1 Introducción

Nos encontramos a las puertas de un nuevo milenio y observamos cómo la tecnología de información ha revolucionado nuestra forma de vida. Esto se refleja en las comunicaciones, la industria, la administración, la educación, los empleos, los servicios, la medicina, etc.

Nos enfrentamos a un mundo vertiginosamente cambiante donde la competencia es tan fuerte que empresas o compañías que hoy se encuentran en el mercado, mañana habrán desaparecido, y otras que hoy no existen, mañana resultan ser empresas importantes. Tal situación puede semejarse a una inmensa ola (tecnológica en nuestro caso) que se mueve estrepitosamente arrasando con todo aquello que no está preparado para soportarla, quedando en pie lo que se encuentra en una posición estratégicamente situada, moviéndose y adaptándose flexiblemente a las transformaciones producto de ese empuje arrollador.

La disciplina del software, en medio de ese empuje, necesita someterse a cambios que le permitan moverse acorde con las necesidades que las condiciones actuales demandan.

En el último cuarto de siglo, el desarrollo de software se ha visto impulsado principalmente por las necesidades de los usuarios y por el desarrollo del hardware que le sirve de infraestructura para poderse ejecutar. Así, observamos en los sesentas los sistemas centralizados, girando alrededor de los "mainframes", a lo que posteriormente se agregó el procesamiento de datos en línea y el teleproceso, siguió la aparición de las minis en los setentas y con ellas ambientes de programación más amistosos (así como una proliferación de lenguajes de programación que nos reprodujo una Babel informática a pequeña escala). Su precio más reducido permitió a empresas medianas comenzar su aventura informática. Las minis facilitaban el trabajo, pero no resolvían todos los problemas.

A inicios de los ochentas la computadora personal hizo su aparición como herramienta de gestión. Algunos entusiastas comenzaron a hacer aplicaciones para empresas pequeñas. Esta situación se hizo seria con la aparición de los discos duros a precios más accesibles y, años más tarde, las tecnologías de red de área local.

El desarrollo vertiginoso del hardware dejó retrasado el desarrollo del software [Winblad1990]: las técnicas, herramientas y abstracciones del software convencional resultan inadecuadas mientras que los sistemas de software son cada vez más complejos. Las empresas están sometidas a un entorno de competencia muy cambiante, que las obliga a una fuerte reactividad, y al mismo tiempo la complejidad de los programas y de su desarrollo ha aumentado considerablemente. En muchas organizaciones, el tiempo entre la demanda y la realización de una nueva aplicación se ha hecho inaceptable debido a los tiempos de reacción que los mercados imponen a las empresas. El resultado de estas tendencias divergentes, de problemas que cambian más rápidamente y de programas desarrollados más lentamente, da finalmente un distanciamiento desastroso entre los sistemas de información y las necesidades de la empresa.

Es imperativo encontrar métodos que orienten mejor la labor de los desarrolladores de software. Estos métodos deben ayudar a la construcción de productos que respondan mejor a las necesidades de sus clientes o usuarios, sean económicos en el empleo de recursos, con buenas

características técnicas y cuyo proceso constructivo sea administrable y predecible en cuanto a costos y plazos. Estos métodos deben ayudar a cambiar la perspectiva adoptada para la organización y desarrollo de sistemas. Ya no se trata de desarrollar puntualmente aplicaciones monumentales, sino de implementar una *arquitectura*, es decir una infraestructura funcional utilizable y reutilizable para el desarrollo de *familias de aplicaciones* que respondan a las numerosas necesidades con una alta capacidad de adaptación a los cambios.

Un enfoque orientado a la arquitectura de los sistemas de información se basa en un modelaje de las actividades y estructuras de la empresa, y en una disciplina técnica de integración de sistemas. En este doble sentido, sobrepasa ampliamente el ámbito técnico de la informática. Su éxito se basa en la comprensión fina de los *procesos* de decisión y operación de la empresa y en la evaluación del impacto de las condiciones cambiantes en las que deben ejecutarse estos procesos. Desde un punto de vista técnico, la integración de sistemas de naturaleza heterogénea se convierte en el objetivo fundamental de la construcción de los sistemas de información.

