Ventana(Largo x0,Largo y0,Largo ancho,Largo alto)
{
        xmin = x0; ymin = y0;
        xmax = x0 + ancho; ymax = y0 + alto;
}
```

Pueden ser definidos múltiples constructores para una sola clase, distinguidos por el número y tipo de sus argumentos (*overloading*):

```
Ventana(); // omisión de posición y tamaño
Ventana(Largo x0,Largo y0); // omisión de tamaño
Ventana(Largo x0,Largo y0,Largo ancho,Largo alto);
```

El constructor es ejecutado cuando se crea un nuevo objeto. Al momento de la asignación, el programador puede especificar los argumentos del constructor; y el constructor es ejecutado con argumentos correspondientes. Si no se especifican argumentos, se ejecuta el constructor sin argumentos.

Cada clase puede tener un destructor, el cual ejecuta cualquier limpieza necesaria antes de destruir a un objeto. Destructores no llevan argumento:

```
Ventana::~Ventana()
{
        // borrar la ventana y redibujar la sección
}
```

C++ provee tres tipos de asignación de memoria para objetos: preasignada por el compilador en memoria global fija (estática), asignada a la pila (automática), y asignada al *heap* (dinámica).

• El almacenamiento estático es obtenido declarando las variables fuera de una función o usando la palabra *static* en un atributo. Un atributo estático es un atributo de clase común a todas las instancias de la clase. Almacenamiento estático es preasignado por el compilador y no cambia durante la ejecución.

Ejemplo: Una variable global estática que guarda un objeto *Ventana* inicializado:

```
Ventana ventana_principal = Ventana(0.0,0.0,8.5,11.0);
```

• Las variables locales dentro de funciones usan normalmente un almacenamiento automático. Cuando una función es entrada, se reserva suficiente espacio en la pila para guardar todas las variables locales de la función. El almacenamiento es desasignado cuando la función sale, y las referencias a las variables automáticas no deben ser guardadas en otros objetos cuya vida exceda el de las variables automáticas. La declaración de variables locales tiene la misma forma que la declaración de variables globales, excepto que son declaradas dentro del cuerpo de una función. Cuando una función sale, se llama al destructor para cada variable automática. Luego de que el destructor ha sido corrido, el almacenamiento para variables automáticas es implícitamente desasignado cuando la instrucción de retorno de la función actualiza la pila a su valor antes de la llamada.

• Almacenamiento dinámico es asignado de un *heap* en un pedido explícito del programador. El operador *new* asigna almacenamiento para un nuevo objeto y devuelve un apuntador. El operador *new* puede incluir argumentos para el constructor:

```
Ventana *ventana = new Ventana(0.0,0.0,8.5,11.0);
```

Objetos dinámicos pueden ser desasignados solamente aplicando el operador *delete* al apuntador del objeto. El programador debe asegurar que no existan referencias al objeto. El operador *delete* debe invocar primero el destructor de la clase y luego desasignar el almacenamiento para el objeto.

```
delete ventana;
```

Las implementación estándares de C++ no incluyen un recolector de basura, existiendo el riesgo de referencias incoherentes a objetos removidos, al igual que pérdida de memoria a objetos inaccesibles que no han sido removidos.

4.3.1.3 Llamar operaciones

En la mayoría de los lenguajes orientados a objetos, cada operación tiene por lo menos un argumento implícito, el del propio objeto, indicado con una sintaxis especial. Las operaciones pueden o no tener argumentos adicionales. Algunos lenguajes permiten pasar argumentos como valores o como referencias a valores que pueden ser actualizados por el procedimiento.

Una operación en C++ es declarada como un miembro de la clase, junto con sus atributos. Una operación es llamada usando una notación similar al acceso a atributos: el operador de selección de membresia "->" es aplicado a un apuntador de objeto:

```
Forma* forma;
forma->mover(dx,dy);
```

Los argumentos adicionales pueden ser objetos, tipos básicos en C++ como *int*, *float* y *char*, y tipos definidos por el usuario, como tipos *typedef*.

Un nombre de atributo o nombre de operación usado como identificador dentro de un método en C++ se refiere implícitamente al objeto al cual se aplica el método. Por ejemplo, *x* y *y* son atributos de la clase *Forma*:

```
void Forma::mover(Largo deltax,Largo deltay)
{
        x = x + deltax;
        y = y + deltay;
}
```

El atributo implícito *this* contiene una referencia al objeto del contexto, siendo equivalente la siguiente descripción:

```
void Forma::mover(Largo deltax,Largo deltay)
{
        this->x = this->x + deltax;
        this->y = this->y + deltay;
}
```

La referencia a un atributo de un objeto que no es el objeto del contexto requiere una referencia calificada:

```
ventana->xmin = x1;
```

A menos que el objeto calificado sea de la misma clase que el método, tal acceso de atributo directo debe ser evitado ya que viola el encapsulamiento de clase. Se debe usar un método de acceso en la clase, que permita que la representación de la clase pueda ser cambiada. A diferencia de Smalltalk y Eiffel, C++ distingue entre el objeto y su referencia. Acceso a objetos directamente (no a través de apuntadores) es descrito por la siguiente notación de punto:

```
ventana.mover(dx,dy);
```

4.3.1.4 Usar herencia

Los lenguajes orientados a objetos varían según los mecanismos provistos para implementar herencia. Se puede analizar tres dimensiones independientes para clasificar herencia:
• estática o dinámica,
• implícita o explícita, y
• por objeto o por grupo.
Muchos de los lenguajes populares son estáticos (herencia es definida durante la compilación), con herencia implícita (el comportamiento del objeto depende de su clase, que no puede ser cambiada), y por grupo (las características de la herencia están especificadas para una clase, no para objetos específicos). En la mayoría de lenguajes, la declaración de cada clase incluye una lista de superclases de las cuales se hereda los atributos y métodos.

En C++, la superclase o superclases de una clase son especificadas como parte de la declaración de la clase. Una subclase es llamada una clase *derivada*. Por ejemplo, se declara *Forma* como una subclase de *Artículo,* mientras que *Caja* y *Círculo* son subclases de *Forma*:

```
class Artíiculo
{
public:
        virtual void cortar() = 0;
        virtual void mover(Largo deltax,Largo deltay) = 0;
        virtual Boolean escoger(Largo px,Largo py) = 0;
        virtual void desagrupar() = 0;
};
```

```
class Forma: public Artículo
{
        Largo x;
        Largo y;
public:
        void cortar();
        void dibujar() { escribir(COLOR_FOREGROUND); }
        void borrar() { escribir(COLOR_BACKGROUND); }
        void mover(Largo deltax,Largo deltay);
        virtual Boolean escoger(Largo px,Largo py) = 0;
        void desagrupar() {}
        virtual void escribir(Color color) = 0;
};
```

```
class Caja : public Forma
{
protected:
        Largo ancho;
        Largo alto;
public:
        Caja(Largo x0,Largo y0,Largo ancho0,Largo alto0);
        Boolean escoger(Largo px,Largo py);
        void escribir(Color color);
};
```

```
class Círculo : public Forma
{
protected:
        Largo radio;
public:
        Circulo(Largo x0,Largo y0,Largo radio0);
        Boolean escoger(Largo px,Largo py);
        void escribir(Color color);
};
```

El conocimiento de los ancestros de una clase se puede declarar en C++ como *public* o *private*. Si la derivación de una superclase es *private,* entonces los clientes de la clase no pueden llamar directamente a las operaciones heredadas o accesar atributos de los ancestros. Atributos u

operaciones declarados *protected* son accesibles a subclases, pero no a otras clases fuera de la jerarquía de herencia.

```
Boolean caja::escoger(Largo px,Largo py)
{
        return x<=px && px<=x+ancho && y<=py && py<=y+alto;
}
```

Los métodos declarados en una superclase son también heredados. Si un método puede ser sobrescrito por una subclase, entonces debe ser declarado en C++ como *virtual* en su primera aparición en la superclase.

Ejemplo: El método *escribir* en la clase *Forma* puede ser sobrescrito en las clases *Caja* y *Círculo*, y es por lo tanto virtual. Los métodos *dibujar* y *borrar* en la clase *Forma* no están sobrescritos por las subclases y no tiene que ser declarados virtuales. El método *Forma::escribir* es declarado virtual e inicializado a 0, esto lo identifica con una *función virtual pur*a, o sea una operación abstracta. Cualquier clase con una función virtual pura es una clase abstracta y no puede ser instanciada directamente. El compilador verifica que cualquier subclase concreta defina o herede implementaciones para todas las funciones virtuales puras. Los métodos que sobrescriben métodos heredados deben ser redeclarados en la subclase, pero aquellos que son heredados (y no sobrescritos) no tiene que repetirse. Las operaciones virtuales son llamadas usando la misma sintaxis como para las operaciones no virtuales. Los atributos heredados no tienen que repetirse.

C++ apoya herencia múltiple, por medio de la especificación de una lista de superclases en la derivación de una clase. No es permitido el uso de nombres conflictivos de atributos u operaciones de diferentes superclases. C++ apoya variaciones complicadas para herencia múltiple, incluyendo cadenas de constructores que son automáticamente invocados cuando se crea una nueva instancia.

4.3.1.5 Implementar Asociaciones

Hay dos enfoques generales para implementar asociaciones: usar apuntadores y crear distintos objetos de asociación. Si el lenguaje no apoya explícitamente la asociación (y la mayoría no lo hacen), entonces apuntadores es lo más fácil de implementar. Apuntadores se pueden añadir durante el diseño o postergar hasta el final. En cualquier caso, los atributos necesarios para implementar los apuntadores deben ser añadidos a las definiciones de las clases. Una asociación binaria es implementada usualmente como un atributo en cada objeto de la asociación, conteniendo un apuntador al objeto relacionado o a un conjunto de objetos relacionados. En muchos casos, la asociación se atraviesa en una sola dirección, y el apuntador se puede agregar a una sola de las clases. Apuntadores en la dirección "uno" son fáciles de implementar, son simplemente referencias a objetos. Apuntadores del lado "muchos" requieren un conjunto de objetos, o un arreglo de asociaciones ordenadas, implementadas más fácilmente con una clase de *conjunto*. Una asociación también puede ser implementada como un objeto contenedor. Un objeto de asociación explícito es conceptualmente un conjunto de *tuplas*, cada *tupla* contiene una valor para cada clase asociada.

Hay una consecuencia importante en el uso de apuntadores. Las asociaciones no pueden ser simuladas por atributos en clases sin violar el encapsulamiento de la clase ya que no son independientes el par de atributos componiendo la asociación. Actualizando un apuntador en la implementación de la asociación requiere que el otro apuntador se actualice para mantener consistente la asociación. Los atributos individuales no deben ser accesibles externamente ya que no se deben actualizar de forma independiente. Por otro lado, los métodos accesibles externamente para actualizar los atributos no pueden ser añadidos a una de las clases de la asociación sin accesar la implementación interna del otro objeto, ya que los atributos están mutuamente restringidos. C++ rompe de forma limitada con el encapsulamiento por medio de la construcción *friend*, mientras que Eiffel provee exportación de aspectos a clases seleccionadas, aunque sin la limpieza para encapsular asociaciones como existe en Smalltalk.

En C++, aa asociación "muchos-uno" entre *Artículo* y *Grupo* se puede implementar usando apuntadores:

```
class Artículo
{
        // otras declaraciones
private:
        Grupo *grupo;
        friend Grupo::agregar_artículo(Artículo*);
        friend Grupo::remover_artículo(Artículo*);
public:
        Grupo *sacar_grupo() { return grupo; };
};
```

```
class Grupo : public Artículo
{
        // otras declaraciones
private:
        ConjuntoArtículo *artículos;
public:
        agregar_artículo(Artículo*);
        remover_artículo(Artículo*);
        ConjuntoArtículo *sacar_artículo() { return artículos; };
};
```

Cuando una nueva liga se añade a la asociación, ambos apuntadores deben ser actualizados, y cuando la liga es removida, ambos apuntadores también deben ser actualizados:

```
void Grupo::agregar_artículo(Arículo *artículo)
{
        artículo->grupo = this;
        artículos->agregar(artículo);
}
```

```
void Grupo::remover_artículo(Artículo *artículo)
{
        artículo->grupo = 0;
        artículos->remover(artículo);
}
```

Los métodos de *Grupo* pueden actualizar al atributo *grupo* en un objeto *Artículo* ya que son declarados como *friend* de la clase *Artículo*. La construcción *friend* provee un mecanismo para dar acceso a detalles internos sin destruir totalmente el encapsulamiento. (En este ejemplo se ha omitido el código para revisar si un artículo ya existe en el grupo. Antes de añadir un artículo al grupo, se debe ver si ya existe y removerlo si es necesario.) La clase *ConjuntoArtículo* es una clase de *colección* que guarda el conjunto de artículos:

```
class ConjuntoArtículo
{
public:
        ConjuntoArtículo();             // crear un conjunto vacío
        ~ConjuntoArtículo();            // destruir el conjunto
        void agregar(Artículo*); // agregar artículo al conjunto
        void remover(Artículo*);        // remover artículo del conjunto
        Boolean incluye(Artículo*);     // ver si el artículo existe
        int número();                   // dar el número de artículos en el conjunto
};
```

Clases de colección que son mas generales (pero con tipos débiles) se encuentran en librerías de clases como en la librería de clases NIH [Gorlen-90] producida por el Insituto Nacional de Salud de USA. La librería incluye clases tipo *iterador* para buscar objetos en las colecciones sin saber la estructura interna de la colección.

```
Boolean Grupo::escoger(Largo x,Largo y)
{
        IteradorArtículo it(Artículos);
        while (it++)
                if (it()->escoger(x,y)) return TRUE;
        return FALSE;
}
```

Clases de arreglos con tamaño variable se pueden usar para implementar asociaciones ordenadas. Algunas aplicaciones requieren otro tipo de estructura de datos, como listas circulares o árboles. (Clases generales como *Conjunto* se pueden hacer específicas a *Artículo* por medio de tipos *parametrizados*.)

Objetos de asociaciones distintas no son difíciles de implementar, en el caso de que la librería de clases no las contenga. *Asociación* puede ser implementada como una nueva clase contenedora y puesta en la librería de clases. El enfoque más sencillo es implementar un objeto de asociación con dos objetos diccionario. El diccionario hace un mapa entre la dirección hacia delante y la dirección hacia atrás de la asociación. Ambos diccionarios deben ser actualizados cuando la asociación es actualizada.

Cada asociación en el modelo de objetos es implementado como un objeto de asociación. Las ligas en la asociación son elementos del objeto de la asociación. Usar objetos de asociación preserva el encapsulamiento de las restricciones entre las clases implícitas en la asociación.

```
Asociación* artículo_grupo_asn = new Asociación(mucho_a_uno);
```

```
void Grupo::agregar_artículo(Articulo *artículo)
{
        artículo_grupo_asn->agregar(artículo,this);
}
```

```
void Grupo::remover_artículo(Artículo *artículo)
{
        artículo_grupo_asn->remover(artículo,this);
}
```

```
Grupo *Artículo::sacar_grupo()
{
        return (Grupo*) artículo_grupo_asn->indice_adelante(this);
}
```

