3. Diseño

Durante el diseño se decide el ambiente de computación donde se va implementar el sistema (*diseño de sistema*), y se refinan los modelos desarrollados durante el análisis (*diseño de objetos*).

3.1 Diseño de Sistema

Diseño de Sistema: Estrategia de alto nivel para implementar el sistema. Incluye decisiones sobre la organización del sistema en subsistemas, la asignación de subsistemas en componentes de hardware y software, y decisiones conceptuales mayores.

Arquitectura del sistema: Organización general del sistema.

Visión general del diseño del sistema

Durante la etapa del análisis del sistema se enfoca en *qué* debe hacerse, mientras que durante la fase del diseño del sistema se toma decisiones sobre *cómo* se va a hacer.

Decisiones durante el diseño de sistema:
1) Dividir el sistema en subsistemas
2) Identificar concurrencia
3) Asignar subsistemas a procesadores y tareas
4) Manejo de almacenamiento de datos
5) Manejo de recursos globales
6) Escoger la implementación de control en software
7) Manejo de condiciones de borde
8) Decidir entre distintas prioridades
9) Arquitecturas

3.1.1 Dividir el sistema en subsistemas

Subsistema: Cada componente mayor del sistema, es agrupado según aspectos comunes. El subsistema no es un objeto o una función sino un paquete de clases, asociaciones, operaciones, eventos, y restricciones que tienen como grupo una interfaz bien definida con otros subsistemas, usualmente identificados por los *servicios* que proporcionan. El subsistema especifica sus interacciones con otros subsistemas, pero no como están implementados internamente. Cada subsistema puede diseñarse independiente sin afectar a otros subsistemas.

Ejemplo: el sistema de archivos dentro del sistema operativo es un subsistema, que comprende a su vez otros subsistemas, como el manejo de memoria.

Servicio: grupo de funciones relacionadas que comparten un propósito común, como proceso I/O, dibujar figuras, o computar aritmética.

La relación entre subsistemas puede ser de *cliente-servidor* o *igual-a-igual*.

cliente-servidor: el cliente llama al servidor, que ejecuta algún servicio y devuelve un resultado. El cliente debe saber la interfaz del servidor, pero el servidor no tiene por qué saber las interfaces de sus clientes.

igual-a-igual: cualquier subsistema puede llamar a otros susbsistemas, y se deben conocer sus interfaces. Las secuencias de comunicación pueden ser complejas. Es más fácil buscar una descomposición de *cliente-servidor*, ya que una interacción de un lado es más fácil de construir y de comprender, que una de dos lados.

La descomposición de un sistema en subsistemas puede ser organizada como una secuencia de *capas* horizontales o *particiones* verticales.

Capas

Un sistema en capas es un conjunto de subsistemas construido en término de subsistemas inferiores. Los objetos en cada capa pueden ser independientes, aunque puede haber alguna correspondencia entre capas. El conocimiento entre capas va en un solo sentido, donde un subsistema sólo conoce las capas por debajo pero no las de arriba. Por ejemplo, la relación *cliente-servidor* existe entre capas bajas (servidores) y capas altas (clientes).

Ejemplo: En un sistema gráfico las ventanas están hechas de operaciones de pantalla, que están implementadas usando operaciones de *pixels* que se ejecutan como operaciones de entrada y salida (I/O). Cada capa puede tener su propio conjunto de clases y operaciones, y puede ser implementada por medio de clases y operaciones de más bajo nivel. En la figura 3.1 se muestra las diferentes capas del sistema gráfico para las librerías **Motif**.

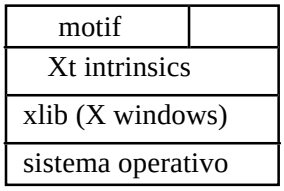


Figura 3.1. Diagrama de capas para el sistema gráfico de Motif.

Las arquitecturas en capas pueden ser *cerradas* o *abiertas*.

Arquitectura cerrada: cada capa esta construida sólo por medio de capas inmediatamente más bajas. Esto reduce la dependencia entre capas y facilita futuros cambios.

Arquitectura abierta: cada capa puede usar partes de cualquier capa de nivel más bajo, que puede resultar en código más eficiente y compacto, pero es menos resistente a cambios.

Usualmente sólo las capas de arriba y abajo se especifican en la descripción del problema:
• Las capas de arriba definen el sistema deseado.

• Las capas de abajo son los recursos existentes (hardware, sistema operativo, librerías).

Si la diferencia entre las diferenbtes capas es muy grande, el diseñador debe introducir capas intermedias para reducir la brecha entre el sistema deseado y los recursos existentes. Un sistema construido en capas puede ser implementado en diferentes plataformas (hardware/software) reescribiendo solo las capas intermedias, por lo cual es bueno introducir por lo menos una capa de abstracción entre la aplicación y los servicios del sistema operativo o hardware.

Particiones

Particiones dividen verticalmente el sistema en subsistemas independientes o conectados débilmente, cada uno ofreciendo otro tipo de servicios.

Ejemplo: Un sistema operativo incluye un sistema de archivos, control de proceso, manejo de memoria virtual, y control de dispositivos. Los subsistemas pueden saber de los otros, pero su conocimiento no es profundo para que no haya dependencias.

Un sistema puede ser descompuesto usando capas y particiones a la vez. Las capas pueden ser divididas en particiones y las particiones en capas.

Ejemplo: Diagrama de bloque de una aplicación que incluye un paquete de simulación, control de usuario, y gráficos interactivos. La Figura 3.2 muestra el diagrama de capas y particiones.

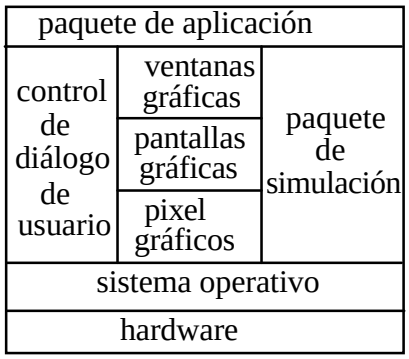


Figura 3.2. Diagrama de capas y particiones para un paquete de aplicación que incluye un control de usuario, aspectos gráficos y un paquete de simulación.

Topología del Sistema

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

Ejemplo: Muchas computaciones pueden están definidas secuencialmente, e.g. compiladores, mientras que otros sistemas pueden estar organizados en forma de estrellas, donde un subsistema maestro controla todas las interacciones con otros subsistemas.

3.1.2 Identificar Concurrencia

Una meta importante del diseño de sistema es identificar cuales objetos deben estar activos concurrentemente y cuales tienen actividades secuenciales. El modelo dinámico es una guía para identificar dicha concurrencia. Dos objetos son por naturaleza concurrentes si pueden recibir eventos al mismo tiempo y no interactuan o están sincronizados entre si en ese momento.

Ejemplo: El motor y los controles del ala de un avión deben operar concurrentemente (si no totalmente independientes). Subsistemas independientes son deseados ya que pueden ser asignados a diferentes unidades de harware sin costo de comunicación.

Ejemplo ATM: uno de los requisitos es que cada máquina continúe trabajando localmente en el evento de una falla del sistema central, entonces sería necesario incluir un procesador en cada máquina de ATM con un programa de control completo.

Hilo de Control: Se define como el camino dentro de un conjunto de diagramas de estado en el cual un solo objeto es activo a la vez. Un hilo queda dentro de un diagrama de estado hasta que el objeto manda un evento a otro objeto, esperando un evento de regreso. El hilo de control se transfiere al recibidor del evento, hasta que este lo regrese al objeto original. El hilo se divide si el objeto manda un evento y continua ejecutando. Examinando los diagramas de estado de objetos y el intercambio de eventos, muchos objetos pueden incluirse en un solo hilo de control.

Tarea: Implementación del hilo de control como un proceso en el sistema

Ejemplo ATM: mientras el banco verifica una cuenta o procesa una transacción, la máquina ATM está inactiva. El ATM está controlado directamente por la computadora central y el objeto ATM puede unirse con el objeto transacción del banco en una sola tarea.

3.1.3 Asignar Subsistemas a Procesadores y Tareas

Cada subsistema concurrente debe ser asignado a una unidad de hardware, donde se deben seguir los siguientes pasos:
1) Estimar los requisitos de recursos
2) Balancear entre hardware y software
3) Asignar las tareas a los procesadores
4) Determinar la conectividad física

3.1.3.1 Estimar los Requisitos de Recursos

La decisión de usar múltiples procesadores se basa en la necesidad de lograr un más alto rendimiento que el de un solo procesador. El número de procesadores depende del volumen de la computación y la velocidad de las máquinas. El diseñador del sistema debe estimar los requisitos de procesamiento, computando la carga continua como el producto del número de transacciones por segundo y el tiempo requerido para procesar cada transacción. La cantidad de capacidad de exceso depende de la tasa aceptable de fallas por faltas de recurso.

Ejemplo: Un sistema de radar militar genera muchos datos en muy poco tiempo para ser manejados por un solo procesador. Los datos deben ser tratados por máquinas con múltiples procesadores.

3.1.3.2 Balancear entre Hardware y Software

El hardware puede ser visto como una forma de software rígida pero optimizada. Se debe considerar compatibilidad, costo, rendimiento,y flexibilidad para cambios futuros.

Subsistemas son implementados en hardware por dos razones principales:
• El hardware existente provee la funcionalidad exacta requerida, siendo más fácil comprarlo que implementarlo en software. Por otro lado, piezas como sensores sólo existen en hardware.
• Cuando un mayor rendimiento es necesario que el proporcionado por un procesador general, se puede recurrir a procesadores especializados, como los que que ejecutan FFT (Fast Fourier Transform), que son muy usados en procesamiento de señales.

3.1.3.3 Asignar las Tareas a los Procesadores

Las tareas deben asignarse a los procesadores para satisfacer necesidades de rendimiento y minimizar comunicación entre procesadores. En particular:
• Algunas tareas se requieren en algún lugar físico, para controlar hardware o para permitir operaciones independientes o concurrentes, e.g. una estación de ingeniería necesita que su propio sistema operatvio permita su operación cuando la red no funcione.
• El tiempo de respuesta o flujo de información excede la banda de comunicación entre la tarea y una pieza de hardware, e.g. gráfica de alta rendimiento requiere controladores estrechamente entrelazados (*tightly-coupled*) por la alta generación de datos.
• Tasas de computación son demasiado grandes para un solo procesador, y deben ser divididos entre procesadores. Subsistemas que interaccionan más deben ser asignados a un mismo procesador para minimizar el costo del procesamiento.

3.1.3.4 Determinar la Conectividad Física

Luego de determinar el tipo y número de procesadores, el diseñador del sistema debe escoger el tipo de conexiones entre ellas:
• Escoger la topología conectadando los procesadores o sistemas de cómputo. Las asociaciones en el modelo de objeto pueden corresponder a conexiones físicas, al igual que relaciones cliente-servidor en el modelo funcional. Algunas conexiones pueden ser indirectas, pero el diseñador debe minimizar el costo de la comunicación.
• Escoger la topología entre unidades repetidas. Si varias copias de un tipo particular de unidad o grupo de unidades es incluido por razones de rendimiento, su topología debe estar especificada. El modelo de objeto y funcional no son guías, ya que el uso de múltiples unidades es primordialmente una optimización de diseño no requerida en el análisis. La topología de unidades repetidas puede tener un patrón regular, linear, matriz, árbol, o estrella.
• Escoger la forma de los canales de conexión y los protocolos de comunicación, e.g. la interacción puede ser sincrónica o asincrónica. El ancho de banda y latencia de los canales deben ser estimados.