En busca de alcanzar estos cambios de perspectivas se ha impulsado la utilización del paradigma de orientación a objetos que llega más o menos a la madurez tras un largo período de gestación en el medio académico de la investigación. Los conceptos que se han convertido en el fundamento de la orientación a objetos tienen su origen a mediados de los años 60 y su evolución ha acompañado constantemente a la del ejercicio de la programación, que ha pasado de ser una actividad de búsqueda, de naturaleza matemática, a una práctica comercial y una disciplina formalizada y normalizada. Hoy los lenguajes de programación orientados a objetos se están convirtiendo en la lengua común de los programadores, las herramientas de desarrollo son legión y sus costos decrecientes refuerzan más esta tendencia [Chauvet 1997].

La orientación a objetos está esencialmente constituida por técnicas de construcción y manipulación de abstracciones. Estas abstracciones pueden ser las del modelo de la empresa, o bien las que constituyen la implementación del sistema de software. En el primer caso, la orientación a objetos es una metodología de desarrollo de sistemas que nos permite plasmar mediante modelos la estructura y comportamiento de los diferentes dominios en que trabaja la empresa. En el segundo caso, la orientación a objetos es una herramienta concreta de construcción y mantenimiento de estos sistemas.

Los métodos de la orientación a objetos demuestran ser formidables herramientas de análisis de sistemas complejos y de integración de sistemas, apropiados para reforzar las capacidades de abstracción y de adaptación de los sistemas actuales. De hecho, es cada vez más frecuente observar cómo sistemas heredados monolíticos son accedidos vía interfaces o capas de manipulación con tecnología de objetos [Jacobson 1994].

Los métodos orientados a objetos para realizar análisis y diseño que han surgido en esta década están incidiendo profundamente en el desarrollo de sistemas y han mostrado ser capaces de ayudar a reducir la brecha entre la necesidad y la capacidad de desarrollar software cada vez más complejo. Ante el empuje arrollador que la competencia impone, se hace necesario utilizar tecnologías que permitan a las empresas adoptar una posición que facilite su adaptación al cambio.

Producto de esa necesidad, este documento muestra los conceptos que soportan el enfoque orientado a objetos así como las características que le dan ventaja sobre las técnicas tradicionales para el desarrollo de software. Se presenta al final recomendaciones para la adopción de alguno de los métodos orientados a objetos.

2 Conceptos

Al diseñar un sistema de software se hace necesario descomponerlo en partes más pequeñas, cada una de las cuales se puede llegar a refinar de forma independiente. Como expresa Booch [Booch 1996] citando a Parnas: "La descomposición inteligente ataca directamente la complejidad inherente al software forzando una división del espacio de estados del sistema."

2.1 Descomposición Funcional vrs. Descomposición Orientada a Objetos

Tradicionalmente la descomposición realizada se orientó a los *procesos*, donde los elementos predominantes consisten en procedimientos y algoritmos. En las etapas iniciales, el sistema es visto en un alto nivel de abstracción en términos de *qué* es lo que hace, se continúa el proceso con la etapa de diseño centrándose en el *cómo* a través de los procesos se consiguen los objetivos planteados en la etapa anterior. La descomposición funcional utilizada para llevar a cabo el desarrollo se basa en la interpretación del espacio del problema y su transformación al espacio de la solución como un conjunto de funciones o procedimientos interdependientes. El sistema final es visto como ese conjunto de procedimientos operando sobre un estado compartido [Coleman 1994] (fig. 2.1).

Aunque posteriormente se desarrolló un enfoque orientado a los datos, tipificado por métodos como el de Jackson y el de Warnier [Booch 1996] donde se da una mayor atención a la especificación de los datos que en el enfoque anterior, la arquitectura del sistema continúa basada en la descomposición funcional y las estructuras de datos son utilizadas para asistir esa descomposición.