```
ArtículoConjunto* Grupo::sacar_artículos()
{
        return (Grupo*) artículo_grupo_asn->indice_atrás(this);
}
```

Si una asociación es implementada por un objeto separado, no hay necesidad de añadir atributos a las clases en la asociación. Si la asociación está "esparcida" (o sea una pequeña fracción de los objetos participan en ella), entonces los objetos de asociaciones separadas usan menos espacio que los apuntadores para cada objeto. Acceso a valores es un poco más lento que con apuntadores en objetos, pero si las asociaciones son implementadas con diccionarios que contiene tablas de acceso, el tiempo de acceso se puede hacer pequeño, independiente del número de ligas en la asociación.

4.3.2 Aspectos de Lenguajes Orientados a Objetos

Los lenguajes orientados a objetos varían en su apoyo a los conceptos más avanzados de orientación a objetos. No hay un lenguaje que satisfaga todas las necesidades.

4.3.2.1 Herencia Múltiple

Algunos lenguajes, como C++, CLOS, Eiffel apoyan herencia múltiple, donde una clase puede tener más de un ancestro con el mismo atributo. Herencia múltiple introduce la posibilidad de conflictos entre nombres de atributos y operaciones. Hay varios enfoques que toman los lenguajes para resolver el problema. Por ejemplo, el compilador de Eiffel rechaza programas con tales conflictos. CLOS tiene un protocolo para resolverlos. En general se recomienda evitar tales conflictos. No es bueno depender del compilador para resolver tales conflictos, porque se restringe la extensibilidad y puede dar lugar a confusiones.

4.3.2.2 Librería de Clases

La mayoría de los lenguajes orientados a objetos incluyen una librería con clases genéricas, que pueden ser usadas tal como son, o pueden ser modificadas según las necesidades específicas por medio de la creación de subclases. La existencia de librerías de clases significa que muchos componentes no tienen que ser re-implementados por el programador. Las clases más útiles son las que implementan estructuras generales, como conjuntos, arreglos dinámicos, listas, filas, pilas, diccionarios, árboles, etc. Estas clases, a veces conocidas como clases contenedoras, sirven para organizar conjuntos de otros objetos. Las librerías de clases más completas pueden incluir abstracciones de interfaces, como *streams*, o procesos concurrentes, como *corutinas*.

4.3.2.3 Eficiencia

Algunos lenguajes orientados a objetos tienen una reputación de ineficiencia, como Smalltalk y los lenguajes basados en Lisp, ya que originalmente eran interpretados y no compilados. La existencia de compiladores e interpretadores más eficientes ha dado al desarrollador más posibilidades para escoger el lenguaje apropiado. El uso de un lenguaje orientado a objetos con una buena librería de clases significa usualmente que el código corre más rápido que si se escribiera con un lenguaje no orientado a objetos.

Un aspecto de los lenguajes orientados a objetos que parece ineficiente es el uso de *resolución de métodos* durante la ejecución (también conocido como *dynamic binding*) para implementar operaciones polimórficas. La resolución de métodos es el proceso de corresponder una operación de un objeto a un método específico dentro de la jerarquía de herencia (*polimorfismo*). Esto pareciera requerir una búsqueda en el árbol de herencia durante la ejecución para encontrar la clase que implementa la operación en un objeto específico. La mayoría de los lenguajes optimizan los mecanismos de búsqueda para hacerlo más eficiente. Mientras la estructura de clases permanece igual durante la ejecución del programa, el método correcto para cada operación puede ser guardado localmente en la subclase. Con esta técnica, conocida como *caching de método*, la ligadura dinámica puede ser reducida a una sola tabla de búsqueda y ejecutarse en tiempo constante sin importar la profundidad del árbol de herencia o el número de métodos en la clase.

En lenguajes con tipos fuertes, como C++, el costo de la resolución de métodos durante la ejecución puede ser reducido a una sola estructura de referencia, casi insignificante comparado al costo de una llamada de un procedimiento. Cada clase tiene una estructura que contiene métodos accesibles por los objetos de la clase. Un apuntador a la estructura del método para la clase es guardada en cada objeto. Esta técnica no puede ser usada si nuevos métodos pueden ser creados durante la ejecución, ya que la estructura no puede ser precomputada.

Un lenguaje puede optimizar más la resolución de métodos usando tablas de búsqueda dinámica sólo cuando son necesarias. Un gran porcentaje de llamadas de métodos en una aplicación puede ser ligada de forma estática por el compilador si se le da suficiente información. Hay dos formas por la cual el compilador puede adquirir tal información. Una forma es que el programador declare cuales operaciones pueden ser sobrescritas, limitando la ligadura dinámica a sólo esas

funciones. Las funciones virtuales en C++ y las funciones genéricas en CLOS reflejan este enfoque. El otro enfoque es proveer una etapa de optimización final que analiza de forma completa la aplicación, decidiendo qué métodos no son sobrescritos, y luego recompilando la aplicación basada en esta esta información. Este enfoque es usado por Eiffel. El tipo de las declaraciones de las variables permiten al compilador ser mucho más preciso en determinar cuales llamadas de métodos pueden ser optimizados.

4.3.2.4 Tipo Fuerte (*strong type*) y Tipo Débil (*weak type*)

Los lenguajes orientados a objetos varían mucho en su enfoque a *tipos*. El término *tipo* se refiere a si cada valor de una variable y atributo es solamente conocido como un objeto (*tipo débil*), o si puede ser declarado de forma más precisa perteneciendo a una clase particular o a uno de sus descendientes (*tipo fuerte*). El ejemplo de la *Ventana* y las *Forma* asume un tipo fuerte. Por ejemplo, el atributo *radio* es declarado para ser de tipo *Largo*. Smalltalk es un lenguaje con tipo débil, todas las variables en Smalltalk son objetos de una clase no especificada. Eiffel, Objetctive-C, C++ son lenguajes de tipo fuerte. CLOS y Objetctive-C también apoyan tipos fuertes. Una variable que referencia un objeto puede ser declarado, y el compilador verifica el uso de la variable. Lenguajes híbridos, como C++ y Objective-C, permiten que un atributo asuma un valor que no es un objeto pero que es un tipo básico en C, incrementando así la eficiencia de la operaciones.

Tipo fuerte en un lenguaje sirve para dos propósitos. Tipo fuerte permite un apoyo activo al programador en detectar fallas en argumentos de métodos y en oraciones de asignación, y también aumenta las oportunidades de optimización. El compilador puede detectar cuando una operación general puede ser reemplazada por una llamada de una función específica. Ningún poder se pierde con el uso de tipo fuerte, ya que el programador siempre puede declarar todo que sea de una clase *Objeto* para obtener el mismo efecto como en tipo débil.

Aunque tipo débil es flexible y poderoso, permite prácticas peligrosas de codificación. La teoría de los lenguajes de programación modernos ha estado evolucionando en la dirección de tipos fuertes, donde los errores son más fáciles de detectar y corregir durante la compilación que durante la ejecución, mejorando así la calidad del software.

4.3.2.5 Manejo de memoria

La libertad con la cual los objetos pueden ser creados y accesados puede ser un problema cuando el espacio de memoria no puede ser reclamado. La mayoría de los lenguajes orientados a objetos asignan memoria del *heap* y no de bloques fijos de memoria global o de una pila. Un sistema de asignación de memoria dinámica puede terminar con la memoria a menos que los objetos que no son necesarios sean desasignados. El principal problema es determinar cuando un objeto ya no es necesario o cuando ya el objeto no puede ser accesado. La severidad del problema depende del tipo de aplicación y la arquitectura de la memoria. Muchos programas que tiene acceso a grandes cantidades de memoria virtual pueden simplemente ignorar el problema. Por otro lado, una aplicación muy interactiva que se espera corra indefinidamente debe considerar el manejo de memoria sin importar que tanto espacio de memoria exista.

Hay dos enfoques para el manejo de memoria: se hace automáticamente por el sistema durante la ejecución del lenguaje o se hace explícitamente por oraciones de desasignación hechas por el programador. El enfoque preferido es el manejo automático de la memoria, ya que le quita la responsabilidad al programador de decidir cuando desasignar la memoria, evitando el riesgo de mantener información incoherente. Por otro lado, algunos mecanismos de manejo de memoria automática resulta en largas pausas en lugares impredecibles durante la ejecución. Mecanismos de recolección de basura incrementales evitan largas tardanzas pero imponen un costo promedio más alto.

CLOS y Smalltalk ofrecen recolección de basura totalmente automática. C++ requiere que el programador desasigne objetos innecesarios, pero C++ permite al programador definir funciones destructoras, que son llamadas automáticamente cuando una variable se destruye. Esto da al programador una forma conveniente de organizar y llamar desasignación de almacenamiento explícito con el operador *delete*. Desasignación explícita corre el doble peligro de desasignar objetos que todavía son referenciados (dando la posibilidad de fallas del sistema) y no desasignar objetos que ya no son referenciados (desaprovechar la memoria), pero no implica tiempo adicional o costo de memoria en el programa.

4.3.2.6 Encapsulamiento

Encapsulamiento (ocultar información) consiste en la separación del *protocolo* (los aspectos externos del objeto que son accesibles por otros objetos) de los detalles de implementación interna del objeto (que son ocultados de otros objetos). El encapsulamiento previene que cambios pequeños en el programa afecten de forma crítica al resto del sistema, pudiéndo cambiar la implementación de una clase sin afectar a sus clientes. Se puede cambiar la implementación de un objeto para mejorar su rendimiento, corregir un error, o llevarlo a un nuevo sistema. El encapsulamiento no es único en lenguajes orientados a objetos, pero la habilidad de combinar estructuras de datos con comportamiento en una sola entidad hace que el encapsulamiento sea más limpio y más poderoso que en los lenguajes convencionales que separan estructura de datos y comportamiento. Los lenguajes varían en el grado del encapsulamiento.

Una forma en la cual el encapsulamiento puede ser violado ocurre cuando el código asociado con una clase accesa de forma directa los atributos de otra clase. Acceso directo requiere hacer suposiciones sobre el formato del almacenamiento y el lugar de los datos. Estos detalles de implementación deberían estar ocultos dentro de la clase. El modo correcto de accesar un atributo de otro objeto es preguntar por él por medio de una operación del objeto. Muchos lenguajes, como Smalltalk, prohiben el acceso directo a atributos de otros objetos, en cambio lenguajes como C++ permite que los atributos se declaren como privados o públicos. Eiffel provee quizás el control más fino de encapsulamiento por medio de la oración de *export*, el cual lista los atributos que pueden ser leídos y las operaciones que pueden ser ejecutadas. La declaración en C++ *friend* permite abrir el encapsulamiento para acceso de clases o funciones específicas.

El encapsulamiento puede ser violado por el uso sin cuidado de la herencia. Una subclase hereda los atributos de sus superclases, pero si los métodos de la superclase accesan directamente los atributos de la superclase, el encapsulamiento de la superclase es roto. CLOS, Eiffel, y C++ permiten a una clase restringir su visibilidad de sus subclases. A veces es útil escribir algunas

operaciones como privadas que son solamente para uso interno de otros métodos de la misma clase. Es deseable restringir la visibilidad de estas operaciones para que otras clases no las puedan usar. Algunos lenguajes permiten un nivel adicional de acceso a las subclases, como en C++ la declaración *protected*.

4.3.2.7 Empaque

La clase no es una construcción para estructurar grandes sistemas. La mayoría de los lenguajes orientados a objetos no incluyen mecanismos de partición para controlar la visibilidad entre clases, como en el caso de módulos. Esto un problema ya que los lenguajes orientados a objetos requieren que los nombres de las clases sean únicos, por lo cual si se combinan dos o más aplicaciones que fueron desarrolladas de forma independiente, podría existir un conflicto entre nombres. Este problema es similar en lenguajes tradicionales.

En lenguajes como Ada y CLOS, la construcción *package* provee los medios para estructurar un sistema en componentes separados con su propio espacio de nombres. En Ada, los nombres declarados en la especificación del paquete no son visibles fuera del paquete a menos que sea pedido explícitamente, dando control del espacio de nombres y dependencias entre paquetes. Entidades que son conocidas por un nombre fuera del paquete pueden ser renombradas dentro. Paquetes pueden estar anidados, dejando que un subsistema entero sea encapsulado dentro de una interfaz bien definida.

4.3.2.8 Ambiente de desarrollo

Las herramientas que existen para analizar y editar el código fuente, para la compilación, búsqueda de errores, integración del sistema, y verificación, tienen un gran efecto en la productividad del programador. Herramientas de apoyo son especialmente importantes en lenguajes orientados a objetos por la dificultad de manejar herencia y ligaduras de aspecto dinámico.

Un *browser* permite explorar el código fuente en una forma estructurada, viendo qué clases están presentes y qué operaciones están definidas para cada clase. Smalltalk incluye tal herramienta.

El *compilador* o *interpretador* es la herramienta de implementación más importante. Se afecta la velocidad con la cual se hacen cambios y se verifican los resultados en un programa, si el lenguaje es interpretado, compilado, o traducido. Un interpretador permite la búsqueda de errores de la forma más flexible. Los lenguajes como Smalltalk y Lisp permiten que la ejecución del programa se pare, se pueda editar para corregir un error, y luego resumir. Objective-C ofrece un interpretador opcional ademas del compilador.

Algunos compiladores traducen un lenguaje orientado a objetos a un lenguaje intermedio (como C) que es luego compilado. C++ fue originalmente implementado traduciendo a C, pero existen también compiladores verdaderos. La traducción puede causar problemas para los *symbolic debuggers*, ya que no entienden el código original. Al evaluar un *debugger*, se debe ver si muestra el código original fuente o el código intermedio.

4.3.2.9 Metadatos

Metadatos son datos que describen datos. Los metadatos que están presentes durante la ejecución de un programa permiten que se razone sobre la aplicación y posiblemente se pueda modificar su propia estructura y capacidades. Estos cambios pueden incluir las operaciones que el objeto apoya, los atributos que posee, o los propios tipos de atributos. Hay usos más convencionales para metadatos, como procedimientos genéricos para imprimir o guardar cualquier objeto. Los lenguajes que contienen objetos descriptores explícitos de clases, como Smalltalk, contiene metadatos de ejecución sobre las clases. Un objeto *clase* actúa como un *template* para la creación de nuevas instancias. La clase también contiene descripciones de atributos y operaciones, como el nombre, tipo, y argumentos. El uso de metadatos durante la ejecución, más que durante la compilación, permite la construcción de sistemas extensibles con procedimientos generales y clases abstractas que se pueden reusar en futuras aplicaciones. Los metadatos también pueden usarse para apoyar la persistencia o para apoyar herramientas de desarrollo, tales como *debuggers*.

4.3.2.10 Clases parametrizadas

Clases parametrizadas, o genéricas, permiten que un *template* parametrizado sea escrito, puediéndo ser aplicado a varios casos que difieran solamente en el tipo de parámetros. Por ejemplo, una clase genérica *Lista* puede ser instanciada como una *lista de puntos* o una *lista de enteros*. El método genérico *agregar_elemento* en la lista depende del tipo de elemento. Clases parametrizadas existen en Ada, Eiffel, y C++.

4.3.2.11 Afirmaciones (*assertions*) y Restricciones

Afirmaciones y *restricciones* mejoran la posibilidad de que el comportamiento de una clase corresponda a lo esperado por el cliente. Esto se hace informalmente escribiendo una descripción en lenguaje natural del comportamiento deseado. Existen algunas suposiciones críticas sobre el comportamiento de una clase, operación, o métodos que puedan ser expresados matematicamente. Estas suposiciones toman la forma de *afirmaciones* que deben ser ciertas durante puntos particulares en la ejecución, como precondiciones y postcondiciones, y *restricciones* que deben ser mantenidas, o *invariantes* que siempre deben ser ciertas.

Afirmaciones deben ser escritas de forma que se puedan compilar de forma opcional, y que luego se puedan revisar durante la ejecución. Esta técnica se puede incorporar de varias formas, como el macro *assert* en varias de las implementaciones de C, aunque es particularmente bien apoyada en Eiffel.

Restricciones pueden ser más que simplemente condiciones a ser verificadas. Deben verse como una forma de expresar declarativamente en lugar de código procedural. Un lenguaje puede asegurar que las restricciones se mantengan durante la ejecución, y tomar acciones para mantenerlas. El lenguaje lógico Prolog tiene tales capacidades.

4.3.2.12 Persistencia de datos

Todos los programas operan sobre datos. Si se requiere que los datos persistan más allá de la vida de la ejecución de un solo programa, es necesario usar un almacenamiento de datos permanente:
• Datos persistentes pueden proveer la manera más sencilla de pasar datos entre programas.
• Datos persistentes permiten que el mismo programa resuma procesamiento más tarde.
• Almacenamiento de datos son útiles para guardar información.

4.3.3 Comparación de lenguajes orientados a objetos

4.3.3.1 Smalltalk

Smalltalk fue el primer lenguaje popular orientado a objetos, desarrollado en Xerox PARC, y su éxito ha engendrado muchos otros lenguajes orientados a objetos. Smalltalk no es solamente un lenguaje pero también un ambiente de desarrollo incorporando algunas funciones del sistema operativo. Para desarrollo de un solo usuario, ofrece posiblemente los mejores aspectos de lenguaje y ambiente. Smalltalk es un poco limitado en las áreas para las cuales no fue diseñado, como proyectos de múltiples personas y para interactuar con software externo o dispositivos de hardware. Smalltalk es un lenguaje muy elegante, con extensibilidad y reuso. Es un sistema orientado a objetos muy puro con metadatos extensivos, modificables durante la ejecución.

La sintaxis del lenguaje es simple. Las variables y atributos no tienen tipos, todo es un objeto, incluyendo las clases. Clases pueden ser añadidas, extendidas, revisadas, y depuradas interactivamente. Un recolector de basura libera al programador de la responsabilidad de manejar la memoria.

La mayor contribución de Smalltalk es el gran ambiente interactivo de desarrollo, que evita el ciclo *editar-compilar-ligar*, con las tardanzas de los lenguajes compilados. El ambiente de Smalltalk permite un desarrollo rápido de programas. Otro aspecto importante es su librería de clases, que fue diseñada para extenderse y adaptarse añadiendo subclases según las necesidades de la aplicación. Como Smalltalk es un lenguaje sin tipo, los componentes de librerías se pueden combinar rápidamente para crear programas prototipo.

Smalltalk incluye librerías de clases que proveen versiones estándares, a los cuales se les puede añadir subclases y pueden ser extendidos de forma incremental.

4.3.3.2 C++

C++ es un lenguaje híbrido, en donde algunas entidades son objetos y otras no. C++ es una extensión de C, implementada no solo añadiendo capabilidades orientadas a objetos pero también complementando algunas de las debilidades de C. C++ es un lenguaje con tipos fuertes, originalmente implementado como un preprocesador traduciendo C++ a C, y actualmente existiendo como un compilador completo. Muchos de los nuevos aspectos son ortogonales a la programación orientada a objetos, como expansión *inline* de subrutinas y *overloading* de funciones. Por su origen como una extensión de C, C++ es uno de los lenguajes orientados a objetos dominantes.

A diferencia de otros lenguajes orientados a objetos, C++ no contiene una librería de clases estándar como parte de su ambiente, aunque existen librerías para I/O, corutinas y aritmética compleja. Librerías de clases han sido escritas por varios desarrolladores, como las de NIH. Librerías de clases para sistemas de ventanas orientadas a objetos incluyen Interviews [Vlissides-88] y ET++ [Weinand-88]. Y como C++ no provee una guía para la organización de librerías, las diferentes librerías pueden no ser compatibles.

C++ contiene facilidades para herencia y resolución de métodos durante la ejecución, pero una estructura de datos en C++ no es automáticamente orientada a objetos. La resolución de métodos y la habilidad para sobrescribir una operación en una subclase son posibles sólo si la operación es declarada *virtual* en la superclase. Por lo cual la necesidad de sobrescribir un método debe ser anticipado y escrito en la definición de la clase original, restringiendo seriamente la habilidad de reusar librerías de clases, especialmente si el código fuente para la superclase no existe.

La implementación de la resolución de métodos durante la ejecución es eficiente. Para cada clase, una estructura predefinida (*struct*) es inicializada con apuntadores para cada método de la clase. Durante la ejecución, una operación virtual es resuelta buscando la estructura del método del objeto y seleccionando un miembro para encontrar la dirección del método. C++ no apoya objetos descriptores de clases durante la ejecución más que para las estructuras apuntadoras de métodos. C++ apoya herencia múltiple.

C++ contiene buenas facilidades para especificar el acceso a atributos y operaciones de la clase. El acceso es permitido para métodos de cualquier clase (*public*), restringido para métodos de las subclases de la clase (*protected*), o restringido para métodos directos de la clase (*private*). Además se da acceso a una clase particular o una función usando la declaración *friend*.

C++ apoya *overloading* de operadores: varios métodos compartiendo el mismo nombre pero con diferente número y tipo de argumentos. C++ apoya varias estrategias de asignación de memoria para objetos, asignación estática por el compilador, basado en pilas, y asignación dinámica durante la ejecución. El programador debe evitar mezclar objetos de diferente tipo. Cada clase puede tener varios constructores. En resumen, C++ es complejo, con eficiencia de ejecución a expensas de simplicidad y flexibilidad en el diseño.

4.3.3.3 Eiffel

Eiffel es un lenguaje orientado a objetos de tipos fuertes. Los programa consisten de colecciones de declaraciones de clases que incluyen métodos. Se apoya herencia múltiple, clases parametrizadas (*generics*), manejo de memoria, y afirmaciones (*assertions*). Una pequeña librería de clases es provista, incluyendo listas, árboles, pilas, filas, archivos, cadenas, y tablas. El compilador de Eiffel traduce a C para ser más portatil. Eiffel tiene buenas facilidades para encapsulamiento, y control de acceso.

El foco de Eiffel es la declaración de clases, que lista atributos y operaciones. Eiffel provee acceso uniforme a atributos y operaciones abstrayéndolos en un solo concepto llamado una *característica*. Una declaración de clase en Eiffel puede incluir una lista de características

exportadas, una lista de clases ancestras, y una lista de declaraciones de características. Eiffel no trata a la clases ni a las asociaciones como objetos de primera clase.

Eiffel apoya el manejo de memoria por medio de un recolector de basura, el cual detecta objetos que no son referenciados, y desasigna la memoria asignada a ellos. El sistema de ejecución de Eiffel ejecuta el recolector de basura cuando el espacio de memoria existente es bajo. Varios mecanismos son provistos para controlar el manejo de memoria. Ejecución automática del recolector de basura puede suprimirse por medio de un interruptor que se puede prender o apagar durante la ejecución. Para sistemas operativos que no apoyan memoria virtual, hay un interruptor en el compilador para modificar el sistema de ejecución de Eiffel para proveer paginación automática.

Un modelo contractual de programación es apoyado por precondiciones, postcondiciones, invariantes, y excepciones. Una *precondición* es una condición que el llamador de una operación está de acuerdo en satisfacer. Una *postcondición* es una condición que la operación está de acuerdo en lograr. Una *invariante* es una condición que una clase debe satisfacer todo el tiempo. Condiciones e invariantes son partes de la declaración de las clases y deben ser obedecidas por todas las clases descendientes. Si se violan una de esta condiciones durante la ejecución, una excepción ocurre causando la falla de la operación, o la llamada a un manejador de excepción para la clase. Interruptores del compilador proveen varios niveles de verificación de errores. Una vez que la aplicación es depurada se puede apagar el verificador de afirmaciones.

4.3.3.4 CLOS

Common Lisp Object System (CLOS) es una extensión orientada a objetos de Common Lisp, basado en otras extensiones de Lisp, como Flavors y CommonLoops. Originalmente fue implementado de forma híbrida, integrándose bien con las características de Common Lisp. Cada objeto, incluyendo átomos y listas de Lisp, son miembros de una clase. Los métodos para estrcturas de datos primitivos de Lisp pertenecen a la estructura de herencia.

El ambiente de programación de Common Lisp y CLOS es un interpretador que permite que el código se compile para mayor eficiencia de ejecución. Facilidades de depuración bajo Common Lisp dependen de la implementación específica pero son por lo general muy buenas.

CLOS no incluye actualmente una librería de clases estándares. CLOS provee capabilidades poderosas y flexibles de herencia. Herencia múltiple es apoyada, y CLOS tiene reglas para resolver ambigüedades que resultan de características de herencia con el mismo nombre. Las operaciones polimórficas que requieren resolución de métodos dinámicos pueden ser implementadas como métodos genéricos. Todos los argumentos de un método genérico son explícitos.

CLOS provee una gran colección de metadatos que pueden ser accesados y actualizados durante la ejecución. Nuevas clases pueden ser definidas, y los métodos pueden ser añadidos a las clases de forma dinámica. Estas características son parte estándar del lenguaje, conocido como el *protocolo meta-objeto*. A diferencia de la mayoría de los lenguajes, los lenguajes basados en Lisp permiten la construcción de nuevos procedimientos durante la ejecución.

Como Common Lisp, CLOS es un lenguaje con tipos débiles. Tipos básicos son provistos, y las clases se comportan como tipos, pero no hay ningún requisito de declarar el tipo de una variable, y no hay ninguna verificación para asegurar que el uso del objeto sea consistente con su declaración. Mientras se mantiene una máxima flexibilidad de diseño, los tipos débiles pueden afectar el rendimiento y la facilidad de la depuración. Declaraciones opcionales pueden ser usadas por el compilador para optimizar acceso, pero estas optimizaciones dependen de la implementación específica.

Características:	**C++**	**Smalltalk**	**CLOS**	**Eiffel**	**Objective-C**
Integración de clases con tipos primitivos	híbrido	puro	integrado	integrado	híbrido
Tipo fuerte	Si	No	No	Si	Si
Habilidad para restringir acceso a atributos:					
Control de acceso de clientes	Si	Si	No	Si	Si
Control de acceso de subclases	Si	No	No	Si	No
Librería de clases estándar	No	Si	No	Si	Si
Clases parametrizadas	Si	N/A	N/A	Si	No
Herencia múltiple	Si	No	Si	Si	No
Scoping de nombres de clases	No	No	Si	No	No
Modelo de mensajes:					
Objeto sencillo	Si	Si	No	Si	Si
Ligadura dinámica en múltiples argumentos	No	No	Si	No	No
Combinación de métodos:					
concepto SUPER	No	Si	Si	Si	Si
métodos antes y después	No	No	Si	No	No
Afirmaciones y restricciones	No	No	No	Si	No
Metadatos durante ejecución	No	Si	Si	No	Si
Recolección de basura	No	Si	Si	Si	No
Eficiencia:					
Ligadura estática cuando es posible	Si	No	No	Si	Si

Figura 4.2. Compración entre diferentes lenguajes orientados a objetos. (N/A - *No Aplicable*: clases parametrizadas no son necesarias en lenguajes con tipos débiles.)

El concepto de encapsulamiento no está contenido en CLOS. Los programadores pueden definir y documentar la interfaz pública de cada clase y usar solamente las características anunciadas a otras clases, pero no hay nada que prevenga el código de una clase de accesar directamente los detalles de implementación de otra clase. La falta de control sobre encapsulamiento es consistente con la política general de Lisp de proveer máxima flexibilidad.

La tabla que se muestra en la Figura 4.2 compara de forma resumida los diferentes lenguajes orientados a objetos.

4.4 Lenguajes No Orientados a Objetos

Usar un lenguaje orientado a objetos o uno no orientado a objetos no es un problema de funcionalidad. Se puede traducir cualquier construcción orientada a objetos a un lenguaje no orientado a objetos. El poder computacional no es nunca un problema ya que cualquier lenguaje universal puede computar cualquier cosa computable.

El verdadero problema con los lenguajes no es su poder pero la expresibilidad, conveniencia, protección contra errores, y mantenimiento. Un lenguaje orientado a objetos hace que la escritura, mantenimiento, y extensión de los programas sea más fácil y segura, ya que ejecuta tareas que un programador de un lenguaje no orientado a objetos debe hacer manualmente:

• expresibilidad: el programador del lenguaje no orientado a objetos debe hacer corresponder las operaciones orientadas a objetos, como llamadas de métodos o declaración de subclases, a operaciones explícitas que a veces son muy complicadas.
• conveniencia: el programador debe atravesar manualmente la jerarquía de clases cuando llama métodos o pasa argumentos. Si la jerarquía de clases cambia, entonces el programador debe evaluar manualmente las travesías.
• protección contra errores: el programador debe asegurar que todos los métodos sean incluidos en un método despachador o una estructura de despacho. El programador debe inicializar un objeto nuevo según su clase, y debe evitar el acceso a atributos internos de otras clases.
• mantenimiento: si se hacen cambios a las declaraciones de los objetos, el programador debe determinar los efectos secundarios sobre el código, modificándolo de acuerdo. Un lenguaje orientado a objetos provee y obliga la modularidad de clases que previene que los cambios se propaguen a través del programa entero. El programador del lenguaje no orientado a objetos requiere disciplina para lograr una modularidad similar sin la ayuda del lenguaje.

Implementación de los Conceptos Orientados a Objetos

Implementar un diseño orientado a objetos en un lenguaje no orientado a objetos requiere seguir básicamente los mismos pasos que para un lenguaje orientado a objetos. El programador del lenguaje no orientado a objetos debe hacer corresponder los conceptos orientados a objetos en el lenguaje blanco, mientras que el compilador en el lenguaje orientado a objetos hace esto de forma automática.

Los pasos a seguir para implementar el diseño son los siguientes:
1) Traducir las clases en estructuras de datos
2) Pasar argumentos a los métodos
3) Crear objetos
4) Implementar herencia
5) Implementar resolución de métodos
6) Implementar asociaciones
7) Encapsulamiento

Implementaciones en Diferentes Lenguajes

• Implementación en C: Se pierde la verificación de tipos con los tipos débiles de C, aunque se da mayor flexibilidad para implementar varios conceptos orientados a objetos importantes, gracias a los mecanismos como apuntadores y asignación de memoria dinámica. Es bastante sencillo implementar clases, instancias, herencia sencilla, y resolución de métodos durante la ejecución, sin perder eficiencia. (En realidad, este es el enfoque tomado por varios lenguajes orientados a objetos para generar código en C, como Objetive-C, Eiffel, y C++.)

• Implementación en Ada: Ada apoya la abstracción de datos pero no la herencia, por lo cual no puede ser considerada un lenguaje orientado a objetos. El obstáculo principal para la implementación de los conceptos de orientación a objetos es que Ada contiene tipos rígido y no incluye apuntadores para procedimientos. Aunque más difícil de traducir que en C, Ada ofrece excelentes facilidades para encapsulamiento, haciendo posible la construcción de grandes sistemas.

• Implementación en Fortran: La falta de estructura modernas de datos y la falta de asignación dinámica de memoria hacen la implementación muy difícil, aunque no imposible. Muchas de las estructuras apoyadas en C y Ada deben ser traducidas manualmente en Fortran.

• Implementación en Pascal**:** Pascal tiene limitaciones como Ada, incluyendo tipos rígidos y la falta de apuntadores de variables y funciones.

• Implementación en Lisp**:** Lisp es un lenguaje muy versátil donde casi todo es posible, además de ser la base para lenguajes como CLOS o Flavors.

4.4.1 Traducir clases en estructura de datos

Normalmente se implementaría cada clase como un bloque contiguo de atributos, o sea un *récord*. Cada atributo de la clase pasa a ser un elemento del récord, conteniendo un tipo, que puede ser primitivo, como *entero, real* o *carácter,* o puede ser una estructura, como otro récord.

Un objeto tiene estado e identidad y está sujeto a efectos secundarios. Para evitar estos efectos secundarios, una variable que identifica a un objeto debe ser implementada como una referencia, sin copiarse los valores de los atributos del objeto. Una referencia puede ser implementada como una dirección en memoria o como un índice de arreglo, pero siempre debe permitir su acceso compartido.

Cada clase se convierte en C en un *struct*, donde cada atributo pasa a ser un campo dentro del *struct* (*Largo* es un tipo en C, que puede ser definido usando un *typedef*):