También las conexiones lógicas deben analizarse para mejorar el rendimiento del sistema.

Ejemplo: las unidades pueden ser tareas dentro de un solo sistema operativo conectado por medio de llamadas de comunicación entre procesos, e.g. intercomunicación entre procesos (IPC), que funciona de forma más lenta que las llamadas de subrutina dentro del mismo programa. En tal caso se debe integrar las diferentes tareas en una sola, y las conexiones deben hacerse como simples subrutinas.

3.1.4 Manejo de Almacenamiento de Datos

El almacenamiento de datos interno y externo provee una buena separación entre subsistemas con interfaces bien definidas. Cada almacenamiento de datos puede combinar estructura de datos, base de datos implementados en memoria, o dispositivos secundarios. Diferentes tipos de almacenamiento proporcionan diferentes costos, capacidad, y tiempo de acceso:
• Los archivos son simples pero sus operaciones son de bajo nivel, y se debe incluir código adicional para proveer un buen nivel de abstracción, ya que implementaciones varían en diferentes computadoras.
• Las Bases de Datos son otro tipo de almacenamiento de datos, existiendo de diferentes tipos: jerárquicos, en red, relacionales, orientados a objetos. Los sistemas de manejos de base de datos (DBMS) trata de organizar el acceso a datos de forma más eficiente, y están desarrollados independientes de las diferentes plataformas. El problema del DBMS es que sus interfaces son complejas, y muchas de ellas no se integran bien con los lenguajes de programación.

3.1.5 Manejo de Recursos Globales

El diseñador del sistema debe identificar los recursos globales y determinar los mecanismos para controlar su acceso, e.g. unidades físicas como procesadores, controladores, o espacio de disco, o nombres lógicos, como archivos o clases.

Los recursos pueden ser físicos o lógicos:
• recursos físicos se pueden controlar estableciendo un protocolo de acceso dentro del sistema concurrente.
• recursos lógicos se deben controlar evitando acceso conflictivo, en especial nombres.

3.1.6 Escoger la Implementación de Control en Software

Durante la fase del análisis todas las interacciones son mostradas como eventos entre objetos. El control de hardware proviene del análisis, pero el diseñador debe escoger entre varias formas de implementación de control en software:
1) Sistemas impulsados por procedimientos,
2) Sistemas impulsados por eventos, y
3) Sistemas concurrentes.

3.1.6.1 Sistemas Impulsados por Procedimientos

En los sistemas secuenciales impulsados por procedimientos, el control reside dentro del código del programa. El procedimiento hace un pedido de entrada externa y luego espera; cuando la entrada llega, el control se resume dentro del procedimiento que hizo la llamada. El lugar del *contador* del programa y la *pila* del procedimiento definen el estado del sistema. La mayor ventaja de este control es que puede implementarse con un lenguaje convencional. La desventaja es que requiere que la concurrencia natural en el objeto corresponda a un flujo secuencial de control. El diseñador debe convertir eventos en operaciones entre objetos, donde una operación corresponde a un par de eventos (uno en cada objeto, e.g. entrada y salida). Entradas asincrónicas no se pueden acomodar bien en este paradigma. La mayoría de lenguajes orientados a objetos, e.g. Smalltalk, C++, son procedurales y la frase "pasoo de mensajes" se refiere a llamadas de procedimientos sin apoyo concurrente.

3.1.6.2 Sistemas Impulsados por Eventos

El control reside en un despachador o monitor provisto por el lenguaje, subsistema, o sistema operativo. Los procedimientos de la aplicación se añaden a los eventos y son llamados por el despachador cuando el evento ocurre. Los procedimientos devuelven control al despachador, y el estado del programa no se puede preservar usando el *contador* del programa o *pilas*. Los procedimientos deben usar variables globales para mantener el estado del programa, o el despachador debe mantener tal estado. El control impulsado por eventos es mas difícil de implementar con lenguajes normales que el control impulsado por procedimientos, pero es más flexible, permitiendo más patrones de control, simulando procesos cooperativos dentro de una tarea con un solo hilo de control, e.g. X-windows. Se debe usar un sistema impulsado por eventos si es posible ya que el control es más modular y se maneja condiciones de error de mejor forma.

3.1.6.3 Sistemas Concurrentes

El control reside simultaneamente en varios objetos, como diferentes tareas. Cada tarea puede esperar entradas, pero otras pueden continuar ejecutándose. El sistema operativo usualmente provee mecanismos de colas de eventos para guardarse mientras la tarea se está ejecutando, resolviendo conflictos entre tareas. Ninguno de los mayores lenguajes orientados a objetos apoya este control concurrente de forma directa, sólo a través del sistema operativo.

3.1.7 Manejo de Condiciones de Borde

Aunque el mayor esfuerzo en el diseño es en el estado continuo del sistema, se debe considerar las condiciones de borde:

• Inicialización: se debe inicializar los datos constantes, parámetros, variables globales, tareas, y posiblemente la propia jerarquía de clases. Es más dificil inicializar un sistema secuencial que uno concurrente.

• Terminación: es más simple que la inicialización, ya que los objetos internos pueden ser simplemente abandonados. La tarea debe dejar de utilizar cualquier recurso externo que tenga reservado. En un sistema concurrente una tarea puede notificar a otras tareas de su terminación.

• Fallas: puede existir una terminación inesperada del sistema, que puede ocurrir por errores del usuario, agotamiento de recursos, o por fallas externas (e.g. fallas de hardware). Un buen diseño considera posibles fallas, incluyendo errores del programa, y debería imprimir o guardar la máxima información sobre el error cuando este ocurra.

3.1.8 Decidir entre Distintas Prioridades

Se requiere escoger a veces entre metas incompatibles y definir entre distintas prioridades durante el diseño. No hay un criterio exacto de como definir las prioridades del sistema durante el diseño.

Ejemplo: un sistema puede hacerse más rápido con memoria adicional. A veces es necesario sacrificar funcionalidad para tener listo un sistema más rapidamente. Puede que la descripción del problema especifique cuales metas son de más alta prioridad, pero también pueden existir deseos incompatibles en la descripción.

No todas las decisiones son hechas durante el diseño de sistema, pero las prioridades para hacerlas sí deben ser establecidas.

Ejemplo: los primeros juegos de video corrían en un procesador con memoria limitada, donde conservar memoria era la prioridad máxima, seguido por ejecución rápida. Actualmente, la prioridad es de mayor rapidez de ejecución sin importar el espacio de memoria necesario.

3.1.9 Arquitecturas

Hay varios prototipos de arquitecturas que son bastante comunes, cada una es útil para un tipo de sistema:
• transformación en lote (*batch*): una transformación ejecutada una sola vez sobre el conjunto entero de entradas.
• transformación contínua: transformación de datos ejecutados continuamente mientras las entradas cambian.
• interfaz interactiva: sistema dominado por interacciones externas.
• simulación dinámica: sistema que simula objetos del mundo real que evolucionan en el tiempo.
• sistemas de tiempo real: sistema dominado por restricciones estrictas de tiempo.
• administración de transacción: sistema que se ocupa con guardar y actualizar datos, a veces incluyendo acceso concurrente y distribuido.

3.1.9.1 Transformación en Lote

Es una transformación secuencial de entrada y salida, donde las entradas son dadas al principio y la meta es computar la respuesta sin interacción continua del mundo externo. El modelo de estado es trivial, y el modelo de objeto puede ser simple o complejo. Lo más importante es el modelo funcional.

Ejemplo: Un compilador, con las diferentes etapas de procesamiento son mostradas en la figura 3.3.

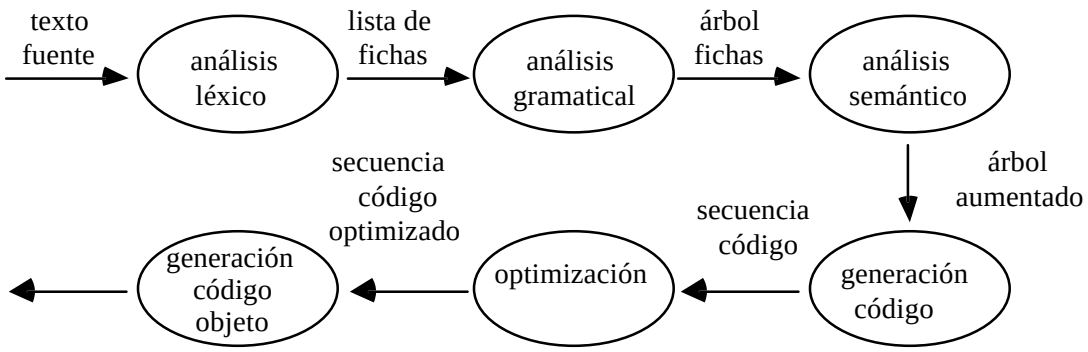


Figura 3.3. Transformación en lote de un compilador.

Los pasos a seguir son:
• Dividir la transformación general en diferentes etapas, cada una ejecutando una parte de la transformación. El diagrama del sistema es el diagrama de flujo de datos.
• Definir las clases de objetos intermedios para flujo de datos entre pares de etapas sucesivas. Cada etapa sólo sabe de los objetos a cada lado de ella, y de sus propias entradas y salidas.
• Expandir cada etapa hasta que las operaciones sean fáciles de implementar.
• Reestructurar la secuencia final para lograr una buena optimización del sistema.

3.1.9.2 Transformación Continua

Es un sistema en el cual las salidas dependen activamente de las entradas que cambian y deben ser actualizadas periódicamente. A diferencia de las transformaciones en lote, las salidas deben ser actualizadas frecuentemente. Los valores de salida son computados de forma incremental (si no sería una transformación en lote). El modelo funcional y el de objetos definen los valores a computar. El modelo dinámico es menos importante ya que la mayoría de la estructura de la aplicación es dada por el flujo estable de datos y no por interacciones discretas.

Ejemplo: procedimientos de señales, sistemas de ventanas, compiladores incrementales.

Se puede implementar como una secuencia de funciones, donde el efecto de cada cambio incremental debe ser propagado, y objetos intermedios deben guardar valores intermedios. La sincronización de valores puede ser importante en sistemas de alto rendimiento, como operaciones ejecutadas con tiempos bien definidos, con valores arribando en lugares precisos en momentos precisos.

Ejemplo: El diagrama de flujo de datos para una aplicación gráfica se muestra en la Figura 3.4.

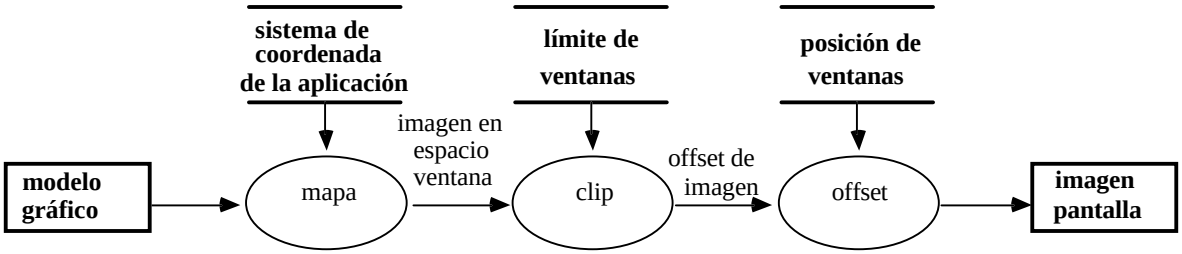


Figura 3.4. Diagrama de flujo de datos para una aplicacion gráfica con una transformación continua.