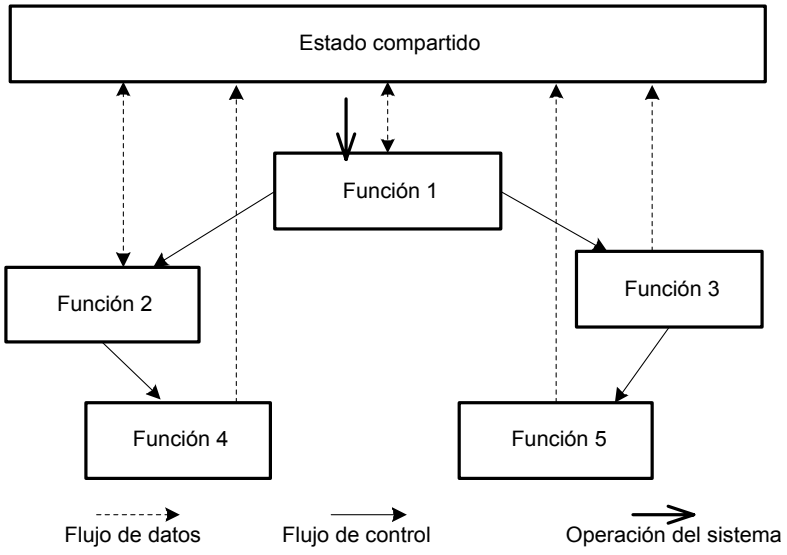


fig. 2.1 Modelo computacional orientado a las funciones

Este enfoque de descomposición, parece derrumbarse ante la creciente complejidad de las aplicaciones actuales, pues presenta varias deficiencias.

- Cualquier función puede operar sobre cualquier parte del estado.
- Las funciones deben saber cómo se almacenan los datos o las estructuras de datos, por lo que se necesitan cláusulas (if - then, case) para verificar los tipos de datos [Jacobson 1994].
- No toma en cuenta cambios evolutivos [Meyer 1988]
- El sistema es caracterizado por una única función.
- El énfasis en la mente de los desarrolladores son las funciones, por lo que a los aspectos relacionados con las estructuras de datos a menudo se les resta importancia.
- Falta de apoyo a la reutilización.

Esto da como resultado que las estructuras internas de las aplicaciones sean inestables, susceptibles a los cambios, difíciles de mantener y cada vez más difíciles de entender por la gran cantidad de interrelaciones que se establecen.

Ante estas dificultades se plantea una descomposición alternativa, *orientada a los objetos*, la cual modifica radicalmente la estructura interna del sistema. En vez de tener una función principal que llama a una gama de funciones, el sistema es visto como un conjunto de bloques interconectados, que se encargan de realizar las funciones del sistema, exhibiendo cada uno de ellos un comportamiento bien definido (fig. 2.2)