```
struct Ventana
{
        Largo xmin;
        Largo ymin;
        Largo xmax;
        Largo ymax;
};
```

En C, una referencia a un objeto puede ser representada por un apuntador al récord del objeto:

```
typedef Ventana * ventana;
Largo x1 = ventana->xmin;
```

Un objeto puede ser asignado estáticamente, automáticamente (en la pila), o dinámicamente (en el heap).

4.4.2 Pasar argumentos a los métodos

Cada método tiene por lo menos un argumento, el argumento implícito *self*. En un lenguaje no orientado a objetos, el argumento debe hacerse explícito. Los métodos también pueden tener, como argumentos, simples valores u otros objetos. Al pasar un objeto como argumento a un método, la referencia del objeto debe ser pasada si el valor del objeto debe ser actualizado dentro del método. Si el método es una simple consulta a un objeto sin modificarlo, un mecanismo de llamada por valor puede ser usada si el lenguaje lo permite.

En C, un objeto debe ser siempre pasado por un apuntador. Aunque C permite a las estructuras ser pasadas por valor, pasarlas por un apuntador es por lo general más eficiente y provee uniformidad de acceso para operaciones de consulta y actualización.

```
Ventana__agregar_a_selecciones(self,forma)
        struct Ventana * self;
        struct Forma * forma;
```

4.4.4 Crear Objetos

Un objeto puede ser asignado estaticamente, automáticamente (en la pila), o dinámicamente (en el heap):

• Objetos asignados estáticamente deben ser implementados como variables globales asignadas por el compilador. Su vida es la duración del programa, y pueden ser útiles para objetos o constantes al nivel del sistema, aunque abusar de esto rompe con la modularidad del sistema.
• La mayoría de los objetos temporales e intermedios van a ser implementados como variables basadas en pilas (variables *automáticas* en C). La ventaja de este mecanismo, es que son asignadas y desasignadas automáticamente. El programador debe asegurar que no existan referencias a objetos basados en pilas luego de salirse del bloque de declaraciones. Este mecanismo no es apropiado para variables que tengan mayor vida que el procedimiento que las ha creado.

• Objetos asignados dinámicamente son necesarios cuando su número no es conocido durante la compilación. Un objeto general puede ser implementado como una estructura de datos asignada durante la ejecución de un heap. Almacenamiento para objetos asignados dinámicamente se llama de forma explícita por un operador como *malloc*, y explícitamente desasignado. Cuando un objeto ya no se necesita, debe ser desasignado con la función *free*.

```
struct Ventana * crear_ventana(xmin,ymin,ancho,largo)
        Largo xmin,ymin,ancho,largo;
{
        struct Ventana * ventana;
        ventana = (struct Ventana *) malloc (sizeof (struct Ventana));
        ventana->xmin = xmin;
        ventana->ymin = ymin;
        ventana->ancho = xmin + ancho;
        ventana->largo = ymin + largo;
        return ventana;
}
```

Objetos globales pueden ser declarados como variables *struct* de nivel alto, e inicializadas durante la compilación:

```
struct Ventana ventana_principal = {0.0,0.0,8.5,11.0};
```

4.4.4 Implementar herencia

Hay varias formas de implementar estructuras de datos para herencia en lenguajes no orientados a objetos:

• evitar la herencia: Muchas aplicaciones no requieren herencia. Las clases que no requieren herencia pueden ser implementadas por simples récords.
• aplastar la jerarquía de clases: Usar herencia durante el diseño, y expandir luego cada clase como una estructura de clases independiente durante la implementación. Cada operación heredada debe ser reimplementada como un método aparte en cada clase concreta. Aplastar la jerarquía introduce duplicación, pero el uso de, e.g. macros en C, puede ayudar. Una técnica útil es agrupar algunos atributos heredados en un tipo de récord e integrarlo en cada clase concreta para reducir el número de líneas duplicadas en cada declaración.
• dividir objetos: en lugar de heredar atributos comunes de una superclase, un grupo de atributos puede ser sacado de la superclase e implementado como un objeto separado. Agrupando atributos en tipos separados permite que un solo método sea escrito para manipularlo. Las subclases deben delegar las operaciones al objeto referenciado.

Estos enfoques impiden la implementación de herencia. Cuando la herencia es realmentete necesaria, la mejor implementación depende del lenguaje y la aplicación.

En C, el primer campo de cada *struct* tendría un apuntador a un objeto descriptor de la clase compartida por todas las instancias directas de la clase. El objeto descriptor de la clase es un

struct conteniendo los atributos de la clase, incluyendo el nombre de la clase, que es opcional, y los métodos para la clase.

La clase *Forma* es una clase abstracta, con clases concretas *Caja* y *Círculo*. Las declaraciones para las clases *Forma, Caja* y *Círculo,* son las siguientes:

```
struct Forma
{
        struct ClaseForma * clase;
        Largo x;
        Largo y;
}
```

```
struct Caja
{
        struct ClaseCaja * caja;
        Largo x;
        Largo y;
        Largo ancho;
        Largo alto;
}
```

```
struct Círculo
{
        struct ClaseCírculo * círculo;
        Largo x;
        Largo y;
        Largo radio;
};
```

Un apuntador a una estructura *Caja* o *Círculo* puede ser pasada a una función en C esperando un apuntador de tipo *Forma*, ya que la primera parte de las estructuras *Caja* y *Círculo* son idénticas a la estructura *Forma*. Por ejemplo, un apuntador a *Caja* es interpretado como un apuntador a *Forma* en la siguiente llamada:

```
struct Caja * caja;
struct Ventana * ventana;
Ventana__agregar_a_selecciones (ventana,caja);
```

El primer campo de cada estructura es un apuntador al descriptor de clases para cada instancia de objeto. Este campo es solo necesario si se hace resolución de métodos durante la ejecución.

La herencia múltiple no puede ser implementada usando este enfoque.

4.4.5 Implementar Resolución de Métodos

En un lenguaje orientado a objetos, una sola operación puede ser implementada por muchos métodos (*polimorfismo)*. La resolución de métodos puede ser manejada de las siguientes formas:

• evitar resolución de métodos: Si cada operación esta definida solamente una vez y no está sobrescrita, entonces no hay polimorfismo, y no hay necesidad para una resolución de métodos dinámica. Los métodos aún pueden ser heredados, pero todas las subclases comparten los métodos de la superclase, sin poder sobrescribirlos.
• resolver métodos durante la compilación: Si la clase de cada objeto es conocida durante la compilación, entonces el método correcto puede ser determinado y llamado directamente, evitando la necesidad para la resolución de métodos dinámica.
• resolver métodos durante la ejecución: Si se desea abstraer una operación de unos objetos de clases, entonces es necesaria la resolución dinámica de métodos. Se debe examinar la clase del objeto para determinar el método correcto. Un lenguaje orientado a objetos hace de forma automática este examen.

Cualquier resolución de método hecha durante la compilación puede ser implementada como una llamada directa de una función en C. Muchas operaciones están implementadas solamente una vez como métodos y nunca están sobrescritas, por lo cual no necesitan resolución de métodos durante la ejecución. Por ejemplo, ningún método de *Ventana* está sobreescrito. El enfoque más general es definir un objeto descriptor de clase para cada clase conteniendo un apuntador a la función del método para cada operación visible de la clase, incluyendo operaciones heredadas. Cada descriptor de clase es una *struct* en C conteniendo todas las operaciones definidas en la clase o heredadas de una superclase. El siguiente código muestra la declaración de los descriptores de clase para *Artículo*, *Forma*, *Caja*, y *Círculo*. (Se agregó la palabra *Clase* al nombre de la clase para el *struct* descriptor de la clase.)

```
struct ArtículoClase
{
        char* nombre_clase;
        void (* cortar)();
        void (* mover)();
        Boolean (* seleccionar)();
        void (* desagrupar)();
}
```

```
struct FormaClase
{
        char* nombre_clase;
        void (* cortar)();
        void (* mover)();
        Boolean (* seleccionar)();
        void (* desagrupar)();
        void (* escribir)();
}
```

```
struct CajaClase
{
        char* nombre_clase;
        void (* cortar)();
        void (* mover)();
        Boolean (* seleccionar)();
        void (* desagrupar)();
        void (* escribir)();
}
```

```
struct CírculoClase
{
        char* nombre_clase;
        void (* cortar)();
        void (* mover)();
        Boolean (* seleccionar)();
        void (* desagrupar)();
        void (* escribir)();
};
```

El descriptor de clase *struct* define los nombres de las operaciones visibles a la clase, pero aún se debe inicializar el objeto descriptor de clase para cada clase. Cada descriptor es una variable global. Se debe inicializar cada campo del descriptor de clase con el nombre de la función del método en C, que puede ser definido o heredado. Por ejemplo, la clase *Caja* hereda la operación *mover* de la clase *Forma* pero sobrescribe la operación *seleccionar* y *escribir* con sus propios métodos:

```
struct CajaClase CajaClase =
{
        "Caja",
        Forma__cortar,
        Forma__mover,
        Caja__seleccionar,
        Forma__desagrupar,
        Caja__escribir
};
```

```
struct CírculoClase
{
        "Círculo",
        Forma__cortar,
        Forma__mover,
        Círculo__seleccionar,
        Forma__desagrupar,
        Círculo__escribir
};
```

Se debe notar que los objetos descriptores de clase son necesarios sólo para las clases concretas, siendo innecesarios para las clases abstractas, como *Forma*. El único uso de los objetos descriptores de clase es guardar los métodos y variables de clase de los objetos.

Cuando un objeto es creado, la dirección del objeto descriptor de la clase es guardado en el campo *clase* en el récord del objeto. La información sobre la clase de un objeto, incluyendo su nombre, atributos, y métodos, pueden ser obtenido durante la ejecución, leyendo campo *clase* del objeto. El campo clase de cada objeto debe ser inicializado con un apuntador al descriptor de clase:

```
struct Círculo * crear_círculo(x0,y0,radio0)
        Largo x0,y0,radio0;
{
        struct Círculo * círculo;
        círculo = (struct Círculo *) malloc (sizeof (struct Círculo));
        círculo->clase = &CírculoClase;
        círculo->x = x0;
        círculo->y = y0;
        círculo->radio = radio0;
        return círculo;
}
```

Si una operación debe ser resuelta durante la ejecución, el objeto descriptor de clase es usado para determinar la función correcta en C. El objeto descriptor de clase es obtenido del objeto e indexado por el nombre de la operación. Por ejemplo, para llamar la operación *seleccionar* de una *Forma* no conocida requiere el siguiente código:

```
struct Forma * forma;
Largo x,y;
Boolean estado;
estado = (*forma->clase->seleccionar) (forma,x,y);
```

Se debe incluir el objeto blanco dos veces, una vez para encontrar el método, y luego como primer argumento al método. La resolución dinámica de método requiere dos accesos más de memoria. Comparado al costo de llamar una función, el costo adicional es realmente insignificante.

4.4.6 Implementar Asociaciones

Implementar asociaciones en un lenguaje no orientado a objetos presenta las mismas dos posibilidades que un lenguaje orientado a objetos: hacer un mapa a apuntadores o implementar las asociaciones directamente como un objeto contenedor.

En C, una asociación binaria es implementada usualmente como un campo en cada objeto asociado, conteniendo un apuntador al objeto relacionado o a un arreglo de objetos relacionados. Por ejemplo, la asociación "muchos-uno" entre *Artículo* y *Grupo* se puede implementar como:

```
struct Artículo
{
```

```
        struct ArtículoClase * clase;
        struct Grupo * grupo;
};
```

```
struct Grupo
{
        struct GrupoClase * clase;
        int artículo_cuenta;
        struct Artículo ** artículos;
};
```

Otras estructuras de datos, como listas ligadas, pueden también ser usadas para guardar conjunto de objetos. En este ejemplo, un grupo es creado de un conjunto de artículos seleccionados. La memoria para los apuntadores *artículos* puede ser asignada de forma conjunta, ya que el número de artículos en un grupo no cambia. Si fuera posible añadir un nuevo objeto a un grupo, entonces ambos apuntadores deben ser actualizados:

```
Grupo__agregar_artículo (self,artículo)
        struct Grupo * self;
{       struct Artículo * artículo;