Los pasos a seguir son:
• Dibujar el diagrama de flujo de datos para el sistema, donde los actores de entrada y salida corresponden a estructuras de datos cuyos valores cambian constantemente. Los almacenamiento de datos dentro de la secuencia muestran parámetros que afectan mapas de entrada y salida.

Ejemplo: En una aplicación gráfica las figuras geométricas están definidas por coordenadas de la aplicación, correspondiendo a las coordenadas de la ventana. Las figuras son recortadas (*clip*) para estar dentro de los límites de la ventana, y finalmente cada figura tiene un desplazamiento (*offset*) de la posición de la ventana para ubicarse en la pantalla.

• Definir los objetos intermedios entre pares de etapas sucesivas como en la transformación de lote.

Ejemplo: Cada figura geométrica tiene un mapa en cada etapa de la computación.

• Diferenciar cada operación para obtener cambios incrementales en cada etapa. Propagar los efectos incrementales de cada cambio a un objeto de entrada como una serie de actualizaciones incrementales.

Ejemplo: Un cambio en la posición de una figura geométrica requiere que la imagen antigua sea borrada, su nueva posición computada y la nueva imagen desplegada. Otras figuras no son cambiadas o recomputadas.

• Añadir objetos intermedios adicionales para optimización.

3.1.9.3 Interfaz Interactiva

Es un sistema que es dominado por interacciones entre el sistema y agentes externos, como usuarios, dispositivos, u otros programas. Los agentes externos son independientes del sistema, y sus entradas no pueden ser controlados, aunque el sistema puede solicitar respuestas. Una interfaz interactiva usualmente es solo parte de una aplicación entera, que puede ser tratada de forma independiente de la parte computacional de la aplicación. Los aspectos más importantes son los protocolos de comunicación entre el sistema y los agentes externos, la sintáxis de las posibles interacciones, la presentación de salida, el flujo de control dentro del sistema, el rendimiento, y el manejo de errores. El modelo dinámico domina la interfaz interactiva. Los objetos en el modelo de objeto representan elementos de interacción, como entrada y salida, y formatos de

presentación. Un sistema interactivo se preocupa de las apariencias externas y no de los aspectos interna de la aplicación.

Ejemplo: sistema de ventanas, lenguaje de comandos para un sistema operativo, el panel de control para una simulación.

Los pasos a seguir son:
• Aislar objetos que definen la interfaz de la aplicación.
• Si es posible, usar objetos predefinidos para interactuar con los agentes externos.

Ejemplo: ventanas, menús, botones

• Usar del modelo dinámico para estructurar el programa, implementado mejor por control concurrente (multi-tarea) o control impulsado por eventos (interruptores o *callbacks*). El control impulsado por procedimientos es usado para secuencias más rígidas de control.
• Aislar los eventos físicos de los eventos lógicos. Un evento lógico puede corresponder a múltiples eventos físicos.

Ejemplo: una interfaz gráfica puede tomar una entrada de un menú, un botón, una tecla, o un archivo de comandos indirecto.

• Especificar completamente las funciones de la aplicación que son llamados por la interfaz.

3.1.9.4 Simulación Dinámica

Modela objetos del mundo real, como moléculas, naves espaciales, sistemas económicos, redes neuronales. Las metodologías tradicionales basadas en diagramas de flujo de datos son pobres representaciones de estos problemas ya que la simulación envuelve muchos objetos distintos que constantemente son actualizados, más que una sola gran transformación. La simulación es quizás el sistema más simple para diseñar usando análisis orientado a objetos. Los objetos y operaciones vienen directamente de la aplicación. El control puede ser implementado como un controlador explícito, externo a los objetos de la aplicación, simulando una máquina de estado, o los objetos pueden intercambiar mensajes entre si, similar al mundo real. A diferencia de los sistemas interactivos, los objetos internos de la simulación dinámica corresponden a objetos reales, y el modelo de objetos es el más importante y complejo. El modelo dinámico también es importante y se puede tener un modelo funcional complejo. El mayor problema en la simulación es proveer un rendimiento adecuado. Idealmente un número arbitrario de procedimientos paralelos ejecutan la simulación. En la práctica se debe estimar el costo computacional de cada ciclo de actualización. Además, los procesos continuos deben ser aproximados por medio de pasos discretos.

Los pasos a seguir son:
• Identificar los actores, los objetos activos del mundo real. Los actores tienen atributos que son periódicamente actualizados.
• Identificar los eventos discretos, correspondiendo a interacciones discretas con los objetos.
• Identificar dependencias continuas, que pueden ser actualizadas a intervalos periódicos usando técnicas de aproximación numérica.

• En general la simulación es impulsada por un ciclo con una escala de tiempo muy corto. Los eventos discretos entre los objetos pueden ser intercambiados dentro del ciclo de tiempo.

3.1.9.5 Sistemas de Tiempo Real

Es un sistema interactivo para el cual son muy críticas las restricciones del tiempo en las acciones, y donde no es tolerada la menor falla en el tiempo. Para garantizar el tiempo de respuesta, el escenario para el caso peor debe ser determinado. El diseño del sistema de tiempo real es muy complejo, y requiere manejo de interruptores, prioridad de tareas, y coordinación de múltiples procesadores. Por los severos límites de los recursos, a veces se estructura los sistemas de forma no lógica encareciendo el costo del mantenimiento.

Ejemplo: control de procesos, adquisición de datos, dispositivos de comunicación.

3.1.9.6 Administración de Transacción

El administrador de transacción en un sistema de base de datos tiene como función principal guardar y accesar información, tratando con múltiples usuarios y concurrencia. Una transacción debe ser manejada como una sola entidad sin interferencia de otras transacciones. El modelo de objetos es el más importante. El modelo funcional es menos importante ya que las operaciones tienden a ser predefinidas y se concentran en actualizar y consultar la información. El modelo dinámico muestra acceso concurrente de recursos distribuidos. Se puede utilizar un DBMS existente. El mayor problema es construir el modelo de objetos y escoger el tamaño o *granularidad* de las transacciones que se consideran indivisibles en el sistema.

Ejemplo: sistema de reservaciones de aviones, sistema de control de inventario, DBMS.

Los pasos a seguir son:
• Corresponder el modelo de objetos con la base de datos
• Determinar las unidades de concurrencia
• Determinar las unidades de transacción, los recursos que se accesan juntos
• Diseñar el control concurrente para las transacciones.

Arquitectura del Sistema ATM

El sistema ATM es un híbrido de interfaz de interacción y administrador de transacción. Las estaciones de entrada son interfaces interactivas que consisten principalmente de los modelos de objetos y dinámico. El *consorcio* y los **bancos** son principalmente un sistema de administración de transacciones distribuido. Su propósito es mantener la base de datos para la información y actualizarse sobre la red distribuida, que consiste principalmente de un modelo de objetos. La arquitectura del sistema se muestra en la Figura 3.5.

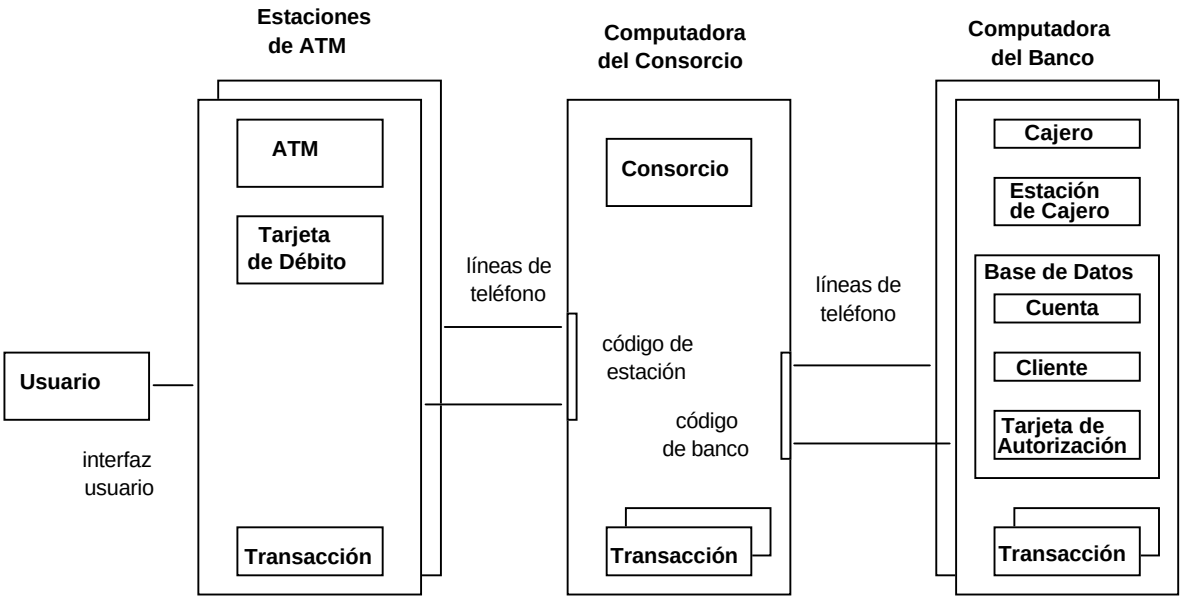


Figura 3.5. Arquitectura para el sistema ATM.

Hay tres subsistemas principales:
• las estaciones ATM
• la computadora del consorcio
• las computadoras de los bancos

La topología del sistema es de tipo estrella, el consorcio se comunica con todos los ATM y todos las computadoras de los bancos. Cada conexión es por medio de una línea telefónica. Los únicos almacenamientos de datos permanentes pertenecen a las computadoras de los bancos, ya que deben ser mantenidas por largo tiempo, y pueden ser accesadas por varias transacciones concurrentes. Cada transacción es procesada en una sola operación de *lote*, donde la cuenta es bloqueada hasta que la transacción se complete.

Existe una concurrencia natural en el sistema, ya que varios ATM pueden trabajar a la vez, aunque sólo puede haber una transacción por estación de ATM a la vez, y cada transacción requiere asistencia de la computadora del consorcio y banco. La transacción involucra varias unidades físicas. Durante el diseño cada una es una clase separada de implementación. Y aunque hay sólo una transacción por estación de ATM, pueden haber varias transacciones concurrentes por computadora de consorcio o de banco. El acceso a las cuentas es sincronizado en la base de datos.

El ATM es un diagrama de estados, impulsados por eventos, guardando eventos de entrada en colas que son procesadas una a la vez. La computadora del consorcio debe ser suficientemente grande para manejar el número máximo esperado de transacciones simultáneas, al igual que el banco que debe tener mucho espacio de disco para guardar todos los registros de transacción.

El sistema debe tener operaciones para dejar que se añadan o quiten las estaciones de ATM y las computadoras de los bancos. Cada unidad física debe protegerse de fallas en el resto de la red. La

base de datos debe proveer protección contra pérdida de datos, incluyendo fallas durante una transacción, para que ni el usuario ni el banco pierdan dinero. Que no fallen las transacciones es la más alta prioridad. La computadora del banco es la única unidad con procedimientos no triviales, pero es principalmente actualizaciones de la base de datos. La única complejidad es en el manejo de errores.