        artículo->grupo = self;
        self->artículos = (struct Artículo **) realloc (self->artículo,
                ++self->artículo_cuenta * sizeof (struct Artículo *));
        self->artículos[self->artículo_cuenta-1] = artículo;
}
```

4.4.7 Encapsulamiento

Encapsulamiento de la representación de los datos y la implementación de los métodos es uno de los temas más importantes en la programación orientada a objetos. Parte del encapsulamiento se pierde cuando el programador debe traducir manualmente los conceptos orientados a objetos en lenguajes no orientados a objetos.

C tiene una reputación de estimular un estilo de programación no muy rígido que es malo para el encapsulamiento. De todas formas se pueden tomar los siguientes pasos para mejorar el encapsulamiento:

• evitar el uso de variables globales
• empacar los métodos de cada clase en un archivo separado. Incluir sólo la declaración para las clases cuya estructura interna es necesaria, como clases ancestras de la clase actual. No se debe accesar los campos de objetos de clases diferentes, se debe en lugar llamar el método de acceso.
• tratar los objetos de otras clases como tipo "void *".

4.5 Bases de Datos Orientadas a Objetos

Se pueden definir tres tipos de necesidades para las bases de datos:

• manejo de datos: involucra estructuras de datos y operaciones simples, e.g. aplicaciones de negocios tradicionales (RDBMS con estructuras simples como enteros y cadenas)
• manejo de objetos: involucra estructuras de datos más complejas, e.g. documentos, programas, diseños (ODMS - Object Data Management Systems)
• manejo de conocimiento: mantenimiento de bases de reglas complejas para sistemas de inferencia

Ejemplo: Una base de datos para manejo de documentos.
• Los autores de los documentos requieren capacidad para representar partes y versiones de los documentos.
• Un bibliotecario requiere capacidad de manejo de datos más tradicional, ejecutando transacciones para verificar documentos o consultar la base de datos para encontrar documentos que deben ser regresados a la biblioteca.
• Un sistema experto usado por los abogados puede aumentar la base de datos de los documentos con reglas sobre la estructura de los componentes legales, o las dependencias lógicas entre documentos.
• Una compañía puede tener los tres tipos de usuarios, y requeriría un sistema de base de datos que pueda proveer todas estas capacidades.

Es difícil construir un sistema de base de datos que satisfaga los tres aspectos, en especial por el rendimiento necesario.

4.5.1 Modelo de Datos

Hay dos tipos de información en la base de datos: *datos* y *esquemas*.
dato: dato simple como *enteros* y *cadenas*, o más complejos como *récords*.
esquema: metadato describiendo la estructura del dato.

modelo de datos: Tipos diferentes de bases de datos tienen diferentes restricciones en el tipo de datos y los esquemas que son permitidos.

Los diferentes modelo están convergiendo en un solo modelo de datos con aspectos combinados, en particular incorporando conceptos de los lenguajes orientados a objetos y extensiones al modelo relacional.

4.5.1.1 Modelo Relacional

Existe solo una estructura de datos en el modelo relacional: una tabla con filas y columnas, conteniendo datos de tipos específicos, como enteros o cadenas. El lenguaje de consultas se basa en operaciones simples para tablas, donde la simplicidad del modelo es un beneficio pero también es una limitación.

4.5.1.2 Modelo Relacional Extendido

Se extiende el modelo relacional con procedimientos, objetos, versiones, y otras nuevas capabilidades. No hay un solo modelo relacional extendido, hay más bien una variedad de ellos,

aunque todos comparten las tablas y consultas básicas del modelo relacional. Todos incorporan algún concepto de "objetos" y todos tienen la habilidad de guardar procedimientos al igual que datos en la base de datos.

4.5.1.3 Modelo Orientado a Objetos

En los DBMS orientados a objetos (OODBMS), varía el tipo de encapsulamiento de los datos y los procedimientos en el objeto. El termino "orientado a objetos" varía también según los autores.

4.5.1.4 Modelos Funcionales

La base del modelo funcional son los *objetos* y las *funciones*. Se usan lenguajes de acceso de datos basados en notación de funciones matemáticas con un lenguaje de consultas funcional declarativo, análogo a los lenguajes de consulta relacionales. Relaciones entre objetos, atributos de objetos, y procedimientos asociados con objetos están todos representados por funciones.

4.5.1.5 Modelos Semánticos

El modelo semántico integra conceptos de varios modelos, incluyendo la incorporación de objetos y relaciones entre objetos, incluyendo herencia. Originalmente, este modelo estaba orientado hacia aplicaciones en Inteligencia Artificial.

4.5.2 Arquitectura de Base de Datos

Algunos sistemas no siguen un modelo de dato particular, donde el manejo de datos es combinado con lenguajes de programación.

4.5.2.1 Sistema de Base de Datos Extendido

Estos sistemas presentan una nueva funcionalidad a través de los lenguajes de consultas extendidos que incorporan procedimientos y otros aspectos. Se incluye un lenguaje de programación para la aplicación, y un lenguaje extendido de consultas para la base de datos. Los dos lenguajes se pueden llamar entre si, aunque son diferentes en su estructura y funcionamiento.

4.5.2.2 Lenguajes de Programación de Bases de Datos

Estos sistemas de bases de datos extienden los lenguajes de programación existentes, como C++, para proveer *persistencia*, *control de concurrencia*, y otras capabilidades [Bancilhon y Buneman 1990].

4.5.2.3 Manejadores de Objetos

Extienden sistemas de archivos o memoria virtual, generalmente de un modelo de datos limitado. A nivel físico se provee un repositorio de objetos persistente, aunque sin incluir un lenguaje de consultas.

4.5.2.4 Generadores de Sistemas de Bases de Datos

Dejan al implementador construir un DBMS ajustado a la necesidades particulares. No tienen un modelo de datos propio, puediéndose ajustar según la necesidad.

4.5.3 Sistemas de Bases de Datos

4.5.3.1 Lenguajes de Programación de Bases de Datos Orientadas a Objetos

Los ODM (Object Data Management) basados en el modelo de datos orientado a objetos, conocidos como OODBMS (Object Oriented Data Base Management Systems), unifican dos tecnologías: el manejo de base de datos y la programación orientada a objetos. Los lenguajes de programación orientados a objetos son por lo general bastante expresivos pero carecen de *persistencia* de datos, mientras que los DBMS convencionales tienen persistencia de datos pero carecen de expresibilidad. Los OODBMS tratan de proveer persistencia y expresibilidad, manejando programas grandes que operan en grandes almacenamiento de datos. Las aplicaciones se escriben como una extensión a lenguajes de programación existentes, y el lenguaje y su implementación (compilador, preprocesador, ambiente de ejecución) han sido extendidos para incorporar la funcionalidad de base de datos.

Para aplicaciones convencionales que combinan programas con acceso a los DBMS, el desarrollador de una aplicación usa técnicas procedurales, como descomposición funcional jerárquica o diagramas de flujo de datos. Una técnica como Entidad-Relacion puede ser usada para el diseño de la base de datos. El diseño de la base de datos puede ser convertido a código SQL el cual puede ser integrado con lenguajes como C, Pascal, o Cobol. Existen dos problemas fundamentales con este enfoque:
• Diferentes paradigmas son usados para diseñar varias partes del sistema.
• Los vehículos de implementación no se integran bien.
Lo ideal sería aplicar de forma uniforme el modelo orientado a objetos a todo el sistema, incluyendo el diseño, el lenguaje de implementación, y la base de datos.

Existen dos tipos de consultas en una base de datos:
• orientadas a conjuntos, y
• de acceso.
Los DBMS relacionales son diseñados para ejecutar operaciones paralelas en grandes conjuntos de datos. En contraste, los lenguajes de programación orientadas a objetos son eficientes para accesar rápidamente de un objeto a otro atravesando apuntadores. Un DBMS relacional ejecuta el acceso usando *joins*, que son varias ordenes de magnitud más lentas que travesías con apuntadores. Un OODBMS debe ejecutar de forma eficiente ambos tipos de consultas. Un aspecto importante del OODBMS es la suposición implícita que el sistema está orientado hacia operaciones en objetos individuales y el programador puede esperar que estos se ejecuten bien.

Ejemplos:
• GBase [Object Database 1990]
• GemStone [Bretl et al. 1988]
• ITASCA [Itasca 1990]

• O_2 [Bancilhon et al. 1988]
• Objectivity/DB [Objectivity 1990]
• ObjectStore [Object Design 1990]
• ONTOS [Ontologic 1989]
• ORION [Kim et al. 1988]
• STATICE [Weinreb 1988]
• VERSANT [Versant 1990]
• ZEITGEIST [Ford et al. 1988]

La mayoría de estos lenguajes de programación de base de datos orientados a objetos han sido integrados con C++, o un derivado. Algunos se han concentrado en Smalltalk o derivado de LISP. Por ejemplo, GemStone incorpora un lenguaje similar a Smalltalk, con interfaces procedurales a C y Pascal; ONTOS tiene una estructura orientada a objetos accesible para consultas SQL y C++; mientras que ORION está escrito en Common Lisp, con sintaxis similar a Flavors y LOOPS.

Algunos de los OODBMS que se parecen más a *sistemas de bases de datos extendidos* que a *lenguajes de programación de base de datos*.

Ejemplo:
• VBase [Ontologic 1986]
• VISION [Innovative Systems 1988]

Los lenguajes de programación persistentes están muy relacionados a los lenguajes de programación de las bases de datos, aunque no estan diseñados para proveer una funcionalidad completa sobre bases de datos, ni un acceso a datos persistentes.

Ejemplo:
• PS-Algol [Atkinson et al. 1983]
• Trellis [Schaeffert et al. 1986]

4.5.3.2 Sistemas de Base de Datos Relacionales Extendidos

El modelo relacional extendido incorpora un lenguaje de consultas extendido con procedimientos, identidad de objetos, y jerarquía de tipo.

Ejemplos:
• ALGRES [Ceri et al. 1990]
• POSTGRES [Stonebraker y Rowe 1986]
• Starbust [Schwarz et al. 1986]

Ejemplos de sistemas con extensiones más modestas:
• INGRES [Ingres 1990]
• SYBASE [Sybase 1990]

4.5.3.3 Sistemas de Base de Datos Funcionales Extendidos

Los atributos, relaciones, y procedimientos todos son considerados funciones en el modelo, y las relaciones entre dos o más tipos de objetos se representan como funciones de múltiples argumentos.

Ejemplos:
• ADAPLEX [Smith 1983]
• IRIS [Fishman et al. 1987]
• PROBE [Dayal and Smith 1986]

4.5.3.4 Sistemas de Base de Datos Semánticos Extendidos

Ejemplos:
• CACTIS [Hudson and King 1988]
• SIM [Fritchman et al. 1990]

4.5.3.5 Generadores de Sistemas de Base de Datos

El modelo de datos conceptual e interno son definidos según la necesidad. El modelo conceptual especifica las capacidades de consulta y representación, como tablas, llaves, relaciones y sintaxis, mientras que el modelo interno especifica los métodos de acceso y como se hace el mapa del modelo conceptual.

Ejemplo:
• EXODUS [Carey et al. 1986]
• GENESIS [Batory et al. 1986]

4.5.3.6 Administradores de Objetos

Administradores de Objetos Persistentes son usados como manejadores para guardar datos, dando acceso concurrente a múltiples usuarios, pero sin tener un lenguaje de programación o consultas.

Ejemplos:
• Camelot [Spector et al. 1988]
• LOOM [Krasner 1986]
• Mneme [Moss and Sinofsky 1988]
• ObServer [Zdonik 1987]
• POMS [Cockshot et al. 1984]

4.5.4 Objetos

Todos los modelos aquí presentados incluyen de alguna forma el concepto de objeto. Se consideran esenciales dos aspectos de objetos:

• agrupación de objetos: objetos que sirven para agrupar datos
• identidad de objetos: objetos que pueden tener una identidad única independiente de los valores que contienen.
Un DBMS que provee agrupación e identidad de objetos se considera *estructuralmente* orientado a objetos.

4.5.4.1 Identificadores de Objetos

La idea de identificadores de objetos únicos aparece en muchos modelos, los cuales reconocen las limitaciones de las llaves primarias y los modelos basados en valores, como los relacionales. Las llaves primarias requieren que el programador genere sus propios identificadores únicos, en particular cuando existen sistemas distribuidos.

4.5.4.2 Llaves de Objetos

Algunos modelos dejan a los objetos tener, además de identificadores de objetos, llaves de objetos, que son nombres equivalentes a las llaves primarias en el modelo relacional. El beneficio es poder llamar a los objetos por llave, con sintaxis simplificada, donde el DBMS puede verificar automáticamente la integridad de que las llaves sean únicas.

4.5.4.3 Atributos

Se pueden distinguir dos tipos de atributos: *simples* y *complejos*.

Atributos Simples

Los valores simples pueden ser:
• *literale,* como entero, cadena, float, o también
• *objetos* que definen entidades del mundo real como documentos o personas.
En algunos sistemas orientados a objetos, los valores literales también pueden ser representados como objetos, mientras que en el modelo relacional los objetos son representados como récords, y los valores literales son guardados como atributos de récords. Por lo general, los usuarios pueden definir sus propios tipos de valores literales.

Como los atributos en récords relacionales, los atributos de objetos tienen nombres, y se les puede referir con notación de punto:

Ejemplo: El atributo *revisión* de un documento *d* se puede referir como *d.revision.*

Todo tipo de atributos, sin importar su tamaño físico, e.g. BLOB (Binary Large OBjetcts) o datos de multi-media, se consideran atributos *simples*.

Atributos Complejos

Hay tres tipos de atributos *complejos*: *referencias*, *colecciones* y *procedimientos*.

Atributos de Referencia: Los atributos de referencia, o asociaciones, se usan para representar relaciones entre objetos. Sus valores son referencias a objetos. Son análogos a los apuntadores en los lenguajes de programación, o a las llaves externas en los sistemas relacionales, aunque hay una diferencia importante:

• Los atributos de referencias no pueden estar corruptos, aunque los apuntadores sí pueden. Los valores de referencia se invalidan automaticamente cuando la referencia al objetos es removida.

• Los atributos de referencia no se asocian con valores visibles del usuario, mientras que las llaves externas sí pueden. Todos los valores en el objeto referido pueden cambiarse, y los atributos de referencia todavía apuntarían al mismo objeto.

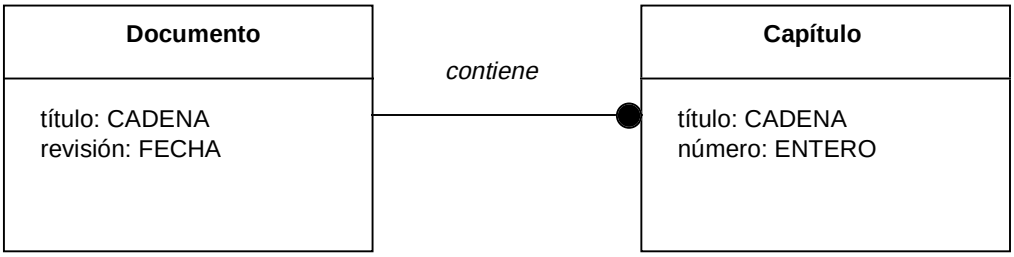


Figura 4.3. Diagrama de clases para *Documento* y *Capítulo*.

Ejemplo: En lugar de usar una llave externa como en el modelo relacional, se puede definir para el diagrama de clases que se muestra en la Figura 4.3, un atributo de referencia a *Documento* para *Capítulo* en la base de datos indicando a qué documento el capítulo pertenece:

```
Capítulo: {
        título:         CADENA;
        número:         ENTERO;
        doc:            Documento; }
```

Atributos de Colección: Los atributos de colección se usan para listas, conjuntos, o arreglos de valores, que pueden estar ordenadas explícitamente, en el caso de listas o arreglos. Las colecciones pueden incluir valores de atributos simples y también referencias.

Ejemplo: Se puede definir un objeto *Documento* con dos valores de atributos que sean de colección:

```
Documento: {
        título:         CADENA;
        revisión:       FECHA;
        claves: CONJUNTO[CADENA];
        caps:           LISTA[Capítulo]; }
```

La primera colección, *claves*, es un conjunto de cadenas de claves asociadas con el documento, que pueden ser usada para buscar un documento según algún tema. La segunda colección, *caps*, es una lista de objetos capítulos para este documento.

Se debe notar que la primera forma normal del modelo relacional no permite atributos que sean valores de colección, al contrario de la mayoría de ODMS, donde las listas de objetos y los conjuntos son objetos de primera clase.

CONJUNTO[CADENA] y *LISTA[Capítulo]* son ejemplos de *tipos parametrizados*, que significa que las construcciones *CONJUNTO* y *LISTA* toman tipos como parámetros, definiendo nuevos tipos dependiendo de los valores de los parámetros. Muchos lenguajes de programación usan tipos parametrizados, como *ARREGLOS*, tomando el tamaño y tipo de los elemento como parámetro.

Atributos Derivados: Los atributos derivados proveen independencia de datos lógicos. Los atributos pueden ser definidos de forma procedural, si un procedimiento es especificado para ser ejecutado cuando el valor esdel dato es leído o asignado.

Ejemplo: Si adoptamos un mecanismo para seguir el número de revisión de unos documentos, definiendo un atributo de *Documento* llamado *creación*, que es un entero con la fecha de creación de los archivos, se podría entoces redefinir el atributo *revisión* de forma procedural en término de *creación*, para que las aplicaciones no tengan que ser reescritas. Se define *creación* y *revisión* usando la siguiente sintaxis:

```
Documento: {
        ...
        creación:       ENTERO;
        revisión:       FECHA PROC () = segsAFecha (creación); }
```

Programas anteriores que llaman *d.revisión* de un documento *d* continuarían funcionando correctamente, llamando el nuevo procedimiento *revisión* en lugar del viejo atributo.

Atributos Virtuales

Se definen como atributos virtuales, atributos que de forma transparente al programador, pueden ser derivados o guardados explicitamente (e.g. POSTGRES, O_2). Los atributos virtuales no puede ser usados en lenguajes donde la sintaxis usada para traer atributos es diferente de la usada para invocar procedimientos, métodos, o funciones (e.g. en C++ la sintaxis es diferente para accesar atributos o métodos. *d.revisión()* se usa para llamar metodos).

4.5.5 Asociaciones

La representación de asociaciones difiere en losdiferentes modelos de datos.

4.5.5.1 Asociaciones Binarias

Las asociaciones son representadas en el modelo de datos orientado a objetos usando atributos complejos. La representacion depende de la multiplicidad y el grado de la relación.

Posibles representaciones binarias:
• asociaciones "1-1": entre dos clases de objetos A y B añadiendo un atributo de referencia a cada clase.
• asociaciones "1-muchos": entre dos clases de objetos A y B añadiendo un atributo de referencia a A, y un atributo de referencia de conjunto a B ("muchos").
• asociaciones "muchos-muchos": entre dos clases de objetos A y B añadiendo un atributo de referencia de conjunto a cada clase.

Ejemplo: Diagrama de clases para *Documento*, incluyendo *Capítulo*, *Persona*, y *Autor*, se muestra en la Figura 4.4.

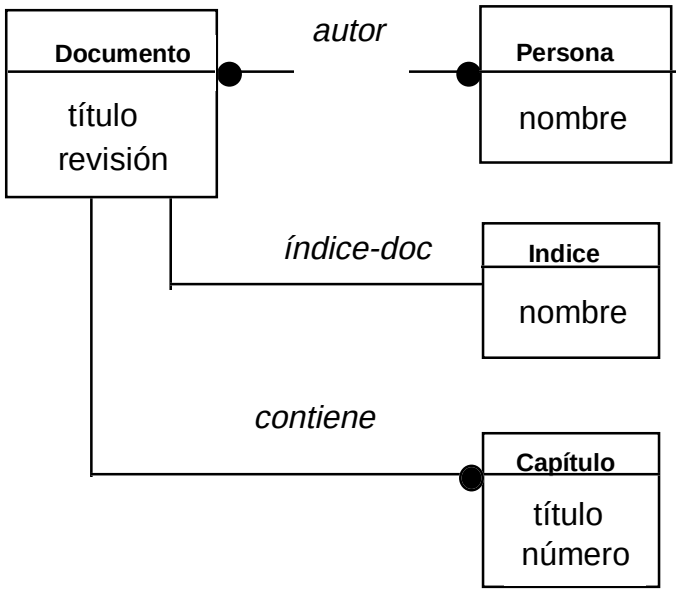


Figura 4.4. Diagrama de clases para un *Documento,* incluyendo *Capítulo*, *Persona*, y *Autor*.

La representación del diagrama se puede hacer de la siguiente forma, donde "<->" representa *atributos inversos*:

```
Documento: {
        título:         CADENA;
        revisión:       FECHA;
        caps:           LISTA[Capítulo] <-> doc;
        autores:        LISTA[Persona] <-> pubs;
        índice:         Indice <-> para; }
```

```
Capítulo: {
        título:         CADENA;
        número:         ENTERO;
        doc:            Documento <-> caps; }
```

```
Indice: {
        entrada:    ENTERO;
        para:       Documento <-> índice; }
```

```
Persona: {
        nombre:     CADENA;
        pubs:       LISTA[Documento] <-> autores; }
```

Atributos Inversos: describen relaciones análogas, representando exactamente la misma información.

Ejemplo: *doc***:** *Documento* <-> *caps, caps***:** *LISTA[Capítulo]* <-> *doc*.

Integridad Referencial: pares de atributos inversos siempre deben corresponder, debiendo actualizarse las referencias si un objeto es removido o una relación es cambiada.

Ejemplo: si se añade otro capítulo al documento, el ODMS debe preocuparse de actualizar la lista de capítulos del documento. Si un capítulo es movido a otro documento, los dos atributos deben ser actualizados. Si un documento es removido, todas las referencias al documento deben ser nulas.

Hay varios niveles de integridad referencial:

• Sin verificación de integridad: El sistema puede no verificar referencias. En este caso la referencia puede ser a un tipo incorrecto de objeto, o a un objeto que no existe. En tal caso la aplicación debe matener apropiadamente las relaciones.

• Validación de referencia: El sistema puede asegurar que las referencias sean a objetos que existan, y que sean del tipo correcto. Hay dos formas de hacerlo:

a) El sistema puede remover objetos automaticamente cuando no son accesibles por el usuario (e.g. GemStone). En tal caso, el sistema no dejar remover de forma explícita los objetos, utilizando en lugar un recolector de basura para determinar que objetos deben ser removidos automaticamente (e.g. Smalltalk y Lisp).

b) El sistema puede requerir que los objetos se remuevan explicitamente cuando ya no son usados, pero que también se puedn detectar automaticamente referencias no válidas. Si los identificadores de objeto no son reusados, las referencias a los objetos removidos pueden ser detectadas luego. (En la mayoría de los lenguajes de programación, las direcciones en memoria son usadas como referencias, sin existen referencias inversas, por lo cual no se puede proveer validación de referencia.)

• Integridad relacional: El sistema puede dejar que se modifiquen los objetos y las relaciones de forma explícita, pero que se mantiengan de forma automática las relaciones vistas por los otros objetos. Los atributos inversos proveen integridad relacional, como en el modelo relacional. Las

dos alternativas anteriores no lo hacen, ya que requieren que las relaciones entre objetos sean representadas por atributos múltiples sin ser mantenidas por el DBMS.

La mayoría de de ODMS proveen integridad relacional (e.g. Objectivity/DB, ObjectStore, ONTOS, Cypress, Probe).

• Semántica de Referencia Adapatable: Los sistemas más sofisticados permiten al diseñador de base de datos especificar una semántica de integridad referencial particular para cada clase de objeto o relación.

Actualmente sólo existen prototipos limitados de semántica de referencia adaptable. La semántica de referencia adaptable podría ser codificada en métodos asociados con objetos, donde todas las actualizaciones a los objetos relacionados estarían garantizadas por la integridad referencial. Las restricciones de integridad referencial aparecen en el contexto de mantener múltiples versiones de los objetos, al igual que operaciones sencillas, como cambiar relaciones y remover objetos relacionados.

4.5.5.2 Asociaciones No-Binarias

Las asociaciones pueden ser de grado más alto que binario, como por ejemplo, relaciones ternarias.

Ejemplo: En la Figura 4.5 se muestra el diagrama de clases para *Documento*, conteniendo asociaciones ternarias y binarias, *prestado* y *referencias*, respectivamente

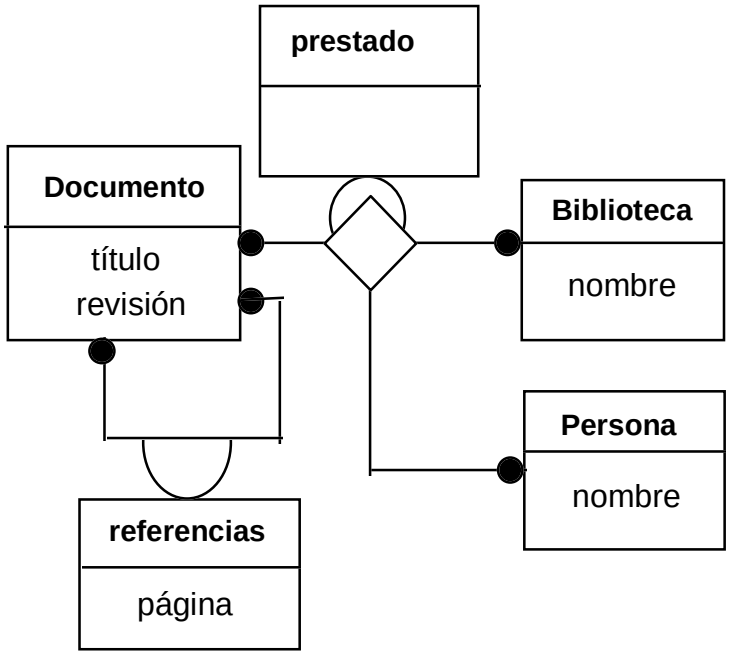


Figura 4.5. Diagrama de clases para *Documento* conteniendo asociaciones ternarias.

Es necesario crear nuevas clases de objetos para representar estas relaciones ternarias. En contraste, las relaciones binarias pueden ser representadas usando solo pares de atributos inversos:

```
Persona: {
        nombre:         CADENA;
        prestados:      LISTA[prestado] <-> quién; }
```

```
Biblioteca: {
        nombre:         CADENA;
        prestamos:      LISTA[prestado] <-> de; }
```

```
Documento: {
        ...
        refs:           LISTA[referencias] <-> de;
        refpor:         LISTA[referencias] <-> a;
        prestados:      LISTA[prestado] <-> libro; }
```

```
referencias: {
        de:             Documento <-> refs;
        a:              Documento <-> refpor;
        página:         ENTERO; }
```

```
prestado: {
        quién:          Persona <-> prestados;
        de:             Biblioteca <-> prestamos;
        libro:          Documento <-> prestados; }
```

4.5.5.3 Relaciones Derivadas

Las relaciones, como los atributos, pueden ser definidas de forma procedural en lugar de explicitamente. Las relaciones derivadas pueden ser definidas en término de atributos básicos. A diferencia de los atributos, las relaciones requieren más de un método de acceso, uno por cada uno de los atributos inversos involucrados en la relación.

Ejemplo: se podrían definir métodos para los atributos *doc* y *caps*, asociando documentos con capítulos.

4.5.5.4 Comparación con el Modelo Relacional

Las siguientes son algunas de las diferencias en la implementación de relaciones entre el modelo orientado a objetos y el modelo relacional:

• Las referencias son representadas usando llaves primarias en el modelo relacional en lugar de identificadores de objeto, aunque todos los modelos usan referencias para representar relaciones.

• Las tablas de relación son aproximadamente equivalentes a los objetos intermedios.

• Los modelo orientados a objetos y los modelos funcionales empacan en el objeto, como atributos o funciones, todas las relaciones en las cuales los objetos participan. En el modelo

relacional es necesario examinar el esquema completo para determinar cuales tablas tiene llaves externas involucrando tablas de entidades.
• Los modelos orientados a objetos permiten ordenar manualmente las relaciones involucrando listas.
• La sintaxis y semántica de acceso son bastante diferente.

4.5.5.5 Comparación con el Modelo Relacional Extendido

La implementacion de las relaciones en los modelos relacionales extendidos es muy parecida que en los modelos relacionales, con excepción de que los identificadores de objeto se pueden usar en lugar de las llaves para referenciar objetos (tablas de entidades).

4.5.5.6 Comparación Modelos Funcionales

Las relaciones en el modelo de datos funcional varían de la notación del modelo de datos orientado a objetos, pero tiene las mismas propiedades básicas.

Ejemplo Documento: para referirse al título de un documento al que el capítulo *C1* pertence, se puede usar *título(doc(C1))*. La función o su inverso puede aparecer en el lado izquierdo de la asignación, *doc(C1) = D1*. La implementación debe tratar con los mismos aspectos que el modelo orientado a objetos, en mantener representaciones inversas de una relación. En lugar de mantener atributos inversos, se mantiene funciones inversas (e.g. *doc(C1), caps(C1)*).

A diferencia de otros modelos, el modelo de datos funcional no requiere la introducción de objetos para representar relaciones ternarias, la relación puede ser representada directamente por funciones.

Ejemplo: *prestado* puede ser representado por *prestatario(D1,B1)* devolviendo las personas que tomaron prestado el documento *D1* de la biblioteca *B1*.

Si se prefiere, los objetos intermedios se pueden introducir para representar relaciones ternarias.

Ejemplo: para *D1.prestado.quién* puede ser *quién(prestado(D1))*

4.5.6 Objetos Compuestos

Agregación es la agrupación de partes en un todo. En los sistemas de base de datos el término ha sido usado para referirse a dos tipos de agrupaciones:
• *atributos:* pueden ser agregados para formar objetos. Por ejemplo, un documento consiste de un autor, fecha, y abstracto.
• *objetos:* pueden ser agregados para formar objetos *compuestos*. Por ejemplo, una sección de un documento está compuesta de un número de componentes u otras secciones. (Aquí se usará para referirse el segundo caso.)

4.5.6.1 Semántica de Agregación

La ventaja de definir relaciones de agregación y objetos compuestos es que el DBMS podría, por ejemplo, remover automáticamente los componentes de un agregado que ya haya sido removido.

4.5.6.2 Relación de Agregación

Los DBMS difieren en su definición de la relación de agregación. Tradicionalmente, la relacion *parte-de* era usada. Más recientemente, se ha dejado al usuario definir sus propia relación de agregación según el tipo de objetos siendo agrupados.

Ejemplo: *doc: PADRE Documento <-> caps*

Se puede especificar una semántica adicional, como *remover*, o *copiar*, por medio de palabras claves adicionales.

4.5.7 Procedimientos

En contraste a los DBMS tradicionales, los ODMS proveen un lenguaje de acceso a la base de datos que es *computacionalmente completo*, o sea que el lenguaje de base de datos puede ejecutar las mismas operaciones que un lenguaje de programacion. Además, el ODMS puede proveer la capacidad de asociar operaciones con objetos de base de datos y guardar procediminetos en la base de datos.

4.5.7.1 Datos Activos

Los procedimientos pueden estar asociados con objetos, o sus atributos, definiéndose de esta fomra *datos activos*. (Datos derivados son sólo un tipo de datos activos.)

Los siguientes son algunos tipos de datos activos:

Tipo de Acción	**Resultado de Acción**		
	invocación explícita, mismo proceso	invocación por predicado, mismo proceso	invocación por predicado, proceso independiente
valor de campo	dato derivado	restricción de integridad	n/a
actualización DB	procedimiento DB	regla (*trigger*)	producción
arbitrario	método	notificador	agente

Los casos de datos activos más importantes en la tabla son los que están en la diagonal:
• *datos derivados*: atributos de objetos o relaciones definidas de forma procedural, correspondiendo a *vistas* en la base de datos relacional.
• *reglas*: un par patrón-acción que puede ser activado por la actualización de la base de datos que hace que el predicado sea cierto. Las reglas pueden ejecutar la actualización de la base de datos, siendo ejecutadas por el mismo hilo de control como el proceso que las activa (*trigger*). Las restricciones de integridad son similares a las reglas, pero no ejecutan actualizaciones, simplemente devuelven una condicion de error. (Ej: POSTGRES)

• *agentes*: las acciones pueden ser iniciadas espontáneamente como reglas, pero ejecutadas como procesos independientes, en paralelo con el proceso activador. Las acciones de los agentes difieren de las reglas ya que están permitidas a ejecutar acciones arbitrarias además de la actualización de la base de datos.

4.5.7.2 Comparación con Modelos Funcionales

Es posible combinar los modelos funcionales y orientados a objetos:
• En el modelo funcional, las funciones asociadas con un objeto se pueden agrupar como operaciones del objeto.
• En el modelo orientado a objetos, los métodos pueden ser definidos como funciones, y métodos funcionales pueden ser definidos automáticamente por el DBMS para cada atributo del objeto, incluyendo atributos inversos usados para representar relaciones.

Estos aspectos presentan una mejoría sobre los modelos orientados a objetos puros ya que las funciones pueden proveer una notación uniforme, otro nivel de independencia de datos, y una mejor base que los métodos para definir un lenguaje global de consultas que provea acceso asociativo a objetos.

4.5.7.3 Comparación con Modelos Relacionales Extendidos

Otra forma de incorporar procedimientos en un modelo de datos, fuera de los enfoques funcionales y orientados a objetos, es extender el lenguaje de consultas relacional para que sea procedural (e.g. SYBASE), o dejar llamar procedimientos de un lenguaje de programación con el DBMS relacional (e.g. POSTGRES).

4.5.8 Tipos y Herencia

4.5.8.1 Tipos

En el modelo relacional, las tablas pueden considerarse tipos. Más precisamente, las definiciones de los atributos (columnas) de la tabla son la definición del tipo, y las filas son las instancias del tipo.

• *intensión*: estructura y comportamiento de los objetos de un tipo particular.
• *extensión*: conjunto de objetos con un tipo particular, clasificación.
• *representativo*: tipos son representados por objetos (metadatos). Define la intesión del objeto.

Ejemplo: si *Documento* es representativo del tipo documento, la operación *Documento.crear()* crearía un nuevo documento y devolvería una referencia al nuevo objeto. La operación *Documento.miembros()* devolvería el conjunto de todos los documentos en la base de datos, la extensión del tipo.
La operacion

Documento.añadir-atributo("mejor-capítulo",Capítulo)

añadiría un nuevo atributo al tipo documento, llamado *mejor-capítulo*, refiriéndose al objeto de tipo *Capítulo*.

4.5.8.2 Declaraciones y Variables

En un ODMS, la distinción tradicional entre un DDL (Data Definition Language) y un DML (Data Manipulation Language) es poco clara:
• El DML por lo general contiene un lenguaje de consultas y un componente de lenguaje de programación.
• El DDL define tipos de objetos, y además el usuario puede declarar procedimientos y variables asociadas con los tipo de objetos.

Algunos sistemas proveen operaciones para cambiar tipos durante la ejecución. Estas variables pueden ser *persistentes* en los ODMS. Esto significa que las variables serían almacenadas permanentemente en la base de datos y retendrían sus nombres y valores, de invocación a invocación, de los programas de la aplicación. En la mayoría de los ODMS los nombres de variables son transitorias, y sólo se usan durante la ejecución del programa.

Las implementaciones del modelo relacional generalmente no permiten variables con valores de tabla. Como resultado de ésto, y la falta de estructuras de control, el DML relacional no puede expresar muchos tipos de operaciones de la base de dato. En el modelo orientado a objetos, el usuario puede definir múltiples variables que son colecciones.

4.5.8.3 Tipos de Datos Literales

La diferencia de un tipo de dato *literal* y uno de *entidad* es que el dato *literal* consiste de un valor simple como un entero, fecha, o cadena, mientras que un valor *entidad* es un grupo de valores, al cual llamamos un objeto, referido por un identificador. En el modelo orientado a objetos, hay a veces poca distinción en la forma en que los dos tipos de datos son manipulados en el lenguaje. En el modelo relacional, esto es bastante diferente. Los sistemas relacionales extendidos proveen dos mecanismos separados para definir nuevos tipos, uno para literales y otro para entidades.

Tipos de datos literales son construidos tradicionalmente dentro del DBMS relacional. La definición de nuevos tipos literales requiere información sobre el tamaño de memoria requerido para guardar los valores, y los procedimientos para codificar y decodificar los valores de su representacion externa.

Los sistemas relacionales extendidos y generadores de sistemas de base de datos proveen parametrización para la definición de tipos literales.

En los lenguajes de programación para base de datos orientada a objetos, la distinción entre literales y entidades es menos extrema, ya que la sintaxis para las declaraciones provee ambos tipos de datos.

4.5.8.4 Jerarquía de Tipos

Los modelos de datos más avanzados incluyen el concepto de *generalización*, que es la abstracción para jerarquía de tipos. La generalizacion también es conocida como herencia,

subtipos, o subclases. Al igual hay diferentes significados para *clases* de objetos (intención de tipo, extensión de tipo, representación de tipo), también hay diferentes tipos de generalizacion:

• especificación: subtipos pueden ser definidos como predicados de tipos, donde el predicado puede ser aplicado durante la ejecución.
• clasificación: subtipos pueden se reusados como conjuntos para clasificar objetos, y para diferenciar entre extensiones.
• especialización: subtipos pueden añadir atributos o métodos adicionales a un supertipo.
• implementación: subtipos pueden proveer implementaciones diferentes de los métodos definidos en los supertipos.

El uso de subtipos para estos cuatro propósitos no es necesariamente excluyente. Generalmente los sistemas de base de datos y lenguajes de programación no tienen diferentes mecanismos de subtipos para la generalización, sino que todos son agrupados de forma conjunta.

4.5.8.5 Jerarquías Avanzadas

Las jerarquías de tipos pueden tener semánticas más sofisticadas, como múltiple subtipos, u objetos con múltiples tipos. La mayoría de los ODMS permiten herencia múltiple.

4.5.8.6 Evolución de Esquema

Muchos ODMS permiten al usuario modificar la definición de los tipos, variando en la cantidad de asistencia que ofrecen al usuario para manejar estas modificaciones. Hay tres implicaciones al hacer cambios en el esquema de datos:

• la modificación a programas que usan los esquemas de datos viejos,
• la modificación a instancias existentes de los tipos modificados, y
• los efectos de los cambios al resto del esquema.

La siguiente es una taxonomía simplificada de posibles cambios a esquemas:

1. Cambios a los componentes de un tipo
    1.1 Cambios a atributos
        1.1.1 Añadir un nuevo atributo
        1.1.2 Quitar un atributo
        1.1.3 Cambiar el nombre de un atributo
        1.1.4 Cambiar el tipo de un atributo
        1.1.5 Heredar una definición diferente de un atributo
    1.2 Cambios a métodos
        1.2.1 Añadir un nuevo método
        1.2.2 Quitar un método
        1.2.3 Cambiar el nombre de un método
        1.2.4 Cambiar la implementación de un método
        1.2.5 Heredar una definición diferente de un método
2. Cambios al árbol/gráfico de herencia

2.1 Añadir un nuevo supertipo/subtipo entre tipos
2.2 Quitar un supertipo/subtipo entre tipos
2.3 Cambiar el order de la herencia (solamente en herencia múltiple)
3. Cambios a los propios tipos
3.1 Añadir un nuevo tipo
3.2 Quitar un tipo existente
3.3 Cambiar el nombre de un tipo

4.5.8.7 Definición de Tipo Dinámico

Sin la definición de tipos dinámicos, es necesario declarar todos los tipos del programa antes de que el programa se ejecute.

Hay dos problemas en la definición de tipos dinámicos:
• La definición de un nuevo esquema, que se puede resolver facilmente si el ODMS permite operaciones en esquemas dinámicos.
• La manipulación de objetos usando el nuevo esquema, que es más dificil de resolver que el problema anterior, por las siguientes razones:

a) Los programas existentes no tienen conocimiento de las nuevas definiciones de los esquemas. La versión y fechas del esquema y el programa, pueden ser usados por el ODMS para evitar anomalías.
b) Las interfaces de programación estándares de los ODMS son compiladas, haciendo imposible la manipulación de objetos usando nuevos esquemas excepto por medio de una interfaz dinámica.

4.5.8.8 Ligadura Dinámica

Ligadura significa hacer un mapa de los símbolos de un lenguaje a sus definiciones. *Ligadura dinámica* es la habilidad de hacer esta ligadura durante la ejecución, al contrario de hacerlo estáticamente durante la compilación.

4.5.9 Consultas y Lenguajes de Programación

Uno de los aspectos más importantes del lenguaje es proveer capabilidades para retiro, manipulación, y definición de datos. Es en estas áreas que los ODMS varían. En los sistemas de base de datos relacionales, el lenguaje de consultas consiste de los sublenguajes DML y DDL. Un ODMS provee un lenguaje de base de datos que incluye un lenguaje de consultas y programación más otras capabilidades:

1) Ambientes: Los lenguajes de programación de base de datos son extensiones a lenguajes de programación existentes, mientras que los sistemas de bases de datos extendidos tienen un lenguaje de base de datos diferente del lenguaje de programación de la aplicación.

2) Resultados de Consulta: Los ODMS difieren en el tipo de resultados que pueden ser obtenidos por consultas no-procedurales o declarativas, como SQL, cuando se pide buscar algo sin decir como producir las relaciones. Otros lenguajes producen un conjunto o lista de objetos.

3) Independencia de Datos: Hay dos formas de proveer independencia lógica de datos en un lenguaje de consultas: por medio de *encapsulamiento* orientado a objetos, y por medio de *datos virtuales*.

4) Encapsulamiento: Algunos ODMS adhieren un encapsulamiento estricto en los objetos, donde sólo los métodos definidos públicamente son visibles al usuario del lenguaje de la base de datos. Otros sistemas violan el encapsulamiento en el lenguaje de consultas.

5) Datos Virtuales: Algunos ODMS permiten que los datos buscados por el lenguaje de base de datos sean definidos usando atributos derivados. Cuando es transparente al usuario que los datos son derivados o guardados explicitamente, estos se llaman *datos virtuales*, permitiendo al esquema de datos conceptuales ser cambiado sin modificar los programas existentes.

6) Modelo de Datos: El lenguaje de base de datos provisto por el ODMS difiere según el modelo de datos: relacional, funcional, u orientado a objetos. El modelo de datos afecta la sintaxis y semántica del lenguaje de la base de datos.

7) Operaciones Extendidas: Además de las capabilidades de los lenguajes de programación, los ODMS generalmente extienden los lenguajes de consultas declarativos convencionales, como SQL, con nuevas operaciones como *cierre transitivo*, *recursión*, y *reglas*.

8) Sistema de Reglas: es un ingrediente esencial para permitir al ODMS ser usado en el *manejo de conocimiento*, además del manejo de datos y objetos.

9) Estandarización: Un aspecto importante en el uso de los ODMS en las aplicaciones del mundo real es la compatibilidad con estándares existentes de lenguajes de consulta, como SQL, y compatibilidad con otros ODMS, en la programación y lenguajes de consulta.

4.5.9.1 Ambientes

El enfoque en la dimensión de los lenguajes depende de si el ODMS fue construido comenzado por un lenguaje de programación existente o por un lenguaje de consultas:

• Sistemas de Base de Datos Extendido: provee un lenguaje de consultas y programación al usuario, con sistemas de tipos diferentes y ambientes de ejecución. Ambos lenguajes satisfacen la *totalidad computacional*, conteniendo construcciones de control procedurales, aunque las oraciones de los lenguajes de consulta deben ser integrados en el lenguaje de programación para poder copiar datos entre la base de datos y el ambiente de programación. Como consecuencia de esto, se pueden ejecutar y escribir las aplicaciones en dos tipos diferentes de lenguajes. Ejemplo de esto son IRIS, SYBASE y POSTGRES, aunque difieren en el modelo de datos.

• Lenguajes de Programación de Base de Datos: Proveen un solo ambiente de ejecución, el lenguaje procedural y el sistema de tipos. Generalmente tiene un lenguaje de consultas, que está integrado con el lenguaje de programación.

La diferencia crítica entre estas arquitecturas es si el lenguaje de base de datos tiene acceso a todas las variables, operaciones, y procedimientos de la aplicación. La diferencia entre la el lenguaje de programación de la aplicación y el lenguaje de consultas de la base de datos en lograr la integración de ambos es conocido como la *impedancia desigualada*.

Aspectos de Impedancia Desigualada

El problema de la *impedancia desigualada* entre los lenguaje de programación de la aplicación y el lenguaje de consultas de la base de datos ha existido por mucho tiempo. Las bases de datos relacionales generalmente proveen un lenguaje de consultas declarativo sin construcciones de control, variables, u otros aspectos de programación.

Los enfoques para tratar con el problema de la impedancia desigualada son los siguientes:

• Integración de Consultas: El enfoque más sencillo es escribir consultas que busquen todos los objetos que son o pueden ser requeridos por la aplicación, traduciendo los datos a la representación del lenguaje de programación, y copiandolos de regreso. Este enfoque permite la operación rápida del lenguaje de programación en los datos, pero incurre en bastante carga en iniciar y terminar la aplicación. Más importante aún, no permite utilizar el poder del DBMS durante la ejecución de la aplicación, limitando el acceso concurrente.

• Representación de la Base de Datos: Otro enfoque sencillo es copiar no más de un solo campo de datos u objeto fuera de la base de datos a la vez, operando en el dato de la base de datos, en lugar del programa de la aplicación. El problema con este enfoque es ejecutar las operaciones sobre los datos.

• BLOB (Binary Large OBject): En lugar de usar un esquema para una base de datos relacional, toda la información sobre un objeto grande debe ser codificado como una estructura de datos grandes en el lenguaje de programación y guardada como un solo campo *BLOB* en la base de datos. El *BLOB* puede ser buscado y guardado eficientemente, ya que solo una llamada es hecha a la base de datos. Por otro lado, se pierde la ventaja de usar un ODMS, ya que las consultas, concurrencia, y otras operaciones no pueden ser ejecutadas en los elementos de un documento usando el ODMS, sin saber éste sobre la estructura interna del *BLOB*.

• Lenguaje de Base de Datos Procedural: Otro enfoque es escribir toda o la mayoría de la aplicación en el lenguaje de consultas, cuando el lenguaje de consultas provee operaciones del lenguaje de programación, como en SYBASE. Este enfoque también es posible cuando los procedimientos del lenguaje de programación pueden ser llamados del lenguaje de consultas, como en INGRES, POSTGRES, o Starburst. El problema es que ambos enfoques sufren por la falta de totalidad de los recursos. Los programas ejecutados en el lenguaje de consultas pueden que no tengan acceso a los datos globales importantes o a las ventanas de usuario, que son accesibles del programa de la aplicación. También, el programador puede preferir escribir más de su apliacación en el lenguaje de la aplicación en lugar del lenguaje de la base de datos.

• Tipo de Datos de la Programación Extendida: Se extiende los lenguajes de programación existentes para que las relaciones se comporten como las otras estructuras de datos en el lenguaje. De todos modos, el modelo de datos de la base de datos y del lenguaje de programación no están integrados, las relaciones son añadidas como nuevos tipos de datos, y no como estructura de datos persitentes en el lenguaje de programación. Hay problemas de eficiencia con este enfoque.

• Lenguajes de Programación Persistentes: Otro enfoque es proveer de forma transparente el lenguaje de programación con estructuras de datos persistentes, guardando los datos en un ODMS relacional extendido. Este enfoque es más promisorio que los otros. De toda forma, estos sistemas no eliminarían la impedancia desigualada cuando el mismo dato es accesado usando el lenguaje de consulta de la base de datos, ya que el modelo de datos es diferente en el lenguaje de programación. Por esta razón, se está dedicando un mayor esfuerzo al diseño de lenguajes de programación para bases de datos basados en lenguajes orientados a objetos que siguen de forma muy cercana los lenguajes populares de programación.

4.5.9.2 Resultados de Consulta

Los ODMS difieren en el tipo de lenguaje de consultas que proveen.
Existen los siguientes enfoques según el resultado de las consultas en el lenguaje de consultar:

• Sin Consulta: Algunos sistemas no proveen una búsqueda asociativa de objetos por medio de un lenguaje de consulta, siendo necesario escribir programas que atraviesen manualmente los caminos de acceso entre objetos.

• Resultados de Colección: Algunos sistemas, como GemStone, ObjectStore, ONTOS y ORION, proveen una facilidad de consultas para una búsqueda asociativa que no opera en la base de datos entera, como lo hacen los lenguajes relacionales. En tales sistemas, las consultas son usadas para seleccionar objetos satisfaciendo restricciones de una colección de objetos particular, que por ejemplo, pueden ser sacadas de los atributos de los objetos.

• Resultados de la Relación: Los ODMS relacionales extendidos, como POSTGRES y Starburst, proveen un lenguaje de consultas que resulta en relaciones. En contraste a consultas de resultados de colección, los resultados pueden ser construidos de cualquier dato en la base de datos, sin tener que ser escogidos de objetos de un tipo existente o colección. Estos lenguajes de consulta son iguales o exceden la expresibilidad de SQL y otros leguajes relacionales.

• Resultados Abiertos: El sistema O_2 provee un lenguaje de consultas que puede resultar en cualquier tipo de objetos, por ejemplo un valor literal, un objeto, o una lista o conjunto de objetos. Las consultas con resultado de colección y relación pueden producir elementos de un solo dato, como caso especial. POSTGRES puede devolver relaciones múltiples, pero un lenguaje de consultas con resultado abierto puede ser distinguido por su simetría con respecto a los tipos de datos.

4.5.9.3 Independencia de Datos

Los requisitos de independencia de datos pueden ser satisfechos por los diferentes modelos de datos:

• En un modelo funcional o relacional, las funciones o vistas definidas en el lenguaje de la base de datos puede proveer mapas, pero los mapas deben ser invertibles para una completa independencia lógica de datos. Además, los esquemas externos y conceptuales deben ser definidos para proveer ocultamiento. Debe ser posible definir componentes públicos cuya interfaz no cambia, y componentes privados con acceso restringido.

• En un modelo orientado a objetos, es útil debilitar la definición estricta del encapsulamiento, para dar acceso a los datos en la porción pública del objeto, dejando compartir más flexiblemente los datos privados, y definiendo métodos "funciones" que actúen sobre los atributos virtuales.

El factor más importante para la independencia física de los datos es el uso de un lenguaje de consultas declarativo en los programas y métodos. Los lenguajes declarativos especifican los datos que el usuario quiere sin especificar como obtenerlos. Como resultado, el optimizador de consultas puede determinar automaticamente si los índices, ligas, u otros métodos de acceso existen en los atributos derivados. De forma transparente, se puede escoger la técnica más eficiente para retirar los objetos satisfaciendo la consulta. Si la estructura física es luego cambiada, la nueva estructura sería usada y los programas no tendrían que ser cambiados. Como la mayoría de los lenguajes de base de datos de ODMS son diseñados combinando un control procedural y un lenguaje de consultas, la independencia física de los datos puede perderse, porque el lenguaje resultante no se da para simples transformaciones algebraicas que pudieran ser hechas para optimizar las operaciones, tales como selecciones relacionales, proyecciones y juntas.

4.5.9.4 Encapsulamiento

Encapsulamiento provee independencia de datos por medio de la implementación de métodos, permitiendo a las porciones privadas del objeto ser cambiadas sin afectar el programa que usa el tipo de objeto. Hay dos requisitos para el encapsulamiento y la independencia de datos que logra:

1. Ocultamiento: Algunos componentes del objeto deben ser públicos, mientras que otros son privados.
2. Mapas: Los métodos definen un mapa entre los componentes públicos y privados.