3.2 Diseño de Objetos

Diseño de objetos es un proceso de añadir detalles al análisis y hacer decisiones de implementación. El diseño de objetos determina la definición completa de las clases y las asociaciones usados en la implementación. Las clases, atributos, y asociaciones del análisis deben implementarse como estructura de datos especificas. Se debe implementar las operaciones como algoritmos, con operaciones complejas descompuestas en operaciones mas sencillas. El diseñador de objetos escoge entre diferentes formas de implementación según el tiempo de ejecución, memoria, y otras medidas de costo. Se pueden añadir objetos internos para la implementación y optimización de las estructuras de datos y algoritmos.

Vista general del diseño de objetos

Al escoger una arquitectura ya se hacen decisiones que son necesarias para la implementación del sistema, incluyendo decisiones de como corresponder los eventos a las operaciones.

Los pasos a seguir son:
1) Combinar los tres modelos
2) Diseñar los algoritmos
3) Optimizar el diseño
4) Implementar el control
5) Ajustar la herencia
6) Diseñar asociaciones
7) Representar objetos
8) Empacar en módulos

3.2.1 Combinar los Tres Modelos

En esta etapa se trata de corresponder la estructura lógica del análisis a una organización física del programa. Cada diagrama de estado describe la historia de un objeto, mientras que una transición es un cambio de estado correspondiendo a operaciones del objeto. Se puede asociar una operación con cada evento recibido por un objeto. En el diagrama de estado la acción ejecutada por una transacción depende del evento y estado del objeto. Si el mismo evento puede ser recibido por más de un objeto, el código implementando el algoritmo debe depender del estado.

Un evento mandado por un objeto puede representar una operación en otro objeto. Una acción o actividad iniciada por una transición en el diagrama de estados puede expandirse a un diagrama de flujo de datos más completo en el modelo funcional. La red de procesos dentro del diagrama de flujo de datos representa el cuerpo de la operación. El flujo de datos en el diagrama

correspondiente a valores intermedios de la operación. Los procesos en el diagrama de flujo de datos constituyen *suboperaciones* que pueden ser operaciones en el objeto original o en otros objetos.

Se debe determinar el objeto blanco (*target*) para cada operación:
• si el proceso extrae un valor del flujo de entrada, entonces el flujo de entrada es el blanco del proceso.
• si el proceso tiene un flujo de entrada y salida del mismo tipo, y si el valor de salida es una versión actualizada del flujo de entrada, entonces el flujo de entrada/salida es el blanco del proceso.
• si el proceso construye valores de salida de varios flujos de entrada, entonces la operación es una operación de clase en la clase de salida.
• si el proceso tiene una entrada o salida a un almacenamiento de datos o actor, entonces el almacenamiento de datos o actor es el blanco del proceso

3.2.2 Diseñar los Algoritmos

Cada operación del modelo funcional debe ser formulada como un *algoritmo*, que puede ser subdividido en llamadas a operaciones más sencillas

Los pasos a seguir son:
1) Escoger los algoritmos
2) Escoger estructura de datos
3) Definir clases internas y operaciones
4) Asignar responsabilidad para operaciones

3.2.2.1 Escoger Algoritmos

Se deben escoger los algoritmos para que minimicen el costo de la implementación de la operación. Muchas operaciones son suficientemente simples, que su especificación en el modelo funcional constituye un algoritmo satisfactorio.
Muchas operaciones simplemente atraviesan las asociaciones entre los objetos para sacar o cambiar los valores de los respectivos atributos.

Ejemplo: En el editor gráfico de OMTool para construir diagramas de objetos, es fácil fácil crear, editar, guardar, e imprimir. Se guarda el modelo gráfico al igual que el lógico, que es útil para chequeo semántico y para interacción con otros programas, como generadores de código y base de datos. Una *caja clase* que contiene una *lista de operaciones* que a su vez contiene un conjunto de *entradas de operación*. No hay necesidad de escribir un algoritmo para buscar la caja clase conteniendo una entrada para una operación ya que el valor es encontrado por una simple travesía de las ligas. Ver la Figura 3.6.
Se necesitan algoritmos no-triviales principalmente por dos razones:
• Para implementar funciones para la cual no es dada una especificación procedural
• Para optimizar funciones para las cuales han sido definidos algoritmos simples pero ineficientes

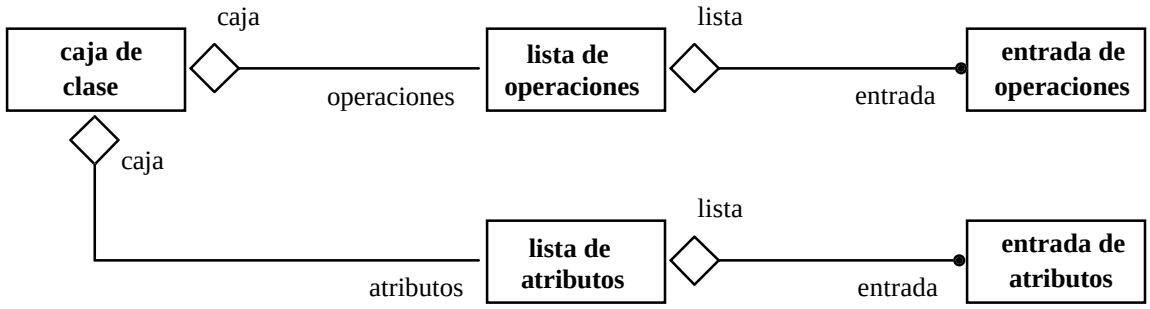


Figura 3.6. Diagrama de clases para el editor gráfico de OMTool.

Algunas funciones son especificadas como restricciones declarativas, sin definición procedural. Se debe usar algún conocimiento adicional para implementar el algoritmo

Ejemplo: "el circulo pasa por tres puntos no-colineares"

Consideraciones entre algoritmos alternativos:
• complejidad computacional: por ejemplo, la incrementación del tiempo de proceso según el tamaño de la estructura de datos.

Ejemplo: un nivel adicional de indirección es insignificante si mejora la claridad, pero es crítico ver como el tiempo de ejecución crece con el número de valores de entrada, al igual que el costo de procesar cada valor. Puede ser constante, linear, cuadrático o exponencial.

• facilidad de implementación y entendimiento: vale la pena bajar un poco el rendimiento de las operaciones no criticas si pueden ser implementadas de forma rápida con un algoritmo más simple.
• flexibilidad: la mayoría de los programas se van a extender tarde o temprano. Un algoritmo muy optimizado a veces sacrifica la facilidad de entendimiento y los posibles cambio futuros.
• refinado del modelo de objeto: si el modelo de objeto fuera estructurado de forma diferente, pudiese variar el rendimiento del sistema.

Ejemplo: se pueden computar uno por uno los diferentes elementos a dibujarse en cada ventana, o se pueden computar conjuntamente por página y luego mostrarlo en la ventana. Ver la Figura 3.7.

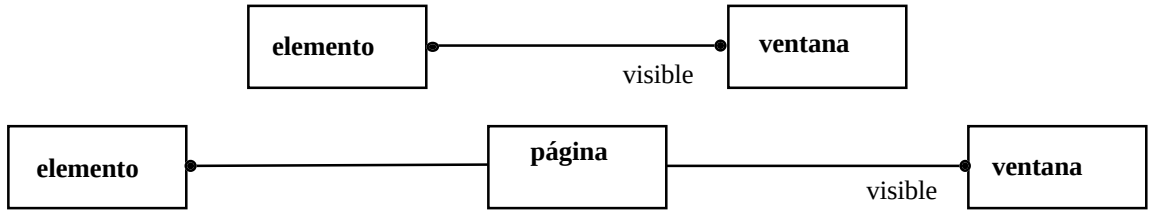


Figura 3.7. Diagrama de clases para desplegar elementos gráficos en una ventana. El de arriba se despliega elemento por elemento, mientras que el de abajo se concentra primero en una página para luego desplegarse en forma conjunta.

3.2.2.2 Escoger Estructuras de Datos

Escoger algoritmos envuelve escoger estructura de datos. Muchas implementaciones de estructura de datos son instancias de *clases contenedoras*, como listas, pilas, filas, árboles, etc.

3.2.2.3 Definir Clases Internas y Operaciones

Puede haber necesidad de definir nuevas entidades internas para complementar las entidades definidas durante el análisis.

3.2.2.4 Asignar Responsabilidad para Operaciones

Muchas operaciones tienen objetos blanco obvios, pero algunas operaciones pueden ser ejecutadas en varios lugares en el algoritmo, por uno o por varios objetos, y no son necesariamente parte de una sola clase.

Ejemplo: mover el diagrama en OMTool propaga el movimiento de objeto a objeto según sus conexiones. ¿ Cuándo se debe redibujar cada objeto; después de que es movido por otro objeto, o después de que todos los objetos son movidos ? ¿ Quién es responsable; el propio objeto o la imagen entera ? Esto es bastante arbitrario.

Durante la implementación las clases internas son bastante imprecisas en sus bordes. ¿ Cómo decidir que clase es dueña de que operación ? Cuando hay un solo objeto la decisión es fácil, pero cuando están involucrados varios objetos en la operación, la decisión es más difícil.

Se debe decidir cual objeto juega el papel más importante en la operación:
• se aplica la operación a un objeto mientras los otros objetos ejecutan una acción. En general es mejor asociar una operación con el *blanco* de una operación, y no el *iniciador*.
• la operación modifica a un objeto, mientras que los otros objetos se consultan solamente por la información que contienen. El objeto que es modificado es el *blanco* de la operación.
• la operación involucra clases y asociaciones. Si la clases y asociaciones forman una estrella alrededor de una sola clase central, entonces la clases central es el *blanco* de la operación.

3.2.3 Optimizar el Diseño

El diseño básico usa el modelo de análisis como marco de implementación. El modelo de análisis puede ser ineficiente pero debe ser semanticamente correcto, y por lo tanto puede ser optimizado para hacer la implementación más eficiente.

Posibles optimizaciones:
1) Añadir asociaciones redundantes
2) Reorganizar el orden de ejecución
3) Guardar atributos derivados

3.2.3.1 Añadir Asociaciones Redundantes

Durante el análisis no es deseado tener redundancia en las asociaciones, ya que éstas no añaden nueva información. Durante el diseño se evalúa la estructura del modelo de objeto para minimizar el costo de acceso y maximizar la conveniencia de la implementación. Las asociaciones que eran útiles durante análisis pueden no ser lo más eficiente para accesar.

Ejemplo: La operación compañía::encontrar-habilidad devuelve un grupo de personas en la compañía con una habilidad particular. Ver el diagrama de la Figura 3.8.

Figura 3.8. Diagrama de clases describiendo a personas dentro de una compañía con ciertas habilidades.

Por ejemplo, se podría preguntar por todos los empleados que hablen inglés. Si la compañía tiene 1,000 empleados, de los cuales cada uno tiene 10 habilidades en promedio, un ciclo anidado atravesaría *emplea* 1,000 veces y *tiene-habilidad* 10,000 veces. Si solo 5 empleados hablan inglés, el promedio de búsqueda es de 1/2000.
Varias mejoras son posibles. Primero *tiene-habilidad* no tiene que implementarse como una lista desordenada pero puede implementarse como una lista indexada. Tal lista puede ser atravesada de forma que el costo de la averiguación de si la persona habla ingles sea constante. Si *habla-inglés* sería representado directamente por un objeto de tipo habilidad reduciendo el número de pruebas de 10,000 a 1,000, o sea 1 por empleado.

En casos donde el número de respuestas correctas es muy bajo se puede construir un índice para mejorar el acceso a objetos que deben ser frecuentemente consultados. Se puede añadir una asociación calificada *habla lenguaje* (una asociación derivada) de *compañía* a *empleado* donde el calificativo es *lenguaje*. Esto permite accesar inmediatamente todos los empleados que hablan un lenguaje particular sin accesos perdidos. El costo es el índice, que requiere memoria adicional y debe ser actualizado cuando las asociaciones básicas son actualizadas. Si la consulta devuelve demasiados objetos, el índice puede que no mejore las cosas. Ver la Figura 3.9.

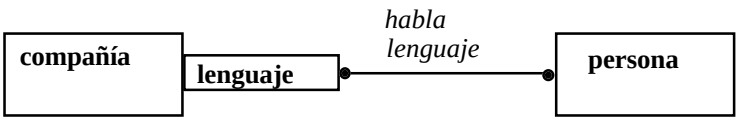


Figura 3.9. Diagrama de clases con índice según el conocimiento de un lenguaje para una persona en una compañía.

Se debe analizar el uso de los caminos en la red de asociaciones:
• Examinar cada operación y analizar cuales asociaciones son atravesadas en ambas direcciones y cuales en solo una.

Para cada operación analizar:

• Que tan frecuente se llama la operación, y que costo tiene de ejecución.
• Cual es el "fan-out" por la red. Estimar la cuenta promedio de cada asociación "muchos" encontrada en el camino. Multiplicar los "fan-outs" individuales para obtener el "fan-out" de todo el camino, que representa el número de accesos de la ultima clase. Ligas "1" no aumentan el "fan-out" aunque suben el costo de la operación.
• Cual es la fracción de aciertos de la clase final. Si la mayoría de los objetos no son los intencionados durante la travesía, entonces un ciclo anidado simple puede ser ineficiente.

Se debe proveer índices para operaciones frecuentes y costosas con aciertos bajos ya que tales operaciones son ineficientes usando una implementación de ciclos anidados.

3.2.3.2 Reorganizar el Orden de Ejecución

Luego de ajustar la estructura del modelo de objetos para optimizar las travesías frecuentes, el próximo paso es optimizar el propio algoritmo para obtener un mejor rendimiento. Un aspecto importante de la optimización es eliminar desde el inicio los caminos sin salida.

Ejemplo: Si queremos encontrar todos los empleados que hablen inglés y francés, y si hay solamente 5 empleados que hablan inglés y 100 que hablan francés, entonces es mejor buscar primero quienes hablan inglés y luego francés.

3.2.3.3 Guardar Atributos Derivados

Datos redundantes que pueden ser derivados de otros datos pueden guardarse en su forma computada para evitar recomputación. Nuevos objetos o clases pueden ser definidos para retener esta información, que debe ser actualizada si un objeto del cual depende es cambiado.

Ejemplo: los objetos y atributos derivados en el editor gráfico de OMTool se muestran en la Figura 3.10. En el diagrama de más arriba, cada caja de clase contiene una lista ordenada de atributos y operaciones, cada una representada como una cadena de texto. Dado el lugar de la caja de clase, el lugar de cada atributo se puede computar añadiendo el tamaño de todos los elementos que se sobreponen. Ya que el lugar de cada elemento se necesita frecuentemente, el lugar de cada cadena de atributo se podría computar y luego guardar. En el diagrama de más abajo, la región que contiene la lista entera de atributos está computada y guardada para que los puntos de entrada no deban ser probados contra elementos en otras cajas. Si un nuevo atributo se agrega a la lista, los lugares de los atributos restantes son desplazados por el tamaño del nuevo elemento.

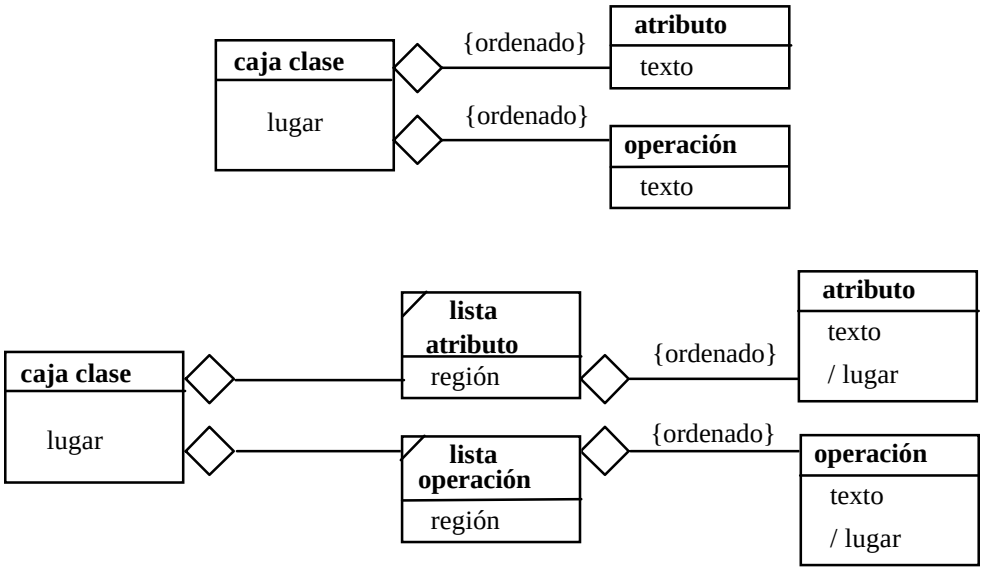


Figura 3.10. Diagrama de clases para el editor gráfico OMTool conteniendo objetos y atributos derivados

Ejemplo: Una página contiene una lista de prioridades de elementos parcialmente superpuestos. Si un elemento es movido o quitado, los elementos por debajo deben ser re-dibujados. Elementos superpuestos pueden encontrarse al buscar todos los elementos en frente del elemento quitado en la lista de prioridades de la página y compararlos al elemento quitado. Si el número de elementos es grande, el algoritmo crece linealmente con el número de elementos. La asociación *superposición* guarda los elementos que se superponen al objeto y le preceden en la lista. La asociación debe actualizarse si se añaden nuevos elementos. Ver la Figura 3.11.

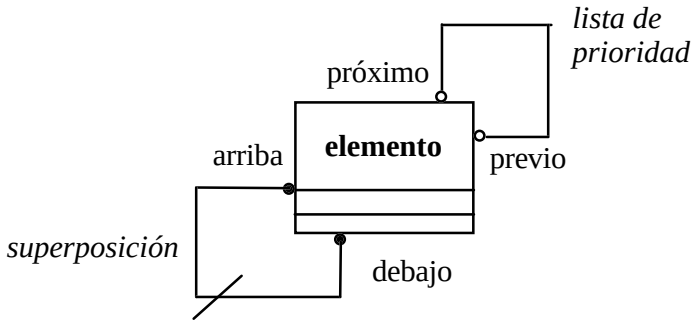


Figura 3.11. Diagrama de clases para elementos superpuestos con asociaciones derivadas.

Los atributos derivados deben actualizarse cuando los valores básicos cambian, y se pueden actualizar de las siguientes formas:

• código explícito: cada atributo derivado es definido en término de uno o más objetos fundamentales. El diseñador determina cuales atributos derivados están afectados por cada cambio de un atributo fundamental, y debe insertar código en la operación de actualización en el objeto base para actualizar explícitamente los atributos derivados que dependan de él.

• recomputación periódica: es a veces posible recomputar todos los atributos derivados periódicamente para no hacerlo después de cada cambio a un valor base. La recomputación de todos los atributos derivados puede ser más eficiente que la actualización incremental ya que algunos atributos derivados pueden depender de varios atributos base y pueden ser actualizados más de una vez en la forma incremental. Pero si sólo algunos atributos cambian, la recomputación periódica puede no ser muy práctica si existen muchos atributos.
• valores activos: valores que tienen valores dependientes. Cada valor dependiente se *registra* con el valor activo que contiene un conjunto de valores dependientes y operaciones de actualización. Una operación para actualizar el valor base causa actualizaciones de todos sus valores dependientes, pero el código no llama explícitamente las actualizaciones. Sirve para modularizar la implementación.

3.2.4 Implementar el Control

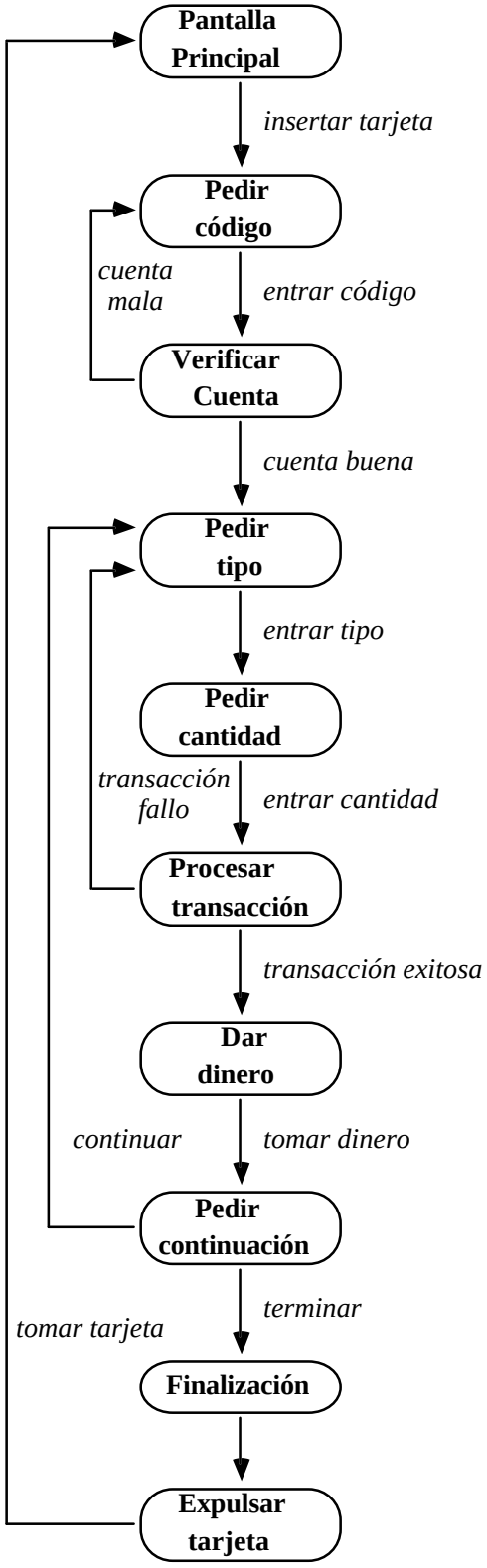
El diseñador debe refinar la estrategia para implementar los diagramas del modelo dinámico. Como parte del diseño de sistema se escogió una estrategia básica para realizar el modelo dinámico, y durante el diseño de objeto se desarrolla esta estrategia.

Alternativas para la implementación de control:
1) sistema impulsado por procedimientos
2) sistema impulsado por eventos
3) tareas concurrentes

3.2.4.1 Sistema Impulsado por Procedimientos

La forma tradicional para el control del programa, es definido explícitamente por el estado del programa. Cada transición corresponde a una oración de entrada. Después de que se lee la entrada, el programa se ramifica dependiendo de los eventos recibidos. Cada oración de entrada necesita manejar sólo los valores que se pueden recibir en ese punto. En código procedural muy anidado, los procedimientos de bajo nivel deben aceptar entradas de las que pueden no saber nada y pasarlas hacia arriba varios niveles hasta que algún procedimiento esté preparado para manejarlas. La falta de modularidad es el problema mas grande.