Cualquier ODMS que provee encapsulamiento satisface ambos requisitos. En la definición estricta de encapsulamiento, ejemplificado por Smalltalk, existen varias restricciones:

- Sólo los procedimientos (métodos), pueden estar en la parte pública del objeto, y no los datos.
- Los métodos están definidos con un lenguaje procedural.
- Los métodos pueden ver y manipular sólo los datos dentro del propio objeto.

Con un lenguaje de base de datos que provee aspectos procedurales (lenguaje de programación) y declarativos (lenguaje de consultas), se puede dejar la segunda restricción. Los métodos pueden definirse en uno o ambos tipos de lenguajes.

El uso de un lenguaje declarativo es importante para la independencia física de los datos. Los lenguajes de consulta tradicionales están diseñados sobre la suposición de una estructura simple, uniforme, y visible para los objetos en donde opera. Un enfoque orientado a objetos dicta que los atributos y relaciones asociadas con los objetos sean invisibles al usuario, teniendo que operar el lenguaje de consultas sólo en los métodos.

Algunos ODMS evitan estos problemas, violando el encapsulamiento en el lenguaje de consultas, permitiendo que los atributos y relaciones de los objetos sean visibles. Esto se considera *ocultamiento parcial*, donde la implementación privada del objeto está oculta de los programas, pero no de las consultas.

4.5.9.5 Datos Virtuales

La noción de datos virtuales es análoga a la de vistas relacionales. La razón más importante del uso de datos virtuales es para la independencia lógica de datos. Un esquema de datos modificable, estaría basado en datos derivados que pueden definirse para los atributos y relaciones viejas, para que los programas y consultas existentes sigan funcionando.

El modelo de datos difiere en el enfoque a los datos virtuales. La idea de vistas relacionales es adaptada y ampliada en los modelos relacionales extendidos. Los datos derivados definidos usando métodos orientados a objetos en un lenguaje procedural puede ser más poderoso que las vistas relacionales, ya que acciones separadas pueden ser especificadas para el retiro y actualización, además de que computaciones arbitrarias pueden ser especificadas. El modelo de datos funcional es ideal para definir datos virtuales, porque las funciones *computadas* asociadas con los objetos son indistinguibles de las funciones *guardadas* (o sea atributos ordinarios).

Vistas relacionales, funciones, métodos orientados a objetos, u otras formas de datos virtuales no son suficientes para la independencia lógica de los datos. Ellos proveen *mapas* pero no *ocultamiento*, al igual que muchos de los ODMS.

4.5.9.6 Modelos de Datos

Otra dimensión en la cual los ODMS difieren, es en el modelo de datos usado por el lenguaje: relacional extendido, orientado a objetos, o funcional. El modelo de datos afecta en estos lenguajes más la sintaxis que el poder semántico del sistema. La porción del lenguaje de programación puede ser usada para compensar las limitaciones del lenguaje de consultas.

Dependiendo del modelo de datos, las consultas y los lenguajes de programación pueden ser combinados en más de una forma. En POSTGRES, por ejemplo, los procedimientos del programa de aplicación pueden ser guardados como atributos de tablas, para que puedan ser invocados por las consultas. En IRIS, O_2, ONTOS y ObjectStore, las consultas están integradas en los programas.

La elección del modelo de datos es posiblemente el aspecto más controversial en el manejo de datos de los objetos: Determinar la representacion más natural y la sintaxis para una aplicación es a veces un proceso subjetivo.

4.5.9.7 Operaciones Extendidas

El poder de expresibilidad del lenguaje de consultas puede variar según los ODMS. Expresibilidad adicional es importante porque permite mayor codificación de la aplicación en el lenguaje declarativo, que puede ser más conciso o proveer una independencia física de datos más alta que del lenguaje de programación.

Por ejemplo, algunos ODMS incluyen alguna forma de *cierre transitivo* en el lenguaje de consultas, permitiendo que las relaciones sean atravesadas iterativamente para encontrar todos los posibles objetos.

4.5.9.8 Sistema de Reglas

POSTGRES incorpora directamente reglas en el lenguaje de consultas. Otro enfoque es construir un sistema experto sobre de ODMS. La ventaja de este enfoque en capas es que permite al sistema experto y al ODMS ser desarrollados, mantenidos, y optimizados de forma independiente. La desventaja es que es más dificil tratar con la funcionalidad o aspectos de rendimiento que afectan a los dos a la vez, como los mecanismos necesarios para detectar cuando los predicados de las reglas deben ser re-evaluados.

4.5.9.9 Estandarización

Al igual que SQL ha sido importante como el lenguaje de consultas estándar para DBMS relacionales, es importante definir un estándar para los ODMS, sin importar si están basados en un modelo de datos orientados a objetos, relacionales extendidos, o funcionales.

Existen cuatro aspectos estandáres para ODMS:

• Modelo de Datos: No existe actualmente un modelo de datos orientado a objetos único, diferenciándose según el tipo de encapsulamiento, herencia, procedimientos, y otrso aspectos. De forma similar, existen muchos modelos de datos relacionales extendidos y funcionales.

• Lenguaje de Consulta: Aunque hubiera un modelo de datos comunes, no existe el equivalente de un lenguaje de consultas estándar para los diferentes modelos.

• Lenguaje de Programación: Es necesario un estándar para los sistemas de base de datos extendidos que proveen extensiones procedurales al lenguaje de consultas, y para los lenguajes de programación de base de datos que proveen extensiones del lenguaje de programación para operaciones de base de datos. También, un modelo de datos estándar es necesario que trabaje con mútliples lenguajes, tal como es posible con SQL.

• SQL: es un estándar para DBMS relacionales, y es actualmente el único estándar para acceso de base de datos que es globalmente implementado y reconocido. Por lo cual afecta los futuros estándares de los nuevos productos de ODMS.

Los sistemas de bases de datos relacionales extendidos tienen ventajas en todas estas dimensiones de estándares, ya que comparten un modelo relacional común apoyado por grupos de estándares. Por otro lado, el progreso en los estándares relacionales ha sido lento, y, ni POSTGRES, ni Starburst, ni las extensiones a sistemas relacionales extendidos, como INGRES y SYBASE, han sido estandarizados.

SQL es importante para los ODMS por la siguientes razones:

• Acceso de base de datos heterogéneas: Para un cliente típico, diferentes tipos de datos en diferentes departamentos serían guardados en DBMS separados. Es necesario usar SQL para buscar datos de otros repositorios, o permitir a otros DBMS buscar datos de los ODMS.

• Familiaridad: Existen ya muchos programadores de aplicaciones de bases de datos que están familiarizados con la sintaxis y semántica de SQL.

• Estándar: SQL está estandarizado, mientras que ningún otro estándar está saliendo de las capabilidades más poderosas de los ODMS.

Por lo tanto, SQL es un punto natural para nuevas estandarizaciones. Pero por otro lado es dificil extender SQL para incorporar la semántica de los ODMS.

4.5.10 Otros Aspectos

El manejo de datos de objetos introduce nuevos aspectos para concurrencia y recuperación de bases de datos, ya que los datos pueden ser utilizados por períodos más largos. Versiones múltiples del mismo objeto pueden ser guardados en la base de datos, y la semántica de los objetos puede influir la granularidad o tipo de control de la concurrencia. En los sistemas de base de datos relacionales, el mecanismo de transacción provee a la vez control de concurrencia y mecanismos de recuperación que son normalmente adecuados para las aplicaciones de negocios. Los ODMS usan modificaciones de transacciones, y nuevas técnicas como versiones.

4.5.10.1 Persitencia

Concurrencia, recuperación, integridad física, e integridad lógica son importantes ya que los datos en el ODMS son *persistentes*; o sea, los datos existen después de la sesión del usuario y la ejecución del programa de la aplicación.

Los datos transitorios duran sólo durante la invocación del programa, mientras que los datos persistentes son retenidos hasta que ya no sean usados, siendo luego removidos. En los ODMS tradicionales, los datos transitorios son guardados en las variables del lenguaje de programación, y los datos persistentes son guardados en la base de datos.

La habilidad para manipular los datos transitorios puede ser útil en una variedad de aplicaciones, como para representar la información del programa que está siendo manipulado, guardando los datos de forma persistente sólo luego de que los cambios han sido completados y revisados, y guardando sólo un subconjunto de los datos transitorios.

Los ODMS pueden variar según cómo los objetos se vuelven persistentes:

• Por tipo: Un objeto puede hacerse persistente cuando es creado, basado en su tipo (tipos persistentes vs. tipo transitorios), como en Objectivity/DB u ONTOS. El tipo puede identificarse como persistente en la declaración, o siendo un subtipo de los tipo de objetos persistentes incluidos en el sistema.

• Por llamada explícita: El usuario puede especificar explicitamente la persistencia de un objeto, como en ObjectStore. Esta llamada puede ser ejecutada cuando el objeto es creado, o en algunos sistemas, puede ser ejecutado en cualquier momento.

• Por referencia: Algunos sistemas determinan automaticamente la persistencia de los objetos según algún objeto raíz persistente. Este enfoque es utilizado por GemStone y PS Algol, siendo análogo al de los lenguajes de recolección de basura como Lisp y Smalltalk.

Los últimos dos enfoques son usados los más convenientes. Persistencia es ortogonal al tipo del objeto, y la persistencia por referencia es el enfoque más poderoso, ya que libera al usuario de pensar sobre la persistencia para cada objeto.

Las implementaciones de objetos persistentes requieren que los objetos sean guardados en un área especial de la memoria, requiriendo que los objetos sean copiados cuando se hacen persistentes. Determinando persistencia por referencia puede involucrar un costo significante.

4.5.10.2 Transacciones

Los ODMS generalmente no descartan transacciones como mecanismos de concurrencia y recuperación; en lugar, simplemente aumentan las transacciones, permitiendo transacciones *largas* y *anidadas*:
• Tranasacciones largas son transacciones que pueden durar varias horas o días.
• Transacciones anidadas son transacciones que permiten un grupo de actualizaciones en una transacción existente, sea hecha o abortada sin terminar la transacción de nivel más alto.

Hay por lo general dos usos para las transacciones:

• En algunos casos, las transacciones pueden ser usadas para controlar concurrencia. Si porciones de la base de datos tienen características de acceso similares a aplicaciones de negocios tradicionales, o sea datos accesados frecuentemente que son compartidos entre usuarios, entonces las transacciones cortas tradicionales son buenas para el control de la concurrencia. Por otro lado, cuando una transacción continúa por horas o días, entonces nuevos enfoques son necesarios. Para el control de concurrencia de largo alcance, dos soluciones han sido sugeridas, *transacciones conversacionales*, que permiten que los datos sean sacados por un período largo, y *versiones*.

• Las transacciones son importantes para recuperación y para ejecutar un grupo de operaciones como una acción indivisible. Recuperación es particularmente importante cuando una sesión de la base de datos dura períodos largos. Transacciones cortas tradicionales son suficientes para ejecutar acciones indivisibles, al igual que la recuperación en la mayoría de los casos. Por otro lado, si las *transacciones conversacionales* de largo alcance son usadas para el control de concurrencia, podría ser necesario para propósitos de recuperación, anidar las transacciones dentro de la transacción de largo alcance.

Los productos DBMS tradicionales no pueden manejar transacciones que duren horas o días, tales transacciones pueden abortarse espontaneamente, o pueden exhibir un pobre rendimiento. Este problema con las transacciones de largo alcance es principalmente un problema de implementación. Otro problema es que los datos pueden estar bloqueados por días o semanas.

4.5.10.3 Versiones de Objetos

El uso de *versiones* provee una alternativa para propósitos de recuperación a las *transacciones converasacionales* o anidadas. El uso de mútliples versiones de objetos permite la coordinación entre múltiples usuarios y diferentes copias de objetos.

La funcionalidad más básica requirida para versiones es la creación y destrucción de versiones de objetos. ORION, ITASCA, VERSANT, ObjectStore, Objectivity/DB y ONTOS proveen la capabilidad de crear y destruir versiones de objetos. Un usuario o programa pueden crear nuevas versiones de objetos explicitamente, y una nueva versión puede ser creada implicitamente cuando el objeto es modificado.

Versiones en un ODMS pueden simplificar mucho las aplicaciones que deben mantener mútliples versiones de datos. Sin esta funcionalidad, la aplicación debe implementar semánticas de versión y mantener información de versión explicitamente en la base de datos.

Historias de versiones muy complejas pueden resultar de un objeto particular. Versiones ramificadas, dos o más nuevas versiones independientes de un objeto, pueden ser útiles en un ambiente de diseño cuando dos o más ingenieros deben trabajar en forma independiente en revisiones que requieren algunos de los mismos objetos. Eventualmente, los ingenieros deben integrar sus trabajos, crear una sola versión del objeto que superpone las versiones ramificadas.

En algunos sistemas, las versiones pueden clasificarse como *transitorias*, *trabajando*, y *sueltas*. Una versión *transitoria* es promovida a una versión *trabajando* visible a otros usuarios al final de la sesión, y una versión *trabajando* es promovida a una versión *suelta* cuando los objetos son congelados.

Versiones se pueden aplicar a objetos compuestos. Un ODMS puede permitir la aplicación de versiones al propio esquema, simplificando el problema de la evolución de esquema.

4.5.10.4 Configuraciones

Una *configuración* es una colección de versiones de objetos en una base de datos que son mutuamente consistentes.

Ejemplo: Si se hacen actualizaciones a 20 módulos de software para crear una nueva versión de un sistema operativo, entonces las versiones de los nuevos 20 objetos, más las versiones existentes de todos los otros módulos en el sistema operativo, representan una configuración. Las referencias entre los objetos en una configuración, por ejemplo, para los módulos definiendo una estructura de datos comunes, deben ser mantenidas según las versiones apropiadas de los objetos en la configuración cuando se hace la actualización.

Hay dos enfoques a configuraciones en los ODMS:

• Los ODMS pueden proveer un *mecanismo* pero no una *política* para el manejo de configuraciones, dejando el problema de manejo de configuración para ser resuelto por el usuario. Este enfoque es tomado por Objetivity/DB.

• Los ODMS pueden implementar configuraciones automaticamente, con algunas opciones de política especificada por el usuario. ObjectStore usa este enfoque.

En el primer caso, los ODMS deben proveer mecanismos suficientes para implementar configuraciones en la aplicación de forma conveniente. Por ejemplo, en Objectivity/DB, los objetos compuestos pueden ser usados para representar configuraciones, dando al usuario tres opciones de mantenimiento de versiones para relaciones entre objetos: *mover*, *sacar*, y *copiar*. Un atributo inverso representando una relación con un objeto con una versión es marcado con una de las opciones. Cuando una nueva versión del objeto es creada, la acción correspondiente es tomada, que puede ser una de las siguientes:

• mover: la referencia debe ser movida a la nueva versión del objeto, y debería ser nula en la versión vieja.
• sacar: la referencia debe quedarse con la versión vieja, y debe ser nula en la nueva versión.
• copiar: la nueva y vieja versión deben tener la misma referencia en ambas.

Una variedad de otros enfoques han sido tomados para mantener referencias consistentes entre versiones. Un ODMS puede proveer referencias *estáticas* y *dinámicas* a objetos:
• las referencias estáticas se integran a la versión actual del objeto,
• las referencias dinámicas siempre apuntan a la versión más reciente del objeto referido.

El usuario puede especificar una versión por omisión de un objeto para ser usado para todas las referencias, a menos que esté especificado de lo contrario. En ORION, una aplicación puede pedir una *notificación de cambio* cuando una nueva versión de un objeto es creado.

El conjunto de mecanismos que un ODMS provee para mantener consistencia para referencias de versiones, junto con los objetos compuestos y métodos definidos por el usuario asociados con objetos, puede implementar una variedad de políticas de manejo de configuraciones básicas.

Hay otras diferencias en la funcionalidad provista por el ODMS. Por ejemplo, ORION provee solamente semántica de *mover* y *sacar* para referencias, y VERSANT no contiene objetos compuestos, pero la filosofía general de estos sistemas es la misma:
a) los usuarios definen su propio manejo de configuración sobre los mecanismos básicos provistos, o
b) se construye el manejo de configuraciones dentro del ODMS. Hay muchas formas de hacer esto, dependiendo de la política para el trato de versiones, referencias, y consultas. Pero el enfoque tomado por ObjectStore es probablemente el más simple:

• Las configuraciones están representadas como objetos especiales, con un identificador, nombre, u otra forma de indentificación.
• Existe siempre una *configuración actual* en la sesión de la base de datos; el usuario puede abrir una configuración actual o crear una nueva configuración para cambiar la configuración actual.
• Una actualización es hecha en la base de datos en el contexto de la configuración actual. Nuevas versiones de los objetos creados, como resultado de las actualizaciones en una sesión, son marcadas con la configuración bajo la cual han sido creadas.
• La lectura de datos durante una sesión se hace en el contexto de la configuración actual. Cuando el objeto es tomado por el usuario, la versión del objeto en la configuración actual, o en la configuración más reciente en el orden parcial de las configuraciones definido por el usuario, es devuelto al usuario.

4.5.10.5 Semántica de Concurrencia basada en objetos

Otro enfoque a la concurrencia en ODMS se basa en una elaboración de transacciones utilizando semántica de concurrencia de objetos definida por el usuario. Este enfoque aprovecha el hecho de que los objetos son actualizados solamente por medio de métodos. Tal enfoque puede permitir acceso más concurrente que en las transacciones tradicionales basadas en la *semántica de leer-escribir*. En la semántica de transacción de leer-escribir, cualquier lectura o escritura de datos es bloqueada implicitamente. Control de concurrencia específica a los métodos, permite que la semántica sea más flexible que en los conflictos de leer-escribir. Los mecanismos de control de concurrencia pueden operar en la granularidad de los objetos.

4.5.10.6 Datos de Usuario

POSTGRES provee una capabilidad que puede ser usada para concurrencia o versiones. Los objetos pueden ser "sellados" para identificar cual usuario los ha actualizado de último, y a que hora fueron actualizados. Este sello puede ser buscado en un récord así como cualquier otro atributo, y puede ser usado en consultas para examinar la historia de la base de datos. Los sellos tañaden una nueva dimensión al ODMS, pero en general no son un reemplazo para versiones, transacciones, o candados para control de concurrencia. Son particularmente útiles para historias de consultas.

4.5.10.7 Protección

Se define protección en dos componentes: *resguardo* (*safeness*) y *seguridad*:

• Resguardo significa que el usuario no puede destruir la estructura de datos por medio del lenguaje de programación, lenguaje de consultas, u otras herramientas.
• Seguridad significa que el usuario no puede accesar datos para los cuales no tiene autorización. Resguardo es un prerequisito necesario para seguridad.

Las soluciones tradicionales a la protección no se aplican a los ODMS como resultado de las consideraciones de rendimiento. Por ejemplo, los ODMS con buen rendimiento hacen por lo general un *cache* de objetos en el mismo espacio de datos de la aplicación. En la mayoría de los ambientes de programación, los datos que están en el mismo espacio que los programas de los usuarios, no están resguardados, ya que es posible examinar los objetos fisicamente adyacentes a los buscados.

Los ODMS con un sistema de base de datos extendido pueden resolver el problema de la protección de la misma forma que los DBMS relacionales tradicionales, poniendo los datos sensibles en un servidor de base de datos en un cuarto con llave que puede ser accesado solamente por la base de datos mediante un procesador de lenguaje de consultas que reside en el servidor. Si el procesador de consultas está resguardado, o sea que no puede ser comprometido por el usuario mediante acceso directo a lugares de memoria fisica, entonces sólo los objetos que pueden ser retirados y puestos en el *cache* de la máquina del usuario son aquellos que el usario puede accesar.

Los ODMS con un lenguaje de programación de base de datos pueden proveer un grado de protección si el lenguaje de programación de la base de datos está resguardado, ya que el compilador pudiera generar código para revisar la autorización de acceso.

En cualquier arquitectura, se pueden definir predicados complejos para la autorización para accesar datos. Acceso a atributos particulares, relaciones, o métodos puede ser limitados a usuarios específicos, y un método puede restringir las operaciones de usuario basado en las listas de acceso agregadas a cada objeto. La seguridad es más complicada que en los DBMS relacionales, ya que la autorización puede estar basada en métodos, objetos compuestos, u objetos individuales.

Una metodología de protección eficiente que puede ser provista en una arquitectura de lenguaje de programación de base de datos puede ser basada en unidades físicas, como páginas, segmentos o archivos. En tal esquema, los objetos deben estar ubicados en áreas físicas separadas para diferente autorización. GemStone hace esto al nivel de segmentos, permitiendo a los objetos, y atributos o métodos de objetos, ser ubicados en segmentos diferentes.

4.6 Bases de Datos Relacionales

El uso de un diseño orientado a objetos trasciende la implementación, que podría ser hecha usando una base de datos (DB). También trasciende la propia elección del DB, que puede ser una base de datos jerárquica, en red, relacional u orientada a objetos.

Conceptos Generales

Base de Datos: Repositorio de datos guardado en uno o más archivos.

DBMS: sistema de manejo de base de datos, administrando repositorios de datos permanentes.

Razones para el uso de DBMS:
• recuperación de caída: el DB está protegido de fallas de hardware y errores de usuarios.
• compartir entre usuarios: múltiples usuarios pueden accesar el DB al mismo tiempo.
• compartir entre aplicaciones: múltiples aplicaciones pueden leer y escribir datos al mismo DB, facilitando la comunicación entre los diferentes programas.
• seguridad: los datos pueden ser protegidos contra acceso no autorizados de leer o escribir.
• integridad: se puede especificar reglas que deben ser satisfechas. El DBMS puede controlar la calidad de los datos más allá del control de la aplicación.
• extensibilidad: los datos pueden ser añadidos al DB sin interrumpir los programas. Los datos pueden ser reorganizados para una ejecución más rápida.
• distribución de datos: el DB puede ser distribuido en diferentes lugares, organizaciones y plataformas de hardware.

Aunque el orden puede variar, el ciclo de vida para la mayoría de las aplicaciones de DB es el siguiente:

1) Diseñar la aplicación.
2) Diseñar la arquitectura para integrar las aplicaciones con el DB.
3) Seleccionar la plataforma específica del DBMS.
4) Diseñar el DB, escribiendo código para construir las estructuras particulares.
5) Escribir código en lenguaje de programación para compensar las limitaciones del DBMS, proveer interfaz de usuario, validar datos, y ejecutar computaciones.
6) Entra los datos al DB.
7) Correr la aplicación actualizando y consultando el DB.

Modelo de Datos o Esquema: Diseño del DB.

Hay dos enfoques generales para el diseño de DB:
• impulsado por atributos: compilar una lista de atributos relevante a la aplicación, y sintetizar los grupos de atributos que preservan las dependencias funcionales.
• impulsado por entidad: descubrir las entidades que son importantes en la aplicación. En general hay menos entidades que atributos y corresponde al modelo de objeto.

La arquitectura ANSI para DB relacionales consiste de tres esquemas, como se muestra en la Figura 4.6:

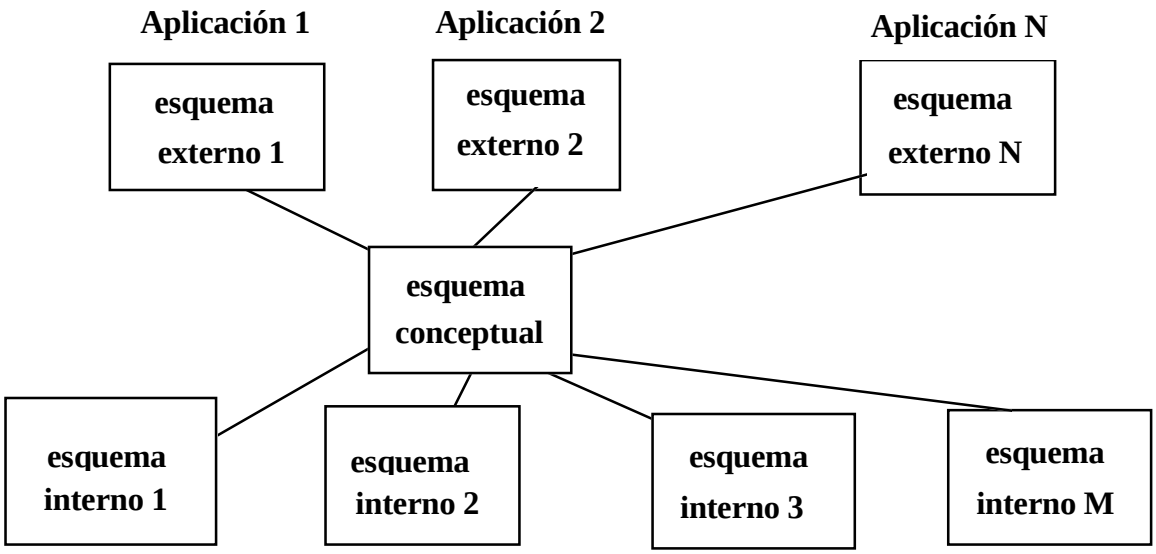


Figura 4.6. Los tres esquemas, externo, conceptual, e interno, de la arquitectura ANSI para bases de datos relacionales.

Las tres capas de la arquitectura ANSI para base de datos relacionales son:
• externa: una vista o abstracción global del esquema conceptual. Aísla las aplicaciones de la mayoría de cambios al esquema conceptual.
• conceptual: es el diseño del DB desde la perspectiva de la empresa, integrando aplicaciones relacionales y ocultando peculiaridades internas del DBMS.
• interna: trata con las limitaciones y características de los DBMS específicos. Incluye código necesario para implementar el esquema conceptual.

Correspondencia con el modelo de objetos:
• se construye un modelo para cada esquema externo
• y otro modelo para el esquema conceptual

4.6.1 Conceptos de DBMS relacional

Codd inventó el modelo de datos relacional basado en el concepto de *tablas*, donde el DBMS relacional (RDBMS) que maneja estas tablas consiste de:

1) estructura de datos
2) operadores
3) reglas de integridad
4) formas normales
5) vistas

4.6.1.1 Estructura de Datos

Un DB relacional aparece lógicamente como una colección de tablas, con un número específico de columnas y un número arbitrario de filas. Un valor simple se guarda en cada intersección de una fila y una columna en una tabla.

atributos: columnas de tablas, correspondiendo directamente a atributos en el modelo de objetos

tuplas: filas de tablas, correspondiendo a instancias de objetos y ligas.

La teoría de DB relacional dicta que cada atributo debe ser asignado a un *dominio*, un conjunto de valores legales que abarca más información que el solo formato de datos. La mayoría de los RDBMS no apoyan el concepto de *dominios*, sólo datos simples, como números y caracteres.

Ejemplo: el *dominio* puede prevenir operaciones incorrectos, como añadir un costo a un peso.

4.6.1.2 Operadores

SQL es el lenguaje más popular para RDBMS. SQL es pequeño e incompleto, teniendo algunas fallas técnicas, como la violación de principios modernos de la teoría de lenguajes.

Los siguientes son algunos de los operadores típicos en SQL:

```
SELECT lista-atributos
FROM tabla-1, tablas-2
WHERE predicado-es-cierto
```

El predicado especifica cuales filas se retienen, siendo devueltos como respuesta a la consulta. Existen otros comandos para crear tablas, insertar filas, eliminar filas, etc. Las interfaces con lenguajes de programación permiten accesar sólo parte del RDBMS, mientras que con SQL se accesa todo.

4.6.1.3 Reglas de Integridad

La mayoría de los RDBMS no tienen apoyo suficiente para la *integridad* de la base de datos:

• Integridad de Entidad: cada tabla debe tener exactamente una *llave primaria*, que es una combinación de uno o más atributos, cuyos valores determinan de forma no ambigua cada fila en la tabla (la *llave primaria* es una *llave candidata*). La *llave externa* es la llave primaria de una tabla usada dentro de otra tabla.

Ejemplo: En las tablas ilustradas en las Figura 4.7 y 4.8, *ID persona* es la llave primaria de la tabla *Persona*. *ID compañía* es la llave primaria de la tabla *Compañía*. *ID compañía* es la llave externa para la tabla *Persona.*

ID persona	nombre persona	dirección	ID compañía
1	Juan Perez	Paseo Reforma 3	1001
5	Luis Miguel	Insurgentes 19	1002
999	Pedro Ruiz	Chapultepec 180	1001

Figura 4.7. Tabla *Persona*

ID compañía	nombre compañía	dirección
1001	Sears	Medellín 49
1002	Palacio del Hierro	Sevilla 92
1003	Aurrera	Horacio 139

Figura 4.8. Tabla *Compañía*.

• Integridad Referencial: requiere que el RDBMS guarde cada llave externa de forma consistente con su llave primaria correspondiente (e.g. si se quita 1001 de la tabla *Compañía*, la tabla *Persona* seráa inconsistente).

4.6.1.4 Formas Normales

Las *formas normales* son reglas desarrolladas para evitar inconsistencias lógicas para las operaciones de actualización de las tablas. Cada forma normal prohibe una forma de redundancia en la organización de las tablas que podría dar un resultado sin sentido si una tabla se actualizara independiente de las otras. Hay múltiples niveles de formas normales. Cada nivel más alto añade restricciones a la forma normal del nivel más abajo. A medida que el diseñador de la base de datos satisface las formas normales más altas, las tablas tienden a fragmentarse. Las formas normales mejoran la consistencia de la base de datos al costo de acceso añadido y consultas más lentas. Se podría violar las formas normales por razones tales como rendimiento.

Primera forma normal: No se puede contener un valor repetido para cada atributo de la tabla.

Ejemplo: En la Figura 4.9 se muestra dos valores, *enfriador* y *calentador*, bajo *nombre de equipo* violando la primera forma normal.

nombre de planta	nombre de equipo	administrador de planta	fabricante de equipo	dirección de fabricante
azúcar	enfriador, calentador	Juan Perez	ABC	Coahuila 10
maíz	bomba	Pedro Gomez	XYZ	Medellín 93
maíz	calentador	Pedro Gomez	ABC	Coahuila 10

Figura 4.9. Tabla violando la primera forma normal.

Ejemplo: En la Figura 4.10 se cambia la doble entrada como dos filas separadas para satisfacer la primera forma normal.

nombre de planta	nombre de equipo	administrador de planta	fabricante de equipo	dirección de fabricante
azúcar	enfriador	Juan Perez	ABC	Coahuila 10
azúcar	calentador	Juan Perez	ABC	Coahuila 10
maíz	bomba	Pedro Gomez	XYZ	Medellín 93
maíz	calentador	Pedro Gomez	ABC	Coahuila 10

Figura 4.10. Tabla que satisface la primera forma normal, pero viola la segunda forma normal. Llave primaria: (*nombre de planta*, *nombre de equipo*)

Segunda forma normal: Luego de satisfacer la primera forma normal, donde cada fila tiene una sola llave primaria, cada atributo de llave no-primaria debe depender totalmente de la llave primaria.

Ejemplo: Se viola la segunda forma normal ya que *fabricante de equipo* y *dirección de fabricante* dependen completamente de la llave primaria (*nombre de planta*, *nombre de equipo*), mientras que *administrador de planta* solo depende en parte de la llave primaria (*nombre de planta*).

Ejemplo: En la Figura 4.11 se parte la dependencia parcial en otra tabla que satisface la segunda forma normal.

nombre de planta	administrador de planta
azúcar	Juan Perez
azúcar	Juan Perez
maíz	Pedro Gomez
maíz	Pedro Gomez

Llave primaria: (*nombre de planta*)

nombre de planta	nombre de equipo	fabricante de equipo	dirección de fabricante
azúcar	enfriador	ABC	Coahuila 10
azúcar	calentador	ABC	Coahuila 10
maíz	bomba	XYZ	Medellín 93
maíz	calentador	ABC	Coahuila 10

Llave primaria: (*nombre de planta*, *nombre de equipo*)

Figura 4.11. Tablas que satisfacen la segunda forma normal, pero violan la tercera forma normal.

Tercera forma normal: Luego de satisfacer la segunda forma normal, cada atributo de la llave no-primaria depende directamente de la llave primaria.

Ejemplo: Se viola la tercera forma normal porque hay una dependencia transitiva; *dirección de fabricante* depende de *fabricante de equipo* que depende de la llave primaria. La tercera forma normal requiere una dependencia directa de la llave primaria.

Ejemplo: En la Figura 4.12, se separan en dos tablas para resolver la violación.

nombre de planta	administrador de planta
azúcar	Juan Perez
azúcar	Juan Perez
maíz	Pedro Gomez
maíz	Pedro Gomez

llave primaria: (*nombre de planta*)

nombre de planta	nombre de equipo	fabricante de equipo
azúcar	enfriador	ABC
azúcar	calentador	ABC
maíz	bomba	XYZ
maíz	calentador	ABC

llave primaria: (*nombre de planta, nombre de equipo*)

fabricante de equipo	dirección de fabricante
ABC	Coahuila 10
ABC	Coahuila 10
XYZ	Medellín 93
ABC	Coahuila 10

llave primaria: (*fabricante equipo*)

Figura 4.12. Tablas que satisfacen la tercera forma normal.

Hay formas normales más altas que por lo general no son necesarias. Cuando hay un número muy grande de filas, las razones para observar las formas normales son más importantes.

4.6.1.5 Vistas

Una *vista* es una tabla virtual que es computada dinamicamente según sea necesario. Una vista no existe físicamente pero es derivada de una o más listas existentes. En teoría, las vistas son los medios para derivar esquemas externos de los esquemas conceptuales para la arquitectura de tres esquemas ANSI. En la práctica las vistas son menos útiles. Los RDBMS comerciales apoyan por lo general la lectura a través de vistas pero rara vez la escritura por medio de las vistas. Hay aspectos semánticos asociados con escribir a través de las vistas que la mayorías de los RDBMS evitan.

4.6.2 Diseño de Base de Datos Relacionales

Durante el diseño de la base de datos se deben seguir los siguientes pasos:
• se debe formular modelos de objetos para los esquemas externos y conceptuales.
• se debe traducir cada modelo de objeto a tablas ideales (*modelo de tabla*). Vistas y/o programas de interfaz conectan las tablas externas a las tablas conceptuales, correspondiendo luego a esquemas internos.

Cada modelo de tabla consiste de muchas tablas ideales, genéricas e independiente del DBMS. Para traducir el modelo de objetos a las tablas ideales se debe escoger entre diferentes

posibilidades. Por ejemplo, hay varias formas de hacer una correspondencia de asociaciones y generalización a tablas. También se debe añadir detalles que faltan en el modelo de objeto, como llaves primarias y llaves candidatas para cada tabla; y se debe considerar si los atributos pueden ser nulos. Se debe asignar un *dominio* a cada atributo y listar los grupos de atributos que pueden ser accesados frecuentemente.

El esquema interno consiste de comandos SQL para crear tablas, atributos, y estructuras para mayor rendimiento. La generación de los comandos SQL requiere asignar tablas a los archivos del DBMS y mantener las restricciones sobre el tamaño y legalidad de los nombres. Se tiene que definir, si es posible, los dominios del RDBMS, o convertir los dominios en tipo de datos.

4.6.2.1 Uso de Identificadores

Cada tabla derivada de una clase tiene un identificador para la llave primaria, y uno o más identificadores para la llave primaria de la tabla derivada de las asociaciones. La estrategia es compatible con la notación de los lenguajes orientados a objetos, donde los objetos tienen una identidad aparte de sus propiedades.

Existen beneficios en el uso de identificadors, ya que éstos son inmutables y completamente independientes a cambios en los valores de los datos y su lugar físico. Su estabilidad es particularmente importante para las asociaciones, ya que ellas se refieren a objetos. Los identificadores proveen un mecanismo uniforme para referenciar los objetos. La mayor inconveniencia es la necesidad de generar los identificadors , ya que los RDBMS no proveen un soporte para su creación.

4.6.2.2 Diseño de Clases en Tablas

Cada clase corresponde, a una o más tablas, donde una tabla puede corresponder a una o más clases. Los objetos en una clase pueden ser partidos horizontalmente y/o verticalmente.

Ejemplo: Si una clase tiene varias instancias de las cuales sólo algunas son referenciadas, entonces, una partición horizontal puede mejorar la eficiencia al poner los objetos accesados frecuentemente en una tabla y el resto en otra, como se muestra en la Figura 4.13.

ID persona	nombre persona	dirección
1 5	Juan Perez Luis Miguel	Paseo Reforma 3 Insurgentes 19

ID persona	nombre persona	dirección
999	Pedro Ruiz	Chapultepec 180

Figura 4.13. Partición horizontal de una tabla.

Si una clase tiene atributos con diferentes patrones de acceso, entonces, puede ayudar partir los objetos verticalmente, como se muestra en la Figura 4.14.

ID persona	nombre persona
1	Juan Perez
5	Luis Miguel
999	Pedro Ruiz

ID persona	dirección
1	Paseo Reforma 3
5	Insurgentes 19
999	Chapultepec 180

Figura 4.14. Partición vertical de una tabla.

Ejemplo: El modelo de objeto para *Persona* se muestra en la Figura 4.15. La clase *Persona* tiene atributos *persona*, *nombre*, y *dirección*. El modelo de tabla incluye estos atributos y añade el identificador implícito del objeto, como se muestra el Figura 4.16. Se añaden detalles durante la formulación del modelo de tabla. Se especifica que *ID persona* no puede ser nulo ya que es una llave candidata; *nombre persona* tampoco puede ser nulo, aunque no sea una llave candidata. El atributo *dirección* si puede ser nulo. Se asigna un dominio a cada atributo, especificando la llave primaria para cada tabla, y se nota los grupos de atributos frecuentemente accesados. El código SQL crea la tabla *Persona*, y hace el mapa de los dominios a los tipo de datos, como se muestra en la Figura 4.17.

Persona
nombre persona dirección

Figura 4.15. Modelo de objeto para *Persona*.

Nombre Atributo	Nulo?	Dominio
ID persona	No	ID
nombre persona	No	nombre
dirección	Si	dirección

llave candidata: (*ID persona*)
llave primaria: (*ID persona*)
acceso frecuente: (*ID persona*) (*nombre persona*)

Figura 4.16. Modelo de tabla para *Persona*.