Una técnica para convertir diagramas de estado a código:
• identificar el camino de control principal comenzando con el estado inicial, identificar el camino por el diagrama que corresponde a la secuencia de eventos normalmente esperado. Escribir los nombres de estado en el camino como una secuencia linear, pasando a ser una secuencia de oraciones en el programa.
• Identificar los caminos alternos que se ramifican del camino principal y luego se juntan. Estos pasan a ser oraciones condicionales en el programa.
• Identificar los caminos hacia atrás que se ramifican del ciclo principal y se juntan anteriormente. Estos son ciclos en el programa. Si hay caminos múltiples hacia atrás que no se cruzan, ellos son ciclos anidados en el programa. Los caminos hacia atrás que se cruzan no son anidados.



Pseudo-código

```
hacer siempre
  desplegar pantalla principal
  leer tarjeta
  repetir
    pedir código
    leer código
    verificar cuenta
  hasta verificación de cuenta buena
    repetir
      pedir tipo de transacción
      leer tipo
      pedir cantidad
      leer cantidad
      esperar que se complete
    hasta transacción exitosa
    dar dinero
    esperar que el cliente lo tome
    pedir si quiere continuar
  hasta cliente pide terminar
  expulsar tarjeta
  esperar que el cliente la tome
```

Figura 3.12. Diagrama de estado para el ATM, y el pseudo-código correspondiente.

• Los estados y transiciones que quedan corresponden a condiciones de excepción. Pueden ser manejadas por varias técnicas, incluyendo subrutinas de error, manejo de excepción apoyado por el lenguaje, o verificación de banderas de estado.

Ejemplo ATM: en la Figura 3.12 se muestra el modelo de estado del ATM y el pseudo-código derivado de él.

Primero se identifica el camino principal de control que corresponde a leer una tarjeta, pedir al usuario información para la transacción, procesar la transacción, imprimir el recibo y expulsar la tarjeta. Los flujos alternos de control ocurren si el cliente quiere procesar más de una transacción, o si el código es malo, y en tal caso se pide al cliente probar de nuevo. El evento *cancelar* puede ser añadido o implementado como código de manejo de excepción.

Eventos de entrada dentro de un programa de un solo hilo de control es codificado con bloqueo de lecturas interactivas. (En un lenguaje de multi-tarea, los eventos de entrada pueden ser codificados con oraciones de espera para llamadas entre tarea. El sistema operativo es responsable de recibir interrupciones y ponerlas en colas.)

3.2.4.2 Sistema Impulsado por Eventos

La manera más directa de implementar el control es por medio de *máquinas de estado*.

Ejemplo: una clase "máquina de estado" puede representarse por una tabla de transiciones y acciones provistas por la aplicación. Cada instancia de objeto contendría su propias variables de estado independiente, y se llama a la máquina de estado para determinar el próximo paso o acción. (Las máquinas de estado son objetos internos del sistema que apoyan la semántica de los objetos de la aplicación.) Este enfoque permite progresar rápidamente del análisis a un prototipo del sistema al definir clases del modelo de objeto, máquinas de estado del modelo dinámico, y fragmentos de rutinas de acción, donde cada fragmento es la definición mínima de la función o subrutina.

Ejemplo: Un analizador gramatical (*parser*), como *yacc*, produce una máquina de estado explícita.

3.2.4.3 Tareas Concurrentes

Un objeto puede implementarse como una tarea en un lenguaje de programación o sistema operativo. Es el camino más general, ya que preserva la concurrencia natural de los objetos. Los eventos son implementados como llamadas entre tareas usando facilidades del lenguaje o sistema operativo.

3.2.5 Ajustar Herencia

Mientras el diseño de objetos progresa, se debe seguir los siguientes pasos:
1) Reorganizar las clases y operaciones
2) Abstraer comportamiento común

3) Delegación
4) Herencia múltiple por medio de herencia sencilla

3.2.5.1 Reorganizar las clases y operaciones

A veces la misma operación es definida en varias clases, pudiéndo ser heredada de un ancestro común. Modificando un poco las definiciones de las operaciones o clases se puede utilizar una sola operación heredada. Antes de que la herencia pueda ser usada, cada operación debe tener la misma interfaz, semántica y firma.

Ajustes posibles para incrementar posibilidad de herencia:
• Algunas operaciones pueden tener menos argumentos que otras. Los argumentos que faltan pueden añadirse aunque se ignoren.

Ejemplo: una operación de dibujar en un despliegue monocromático que no recibe parámetros de color, pero el parámetro puede ser aceptado e ignorado para consistencia con despliegue de color.

• Algunas operaciones pueden tener menos argumentos porque son casos especiales de argumentos más generales. Se debe implementar la operación especial llamando la operación general con valores de parámetros apropiados.

Ejemplo: agregar un elemento a una lista es un caso especial de insertar un elemento a una lista donde el punto de inserción está después del último elemento.

• Atributos similares en clases diferentes puede que tengan nombres diferentes. Dar a los atributos nombres iguales y moverlos a un ancestro común. También se debe estar pendiente de operaciones similares con nombres diferentes. Es muy importante la consistencia de nombres.

• Una operación puede estar definida en diferentes clases en un grupo, pero puede estar indefinida en otras clases. Se debe redefinirla en la clase ancestral común y declararla como una "no-operación" en las clases que no nos interesa.

Ejemplo: En OMTool, la operación *comenzar-editar* ubica algunas figuras, como cajas de clases, en un modo especial de dibujo permitiendo escalar de forma rápida mientras el texto es editado. Otras figuras no tiene modo especial de dibujar, y la operación *comenzar-editar* no tiene efecto.

3.2.5.2 Abstraer Comportamiento Común

Oportunidades para usar herencia no son siempre reconocidas durante la fase de análisis, y es bueno re-examinar el modelo de objetos buscando clases comunes. Cuando el comportamiento común ha sido reconocido, una superclase, posiblemente abstracta, puede ser creada para implementar los aspectos comunes, dejando solo los aspectos de especialización en las subclases. A veces es bueno abstraer una superclase aunque haya solo una subclase, para otras futuras subclases (extensibilidad). Superclases abstraídas tienen beneficios ademas del reuso. Ofrecen modularidad para separar componentes que puedan ser mantenidos por separado.

Ejemplo: la operación *dibujar* de una figura geométrica en un despliegue varía entre figuras diferentes, como círculos o líneas, pero tiene una inicialización común, como color o ancho de línea, que se puede heredar de la clase abstracta *figura*.

3.2.5.3 Delegación

Herencia es un mecanismo para implementar generalización donde el comportamiento de las superclases es compartido por todas sus subclases. Compartir se justifica sólo cuando una verdadera relación de generalización ocurre, y cuando la subclase es una forma de su superclase. Sobre-escribir operaciones de la superclase debe proveer los mismos servicios ofrecidos por la superclase, y a veces más. Por ejemplo, cuando una clase B *hereda la especificación* de una clase A, se puede asumir que cada instancia de B *es* una instancia de A. Algunos programadores usan herencia como una técnica de implementación sin existir una herencia semántica. Se desalienta el uso de *herencia de implementación.*

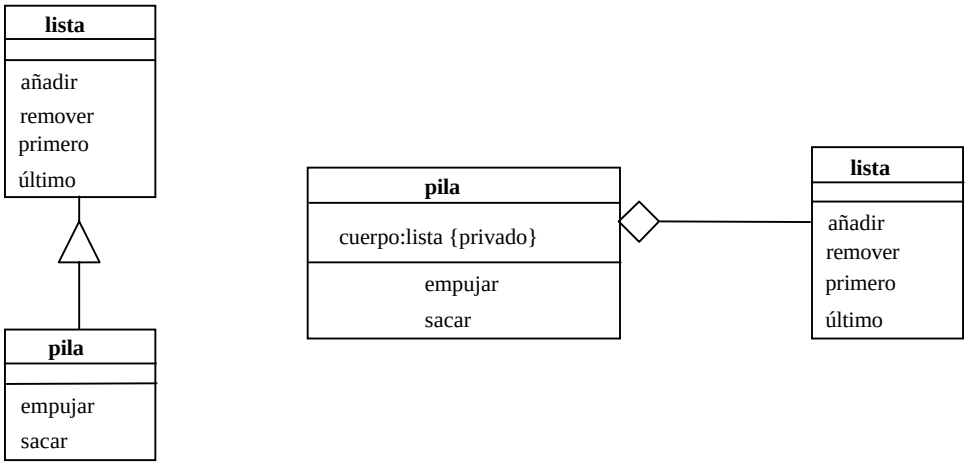


Figura 3.13. Diagrama de clases para una *pila* basada en una *lista*.

Ejemplo: si se quiere implementar una clase *pila* y ya se tiene una clase *lista*, se podría heredar *pila* de *lista*. Empujar un elemento en una pila se puede lograr añadiendo elementos al final de una lista, y sacar un elemento de la pila corresponde a quitar el elemento del final de la lista. Pero también se hereda las operaciones de *lista* no deseadas, como añadir o quitar elementos de posiciones arbitrarias de la lista. Ver la Figura 3.13. a la izquierda.
Se puede lograr lo mismo de una forma mejor. Una clase puede invocar selectivamente las funciones deseadas de otra clase usando *delegación* y no herencia. La delegación consiste en tomar la operación de un objeto y mandarla a otro objeto que está relacionado con el primer objeto. Sólo las operaciones importantes son delegadas al segundo objeto. Ver la Figura 3.13, a la derecha. En el segundo caso, cada instancia de *pila* contiene una instancia privada de *lista*. La operación *pila::empujar* delega la lista llamando sus operaciones *último* y *añadir* para agregar un elemento al final de la lista, y *sacar* es similar usando *último* y *remover*.

3.2.5.4 Herencia Múltiple por Medio de Herencia Sencilla

Ya que muchos lenguajes de programación orientados a objetos no apoyan la herencia múltiple, veremos aquí algunas técnicas para implementar herencia múltiple a través de herencia sencilla y delegación.

1) Implementación de herencia múltiple usando agregación: Una superclase con múltiples generalizaciones individuales se puede redefinir como un agregado en el cual cada componente del agregado reemplaza una de las generalizaciones. Se remplaza cada instancia de clase de la herencia múltiple por un grupo de instancias que componen el agregado. La herencia de las operaciones a través del agregado no es automática, debiendo ser delegadas a los componentes apropiados. Mediante este enfoque, no se crean clases unidas explícitas, si no que se crean de forma implícita como subclases de diferentes generalizaciones.

Ejemplo: Como se muestra en la Figura 3.14, *empleado pago* pasa a ser la superclase de *empleado exento*, *empleado asalariado*, y *empleado por hora*. *Empleado pensionado* pasa a ser la superclase de *empleado con pensión* y *empleado sin pensión*. *Empleado* es entonces modelado como un agregado de *empleado pago* y *empleado pensionado*. Una operación de *empleado*, como *computar pago*, debe ser propagada a ambos componentes de empleado.

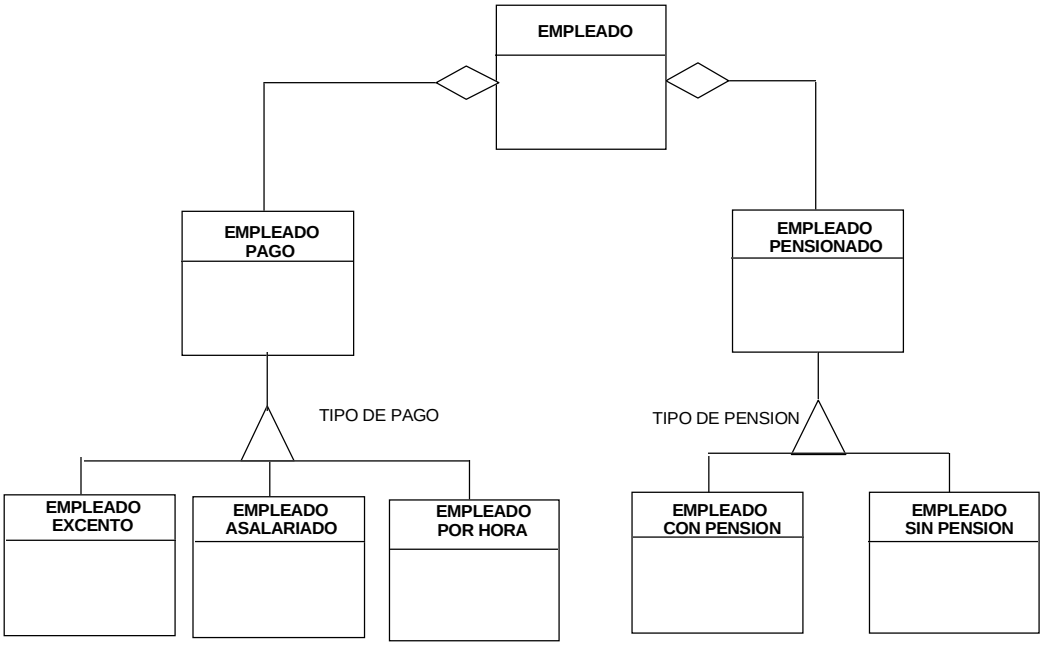


Figura 3.14. Diagrama de clases para implementar herencia múltiple por medio de herencia sencilla y agregación. Un empleado está ahora compuesto de un objeto de tipo *empleado pago* y otro de tipo *empleado pensionado*.

2) Implementación de herencia múltiple heredando de la clase más importante y delegando el resto: La clase unida es una subclase de la superclase más importante, combinándose con un agregado correspondiendo a las generalizaciones restantes.

Ejemplo: Como se muestra en la Figura 3.15, se mantiene la herencia de la superclase *empleado* de las subclases *empleado exento, empleado asalariado,* y *empleado por hora*. *Empleado pensionado* es la superclase de *empleado con pensión* y *empleado sin pensión*. Un *empleado* es entonces modelado como un agregado de *empleado pensionado.*

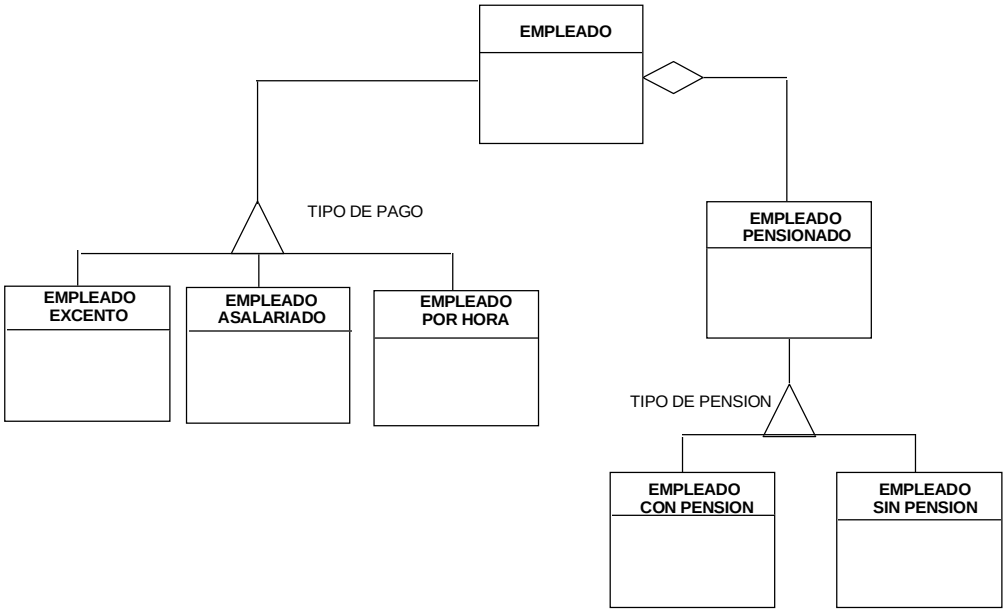


Figura 3.15. Diagrama de clases para implementar herencia múltiple por medio de herencia sencilla y agregación. Un empleado está ahora compuesto de un objeto de tipo *empleado pensionado*, manteniendo la herencia de *empleado pago*.

3) Implementación de herencia múltiple usando generalización anidada: Se crean varios niveles de generalización, terminando la jerarquía con subclases para todas las posibles combinaciones de clases unidas. En este caso no se utiliza agregación. Se preserva la herencia pero se duplica las declaraciones, rompiendo con el espíritu de orientación a objetos. Se debe factorizar primero según el criterio de herencia más importante, y luego según los demás.

Ejemplo: Como se muestra en la Figura 3.16, se mantiene crean diferentes niveles de herencia, creando al final todas las posibles combinaciones de subclases *empleado exento con pensión*, *empleado exento sin pensión*, *empleado asalariado con pensión*, *empleado asalariado sin pensión*, *empleado por hora con pensión,* y *empleado por hora sin pensión.*

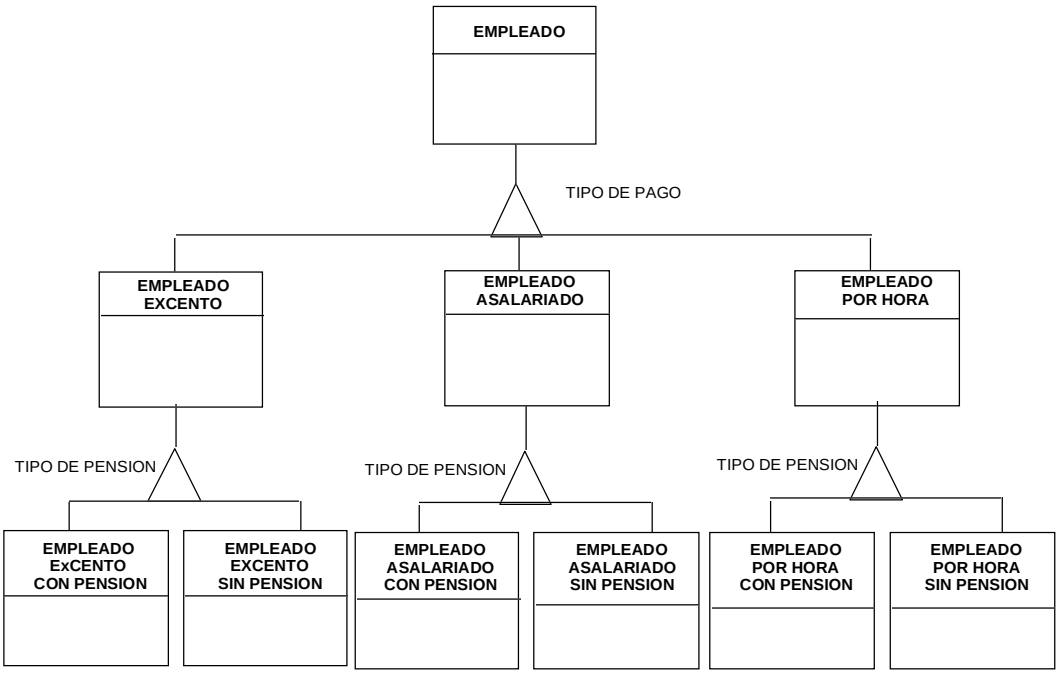


Figura 3.16. Diagrama de clases para implementar herencia múltiple por medio de solamente herencia sencilla. Se crean combinaciones para todos los posibles tipo de *empleado*.

Consideraciones:
• Si una subclase tiene varias superclases, todas de igual importancia, es mejor usar delegación y preservar la simetría (caso 1).
• Si una superclase domina, se implementa la herencia múltiple a través de herencia sencilla y agregación (caso 2).
• Si el número de combinaciones es pequeño, se puede usar generalización anidada (caso 3). Si el número de combinaciones, o el tamaño del código, es grande se debe evitar este tipo de implementación.
• Si una superclase tiene bastante más características que las otras superclases, o si una superclase es el cuello de botella en el rendimiento, se debe preservar la herencia en relación a esa clase (casos 2 y 3).

3.2.6 Diseñar Asociaciones

Las asociaciones proveen el acceso entre los objetos. Durante el diseño de objetos se debe formular una estrategia para implementar las asociaciones.
Los siguientes aspectos deben ser analizados durante el diseño de asociaciones:
1) Añadir travesías
2) Asociaciones de una dirección
3) Asociaciones de dos direcciones
4) Atributos de liga

3.2.6.1 Añadir Travesías

Se ha asumido hasta ahora que las asociaciones son por naturaleza bidireccionales. Esto es cierto en un sentido abstracto, pero si las asociaciones son atravesadas en una sola dirección, su implementación se podría simplificar. En el modelo conceptual se usa asociaciones bidireccionales, mientras que en la implementación se puede optimizar algunas asociaciones en un solo sentido.

3.2.6.2 Asociaciones de una Dirección

Si la asociación se atraviesa en una sola dirección, se puede implementar como un *apuntador*, o sea, un atributo que contiene una referencia a otro objeto.

Ejemplo: multiplicidad de "1" => apuntador simple, ver la Figura 3.17.

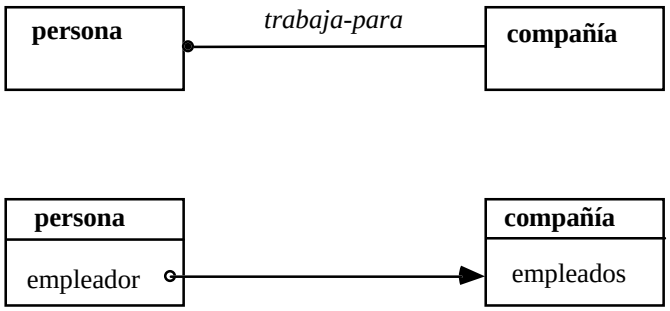


Figura 3.17. Implementación de asociación con apuntador sencillo.

Si la multiplicidad es "muchos" se necesita un conjunto de apuntadores; y si el lado "muchos" está ordenado, entonces una lista puede usarse en lugar de un conjunto.

3.2.6.3 Asociaciones de dos Direcciones

Muchas asociaciones se atraviesan de ambos lados, aunque no frecuentemente.

Enfoques para la implementación:
• se implementa con un atributo en una sola dirección y se hace una búsqueda cuando una travesía hacia atrás se requiere. Este enfoque es útil si hay una gran disparidad en la frecuencia de travesías en las dos direcciones. Se minimiza el costo de almacenamiento y actualización, aunque la travesía hacia atrás puede que sea muy costosa.
• se implementa como atributo en ambas direcciones, con dos apuntadores. Este enfoque permite acceso rápido, pero si cualquier atributo es actualizado, entonces el otro debe ser actualizado para mantener la consistencia de la conexión. Es útil si el numero de accesos es mayor que el de actualizaciones. Ver el diagrama de la Figura 3.18.