```
CREATE TABLE Persona
        ( ID-persona        ID          not null,
          nombe-persona     char(30)    not null,
          dirección         char(30)    ,
        PRIMARY KEY (ID-persona));

CREATE SECONDARY INDEX nombre-índice-persona
        ON Persona (nombre-persona)
```

Figura 4.17. Código SQL correspondiente a la tabla *Persona*.

4.6.2.3 Diseño de Asociaciones Binarias en Tablas

En general, una asociación puede o no hacer corresponder a una tabla. Depende del tipo y multiplicidad de la asociación y de las preferencias del diseñador de la base de datos, según la extensibilidad, número de tablas, y el rendimiento deseado.

Asociación "muchos-muchos": Siempre corresponden a tablas distintas, satisfaciendo la tercera forma normal. La primera llave para ambas clases y cualquier atributo de liga se convierten en atributos de la tabla de asociación.

Ejemplo: El modelo de objetos para muchas *personas* que trabajan en varias *compañías* ganando diferentes *salarios*, se muestra en la Figura 4.18. La tabla para las clase *compañía* se muestra en la Figura 4.19, mientras que la tabla para la asociación *trabaja* se muestra en la Figura 4.20. (La tabla para la clase *persona* es la misma que en la Figura 4.16.)

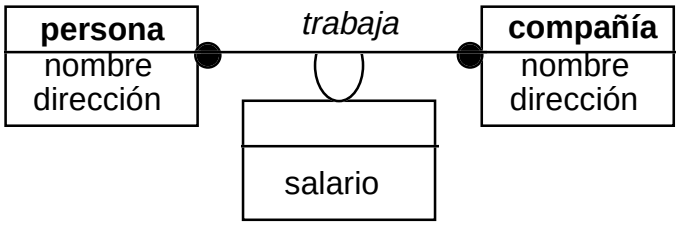


Figura 4.18. Modelo de objetos para muchas *personas* que pueden trabajar en muchas *compañías*, y ganando diferentes *salarios*.

Nombre Atributo	Nulo?	Dominio
ID compañía	No	ID
nombre compañía	No	nombre
dirección	Si	dirección

llave candidata: (*ID compañía*)
llave primaria: (*ID compañía*)
acceso frecuente: (*ID compañía*) (*nombre compañía*)

Figura 4.19. Tabla *compañía* correspondiente al modelo de objetos para la clase *compañía*.

Nombre Atributo	Nulo?	Dominio
ID compañía	No	ID
ID persona	No	ID
salario	Si	salario

llave candidata: (*ID compañía*, *ID persona*)
llave primaria: (*ID compañía*, *ID persona*)
acceso frecuente: (*ID persona*) (*ID compañía*)

Figura 4.20. Tabla *trabaja* correspondiente al modelo de objetos para la asociación *trabaja*.

Los atributos *ID compañía* y *ID persona* se combinan para formar la única llave candidata para la tabla *trabaja*. En general, una asociación puede ser atravesada de ambos lados, entonces *ID compañía* y *ID persona* pueden accesarse frecuentemente. Una tabla de asociación siempre asigna las llaves externas de los objetos a "*no nulo*", por definición, y una liga entre dos objetos requiere

que ambos objetos sean conocidos. Si un par de objetos no tienen una liga, se omite la entrada en la tabla de asociación.

Asociaciones "1-muchos": Existen dos opciones para el diseño de las tablas. Se puede crear una tabla distinta para la asociación o poner una llave externa en la tabla para la clase "muchos".

Las ventajas de juntar la asociación a la clase son las siguientes:
• menos tablas,
• mejor rendimiento al tener menos tablas;
mientras que las desventajas de juntar la asociación a la clase son las siguientes:
• menor rigor de diseño: las asociaciones son entre objetos independientes de igual peso sintáctico,
• extensibilidad reducida: es difícil obtener la multiplicidad correcta las primeras veces. Las asociaciones "1-1" y "1-muchos" pueden externalizarse, pero las asociaciones "muchos-muchos" deben externalizarse,
• mayor complejidad: una representación asimétrica de la asociación complica la búsqueda y la actualización.

Ejemplo: El modelo de objetos para personas que pueden trabajar en una sola compañía ganado un salario. El primer diseño con distintas asociaciones se muestra en la Figura 4.21. Las tablas para las clases *persona* y *compañía* son iguales a las de la Figura 4.16 y 4.19, mientras que la asociación *trabaja* se muestra en la Figura 4.22 (tabla similar a la Figura 4.20, pero con diferentes llaves).

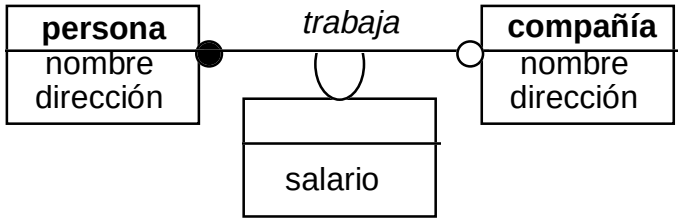


Figura 4.21. Modelo de objeto para muchas personas que solo pueden trabajar en una compañía, y ganando un salario.

Nombre Atributo	Nulo?	Dominio
ID compañía	No	ID
ID persona	No	ID
salario	Si	salario

llave candidata: (*ID persona*)
llave primaria: (*ID persona*)
acceso frecuente: (*ID persona*) (*ID compañía*)

Figura 4.22. Tabla *trabaja* correspondiente al modelo de objetos para la asociación *trabaja*, con distintas asociaciones.

El segundo diseño, con llaves externas enterradas en las tablas para las clases *persona* se muestran en la Figura 4.23, mientras que la tabla para *compañía* es similar a la Figura 4.19.

Nombre Atributo	Nulo?	Dominio
ID persona	No	ID
nombre persona	No	nombre
dirección	Si	dirección
ID compañía	Si	ID
salario	Si	salario

llave candidata: (*ID persona*)
llave primaria: (*ID persona*)
acceso frecuente: (*ID persona*) (*nombre persona*) (*ID compañía*)

Figura 4.23. Tabla *persona* correspondiente al modelo de objetos para la clase *persona*, con llaves externas enterradas.

Ejemplo: Todas las personas deben ser empleadas, como se muestra en la Figura 4.24. No hay beneficios de tener una tabla de asociación distinta.

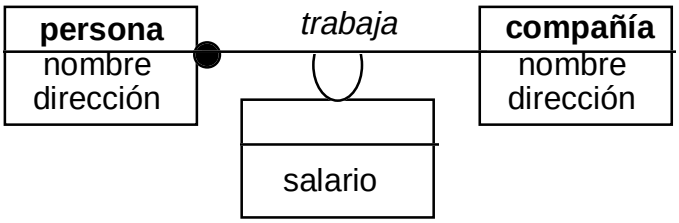


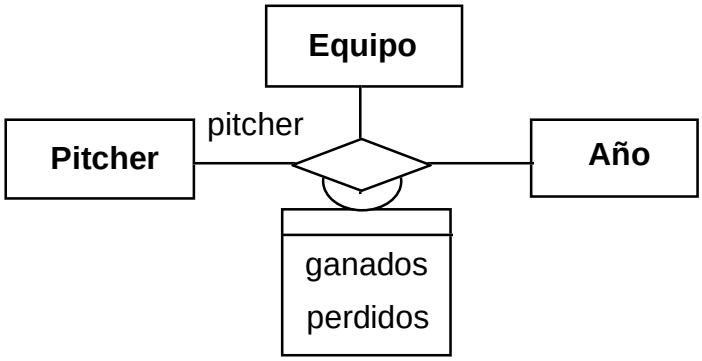
Figura 4.24. Modelo de objeto para muchas personas que trabajar en una compañía, ganando un salario.

El único cambio en el modelo de tabla para el caso de las llaves externas enterradas es que *ID compañía* no puede ser nula.

Asociación "1-1": se puede diseñar en una tabla de objetos, o incluso combinar los objetos y la asociación en una sola tabla., pero hay que tener cuidado, ya que podría violar la tercera forma normal.

4.6.2.4 Diseño de Asociaciones Ternarias en Tablas

Ejemplo: Modelo de objetos para un *pitcher* que juega en un *equip*o durante cierto *año* con estadística sobre juegos ganados y perdidos, como se muestra en la Figura 4.25. Las tablas para las diferentes clases son similares a las de los otros ejemplos. Lo único que cambia es la tabla para la asociación ternaria, como se muestra en la Figura 4.26. Existe una tabla para cada clase participante en la asociación ternaria incluyendo clases triviales, como *año*. Aunque la tabla *Persona* se refiere al *ID persona*, la tabla ternaria se refiere al *ID pitcher*, reflejando el rol de pitcher que asume la persona. Si la tabla *Año* fuese solo un par *ID año* y *año*, se podría hacer una mínima optimización, como descartar la tabla *Año* y reemplazar *ID año* y *año* en la tabla ternaria.



Llave candidata: (*ID pitcher, ID equipo, ID año*)

Figura 4.25. Modelo de objeto para un *pitcher* que juega en un *equip*o durante cierto *año* con estadística sobre juegos ganados y perdidos.

Nombre Atributo	Nulo?	Dominio
ID pitcher	No	ID
ID equipo	No	ID
ID año	No	ID
ganados	Si	juegos
perdidos	Si	juegos

llave candidata: (*ID pitcher, ID equipo, ID año*)
llave primaria: (*ID pitcher, ID equipo, ID año*)
acceso frecuente: (*ID pitcher*) (*ID equipo*) (*ID año*)

Figura 4.26. Tabla de asociación ternaria correspondiente al modelo de objetos.

4.6.2.5 Diseño con Calificativo en la Asociación

Ejemplo: Una compañía tiene muchas personas sirviendo como administradores (oficina). La mayoría de las oficinas dentro de la compañía son administradas por una persona, como el presidente o el tesorero. Una persona puede tener varias oficinas combinadas, incluyendo en varias compañías diferentes. El modelo de objetos se muestra en la Figura 4.27.
Las tablas de *Compañía* y *Persona* son similares a los ejemplos anteriores. La tabla de asociación tiene tres atributos: una llave primaria para *compañía*, una llave primaria para *persona*, y el calificador *oficina*. Ninguno de estos atributos puede ser nulo, ya que todos son partes esenciales de la asociación calificada. Los tres atributos deben aparecer en la llave primaria y llaves candidatas ya que la asociación es "muchos-muchos" después del calificativo. El modelo de tabla para la asociación calificada se muestra en la Figura 4.28.

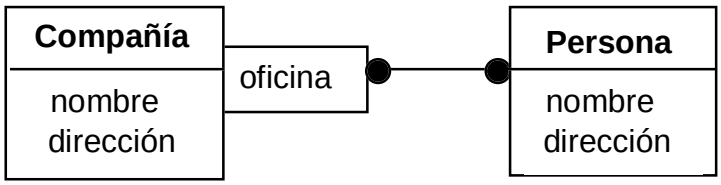


Figura 4.27. Modelo de objeto para muchas personas sirviendo como administradores (oficina) de varias compañías.

Nombre Atributo	Nulo?	Dominio
ID persona	No	ID
ID compañía	No	ID
oficina	No	nombre oficina

llave candidata: (*ID compañía*, *ID persona*, *oficina*)
llave primaria: (*ID compañía*, *ID persona*, *oficina*)
acceso frecuente: (*ID compañía*) (*ID persona*, *oficina*)

Figura 4.28. Tabla de asociación calificada correspondiente al modelo de objetos.

4.6.2.6 Diseño de Generalización en Tablas

Hay diferentes enfoques para el diseño de herencia en la base de datos.

• Herencia Sencilla

Ejemplo: Las subclases *Bomba* y *Calentador* heredan de la superclase *Equipo*, como se muestra en la Figura 4.29.

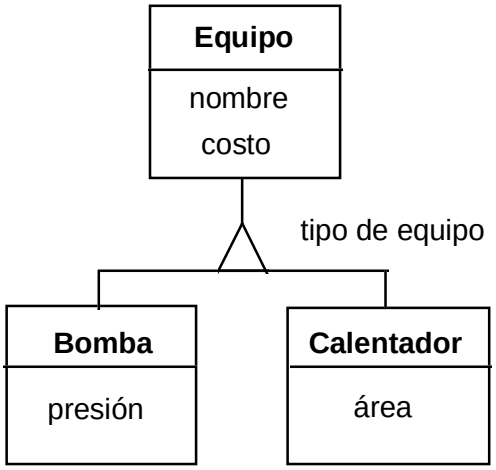


Figura 4.29. Modelo de objeto para *Bomba* y *Calentador* como subclases de *Equipo*.

4.6.2.6.1 Una tabla por superclase y subclases

Para el ejemplo anterior de herencia se crean tablas para cada superclase y subclase. La tabla para *Equipo* se muestra en la Figura 4.30, mientras que las tablas para *Bomba* y *Calentador* se muestran en la Figura 4.31 y 4.32, respectivamente.

Nombre Atributo	Nulo?	Dominio
ID equipo	No	ID
nombre equipo	No	nombre
costo	Si	dinero
tipo equipo	No	tipo equipo

llave candidata: (*ID equipo, nombre equipo*)
llave primaria: (*ID equipo*)
acceso frecuente: (*ID equipo*) (*nombre equipo*)

Figura 4.30. Tabla para la superclase *Equipo*.

Nombre Atributo	Nulo?	Dominio
ID equipo	No	ID
presión	Si	presión

llave candidata: (*ID equipo*)
llave primaria: (*ID equipo*)
acceso frecuente: (*ID equipo*)

Figura 4.31. Tabla para la subclase *Bomba*.

:

Nombre Atributo	Nulo?	Dominio
ID equipo	No	ID
área	Si	área

llave candidata: (*ID equipo*)
llave primaria: (*ID equipo*)
acceso frecuente: (*ID equipo*)

Figura 4.32. Tabla para la subclase *Calentador*.

La identidad del objeto a través de la generalización es preservada por el uso de un identificador. Aunque el enfoque es lógico y extensible, requiere muchas tablas, y el acceso de superclase a subclases puede ser lento:
• el usuario da el nombre del equipo
• se encuentra la fila de *equipo* que corresponde al nombre del equipo
• se saca el *ID equipo* y *tipo de equipo* para la fila
• se va a la tabla de subclases indicada por *tipo de equipo*, y se encuentra la fila de la subclase con el mismo identificador de *equipo*.

4.6.2.6.2 Solamente tablas para subclases

Para el ejemplo anterior de herencia se crean tablas solamente para las subclases. La tabla para *Bomba* y *Calentador* se muestran en la Figura 4.33 y 4.34, respectivamente.

Nombre Atributo	Nulo?	Dominio
ID equipo	No	ID
nombre equipo	No	nombre
costo	Si	dinero
presión	Si	presión

llave candidata: (*ID equipo*) (*nombre equipo*)
llave primaria: (*ID equipo*)
acceso frecuente: (*ID equipo*) (*nombre equipo*)

Figura 4.33. Tabla para la subclase *Bomba*.

Nombre Atributo	Nulo?	Dominio
ID equipo	No	ID
nombre equipo	No	nombre
costo	Si	dinero
área	Si	área

llave candidata: (*ID equipo*) (*nombre equipo*)
llave primaria: (*ID equipo*)
acceso frecuente: (*ID equipo*) (*nombre equipo*)

Figura 4.34. Tabla para la subclase *Calentador*.

Se elimina la tabla de superclase y se replican todo sus atributos en cada tabla de subclase. Se puede usar este enfoque si la subclase tiene muchos atributos y la superclase sólo algunos, y la aplicación sabe cual subclase buscar. Se respeta la tercera forma normal, aunque el enfoque anterior es más preferible.

4.6.2.6.3 Solamente tablas para superclases

Para el ejemplo anterior de herencia se crean tablas solamente para la superclase. La tabla para *Equipo* se muestra en la Figura 4.35.

Nombre Atributo	Nulo?	Dominio
ID equipo	No	ID
nombre equipo	No	nombre
costo	Si	dinero
tipo equipo	No	tipo equipo
presión	Si	presión
área	Si	área

llave candidata: (*ID equipo*) (*nombre equipo*)
llave primaria: (*ID equipo*)
acceso frecuente: (*ID equipo*) (*nombre equipo*)

Figura 4.35. Tabla para la superclase *Equipo*.

Todos los atributos de las subclases se llevan al nivel de la superclase. Cada récord en la tabla de superclase usa atributos correspondientes a una subclase; los otros atributos son nulos. La tabla viola la tercera forma normal. *ID equipo* o *nombre de equipo* es la llave primaria, pero los valores de atributo también dependen del *tipo de equipo*. Puede se un enfoque útil si existen sólo tres o cuatro subclases con pocos atributos.

• Herencia Múltiple

La mejor forma de manejar herencia múltiple de clases disjuntas es usar el primer enfoque, una tabla por superclase y una tabla por subclase.
La mejor forma de manejar herencia múltiple de clases no-disjuntas (entrelazadas) es usar una tabla para la superclase, una tabla para cada subclase, y una tabla para la relación de generalización.

4.6.3 Resumen de Mapas entre Modelo de Objeto y Reglas de Mapas de Tablas

Mapa de clases de objetos a tablas:

• Cada clase corresponde a una o más tablas, y una tabla puede corresponder a una o más clases si éstas están conectadas con asociaciones "1-1" o "1-muchos".

Mapa de asociaciones a tablas:

• Cada asociación "muchos-muchos" corresponde a una tabla distinta.
• Cada asociación "1-muchos" corresponde a distintas tablas o puede ser enterrada como una llave externa en la tabla para la clase "muchos".
• Cada asociación "1-1" corresponde a tablas distintas o puede ser enterrada como una llave externa en cualquiera de las dos clases.
• Para las asociaciones "1-1" y "1-muchos", se puede combinar los objetos y la asociación en una sola tabla.
• Los nombres de rol se incorporan como parte del nombre del atributo de la llave externa.
• Asociaciones n>2 corresponden a distintas tablas.
• Asociaciones calificadas corresponden a tablas distintas con tres atributos al menos, la llave primaria para cada clase relacionada y el calificador.
• Agregación sigue las mismas reglas de la asociación

Mapa de herencia sencilla a tablas

• La superclase y cada subclase corresponden a una tabla.
• Sin tabla de superclase, atributos de superclase se replican para cada subclase.
• Sin tabla de subclases, todos los atributos de subclase se llevan al nivel de la superclase.

Mapa de herencia múltiple disjunta a tablas:

• La superclase y cada subclase corresponden a una tabla

Mapa de herencia múltiple no-disjunta a tablas:

• La superclase y cada subclase corresponden a una tabla. La relación de generalización también corresponde a una tabla.