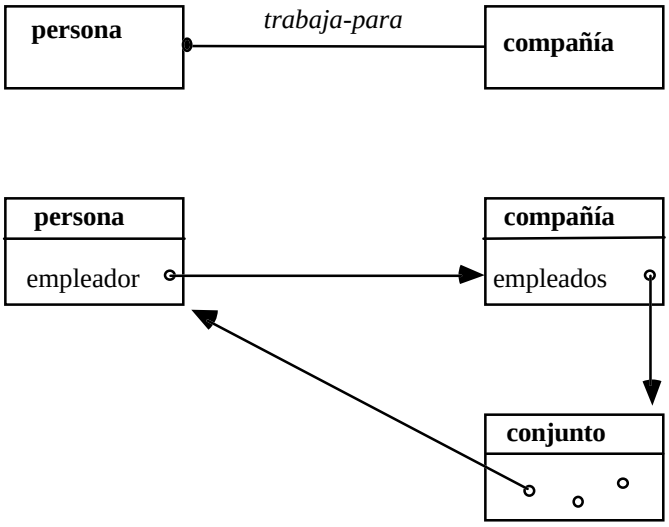


Figura 3.18. Implementación de asociación con apuntadores de dos direcciones.

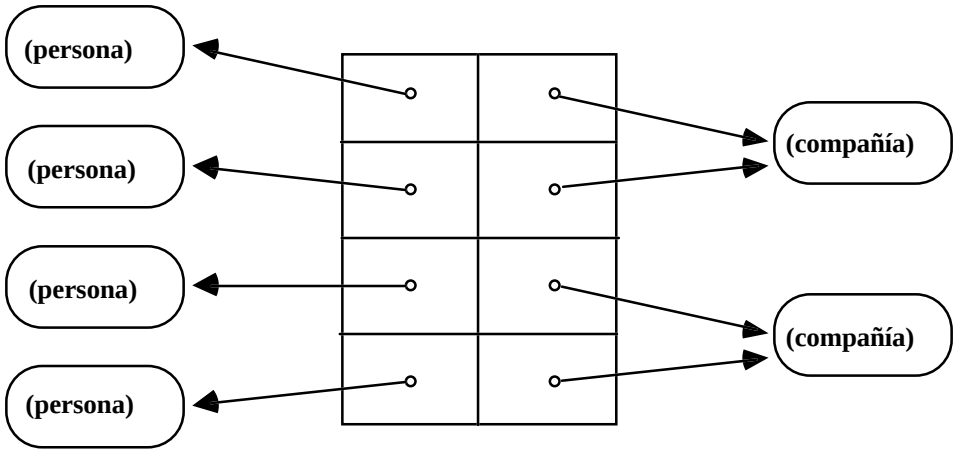


Figura 3.19. Implementación de asociación con objeto distinto.

• se implementa como un objeto de asociación distinto, como un conjunto de pares de objetos asociaciones guardadas en un solo objeto de tamaño variable. Para eficacia el objeto de asociación puede ser implementado usando dos objetos diccionario, uno hacia adelante y otro hacia atrás. El acceso es un poco más lento que con atributos de apuntador. Este enfoque es útil para extender clases predefinidas de una librería que no se pueden modificar, ya que el objeto de asociación puede ser añadido sin agregar atributos a las clases originales. Objetos distintos de asociación son útiles para asociaciones esparcidas en donde la mayoría de objetos de la clase no participan en las asociaciones. Ver la Figura 3.19.

3.2.6.4 Atributos de Liga

Atributos de liga se pueden implementan según su multiplicidad:
• multiplicidad "1-1": los atributos se pueden guardar como atributos de uno de los objetos.
• multiplicidad "1-muchos": los atributos se guardan como atributos del objeto del lado "muchos", ya que cada objeto aparece una sola vez en la asociación.

• multiplicidad "muchos-muchos": el atributo no puede asociarse con ninguno de los objetos. El mejor enfoque es implementar la asociación como una clase distinta, en la cual cada instancia representa una liga y sus atributos.

3.2.7 Representar Objetos

La implementación de los objetos es directa, pero el diseñador debe escoger cuando usar tipos *primitivos* o *literales*, como *enteros*, *cadenas*, o tipos *enumerados*, en representación de objetos y cuando combinar grupos de objetos relacionados.

Ejemplo: la implementación del número del seguro social dentro de un objeto *empleado* puede ser hecha de diferentes formas, como se muestra en la Figura 3.20. El atributo del número del seguro social puede ser implementado como un entero o cadena, o como una asociación a un objeto número del seguro social que puede contener un *entero* o *cadena*. Definir una nueva clase es más flexible pero introduce una indirección adicional.

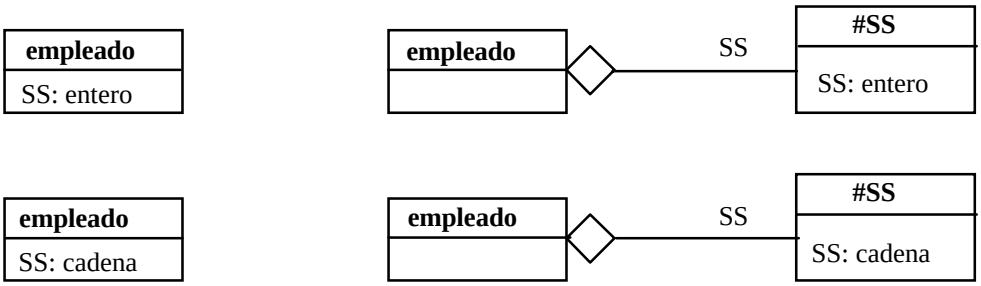


Figura 3.20. Representación de objetos de diferentes formas.

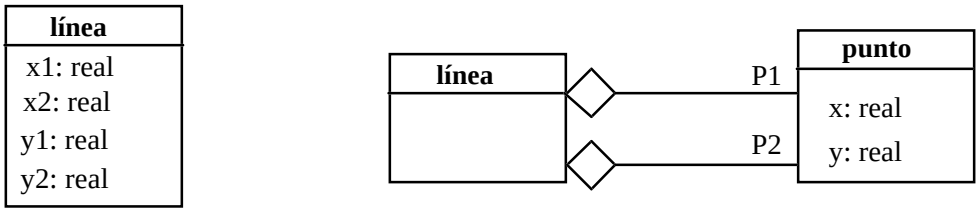


Figura 3.21. Representación de objetos de diferentes formas.

Ejemplo: la representación de líneas de 2 dimensiones puede hacerse de varias formas, como se muestra en la Figura 3.21. Ninguna representación es mejor que otra ya que las dos son matemáticamente correctas.

3.2.8 Empacar en Módulos

Los programas son unidades físicas discretas que pueden ser editadas, compiladas, importadas, o manipuladas de otra forma. Los lenguajes orientados a objetos tienen varios niveles de empaque.

Aspectos a diseñar:
1) Ocultar información
2) Coherencia de entidades

3) Construir módulos

3.2.8.1 Ocultar información

Durante el análisis no nos concierne ocultar información. Durante el diseño se debe definir la interfaz pública de cada clase, decidiendo qué atributos y operaciones son accesibles fuera de la clase. Ocultar la información interna permite que la implementación de una clase pueda ser cambiada sin requerir a los clientes de la clase modificar su código. El diseñador debe balancear las demandas de ocultamiento y las de optimización.

En caso extremo una operación de una clase puede atravesar todas las asociaciones de un modelo de objetos y accesar otros objetos. Esta visibilidad no restringida es apropiada durante el análisis, pero los métodos que saben demasiado son frágiles ya que cualquier cambio los invalida. Durante el diseño se trata de limitar el alcance de cada método, definiendo que otras clases el método ve.

Principios de diseño para limitar el alcance de operaciones:
• Asignar a cada clase la responsabilidad de ejecutar operaciones y proveer información que le concierne
• Llamar una operación para accesar atributos que pertenecen a un objeto de otra clase.
• Evitar atravesar asociaciones que no están conectadas a la clase
• Definir interfaces en el nivel más alto de abstracción posible
• Ocultar objetos externos al borde del sistema definiendo clases de interfaz abstractas, que median entre el sistema y los objetos externos
• Evitar aplicar un método al resultado de otro método. En su lugar considerar escribir un método que combine las dos operaciones

3.2.8.2 Coherencia de Entidades

Un importante principio de diseño es la *coherencia* de entidades. Las clases, operaciones, y módulos, deben organizarse con un plan consistente y todas sus partes deben encajar para una meta común. Una entidad no debe ser una colección de partes no relacionadas. Una operación debe hacer una cosa bien, y no debe mezclar a la vez *política* e *implementación*.
Política: tomar decisiones que dependen del contexto
Implementación: ejecución del algoritmo totalmente especificado

La política involucra tomar decisiones, obtener información global, interaccionar con el mundo externo, e interpretar los casos especiales. Una operación política no contiene algoritmos complicados, pero llama en su lugar varias operaciones de implementación. Una operación de implementación hace exactamente una operación, sin tomar decisiones, suposiciones, omisiones, o desviaciones. Toda su información es provista por sus argumentos, que pueden ser muchos.

Separando política de implementación incrementa mucho la posibilidad de reuso. Las operaciones de implementación no contiene dependencias de contexto y podrían ser re-utilizadas. Las operaciones políticas por lo general deben re-escribirse en nuevas aplicaciones, pero son más sencillos, se ocupa de decisiones de alto nivel, y llamando operaciones de bajo nivel.

Ejemplo: Una operación para acreditar interés en una cuenta de cheques. El interés es calculado diariamente basado en el balance diario, pero todo el interés de un mes se pierde si la cuenta se cierra. El acreditar interés se debe separar en dos:
• la implementación que computa el interés entre un par de días, sin importar provisiones aparte, y
• el método político que decide cuando se llama la operación de implementación.
Esta separación permite que la política o implementación se modifiquen independientemente e incrementa la posibilidad de re-utilizar la implementación que es más complicada.

3.2.8.3 Construir Módulos

Durante el análisis y diseño del sistema se divide el modelo de objeto en módulos. La organización inicial puede que no ea buena u óptima para el empaque final de la implementación del sistema. La conectividad del modelo de objeto se puede usar como guía para dividir en módulos. Las nuevas clases añadidas durante el diseño se pueden agregar a un modulo o capa existente, o se pueden organizar en un módulo o capa nueva. Los módulos deben ser definidos para que sus interfaces sean mínimas y bien especificadas. La interfaz entre dos módulos consiste de las asociaciones que relacionan clases entre los módulos, y las operaciones que accesan clases a través de los bordes de los módulos.

3.3 Documentar el Diseño

Es imposible recordar los detalles del diseño, siendo la documentación la mejor forma de transmitir el diseño a otros personas, aunque sea como referencia para mantenimiento. El documento de diseño debe ser una extensión del documento de requisitos de análisis, incluyendo una descripción detallada del modelo de objetos en forma gráfica y textual. Notación adicional es apropiada para mostrar decisiones de implementación, como flechas mostrando la dirección de travesía de las asociaciones y los apuntadores de los objetos. También el modelo dinámico y funcional deben ser documentados.