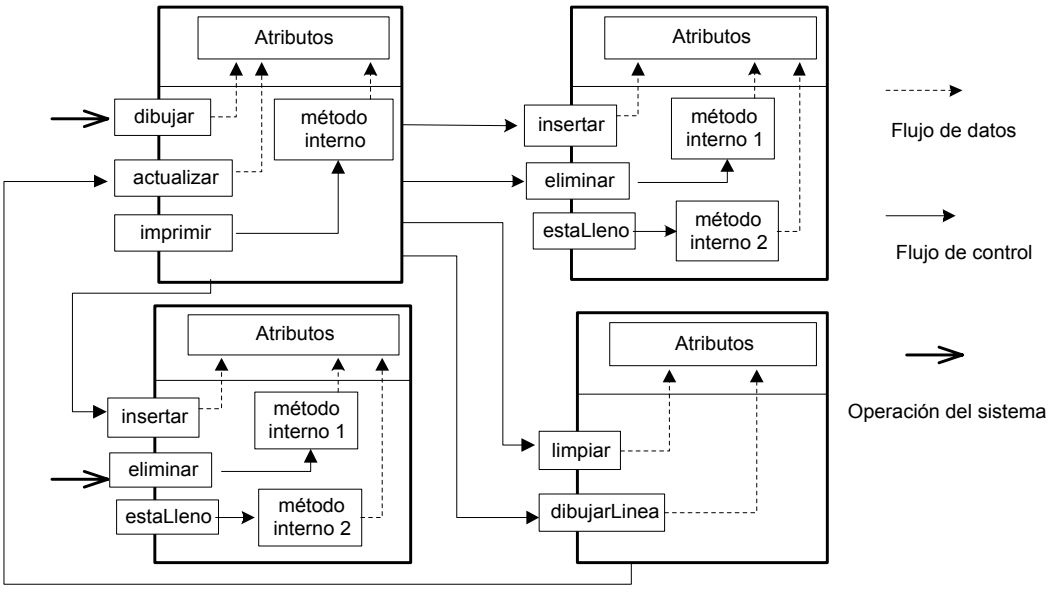


fig. 2.2 Modelo computacional orientado a objetos

A diferencia del modelo funcional donde los componentes esenciales (funciones) operan sobre un único estado, el enfoque orientado a objetos *encapsula* dentro de los objetos aquellas funciones (llamadas métodos) junto con piezas del estado sobre las que operan. Bajo esta arquitectura la funcionalidad del sistema queda asignada, encapsulada y distribuida entre los diversos objetos.

2.2 Funcionalidad del sistema

El lector debe tener claro que la representación que en esta sección se hace es una representación atípica ideada por los autores de este documento. Al representar los objetos y sus interacciones en un modelo tridimensional se pretende facilitar el entendimiento de los principios estructurales en que se basan los sistemas orientados a objetos.

Al tener cada objeto asignado un aspecto del comportamiento del sistema, la funcionalidad es proporcionada por la interacción de los diversos objetos, que se comunican mediante el paso de mensajes (fig. 2.3). Las estructuras de colaboración establecen las vías de comunicación, a través de las cuales circulan los mensajes que llevan el flujo de control y de datos que corresponden a las solicitudes de servicios, información y respuestas.

A fin de comprender las diversas formas en que se relacionan los objetos en un sistema desarrollado bajo un enfoque orientado a objetos, se propone una representación tridimensional, donde el elemento básico de construcción es el bloque que aparece en la fig. 2.4 denominado simplemente "el objeto". En la figura se muestran y describen las conexiones que le permiten al objeto interactuar de múltiples formas con otros objetos.

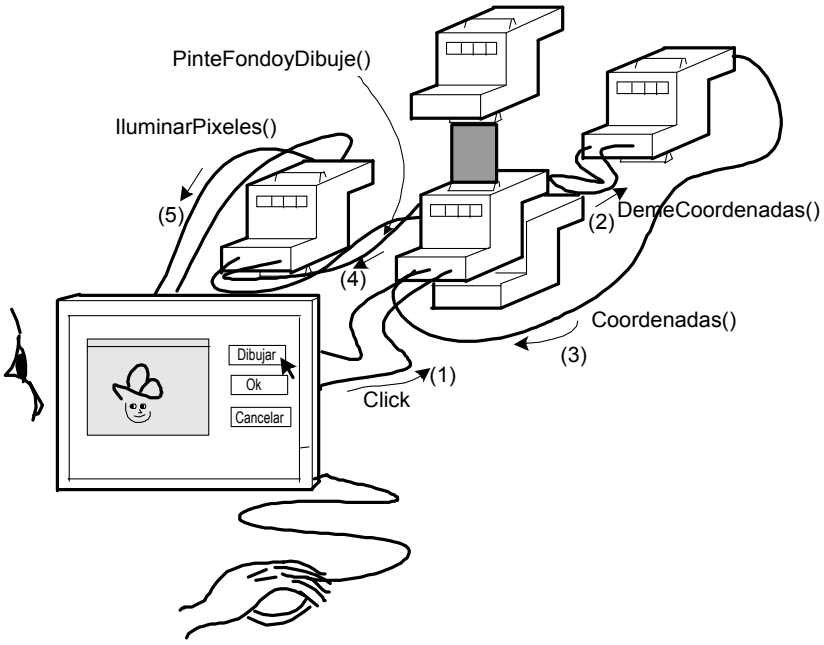


fig. 2.3 Interacción de objetos en un sistema de software

En la fig. 2.5 se muestra el interior del objeto, con una área para el procesamiento de los servicios solicitados por otros objetos, otra para los servicios internos que apoyan a los anteriores, una tercera para el almacenamiento de las características o atributos del objeto y una última donde se encuentran objetos en los que se delegan responsabilidades a fin de permitir hacer manejable la funcionalidad asignada al objeto. Dependiendo de su complejidad, esta área de objetos contenidos puede no existir. También se observan las rutas que asocian las áreas descritas con las diferentes conexiones hacia los otros objetos.

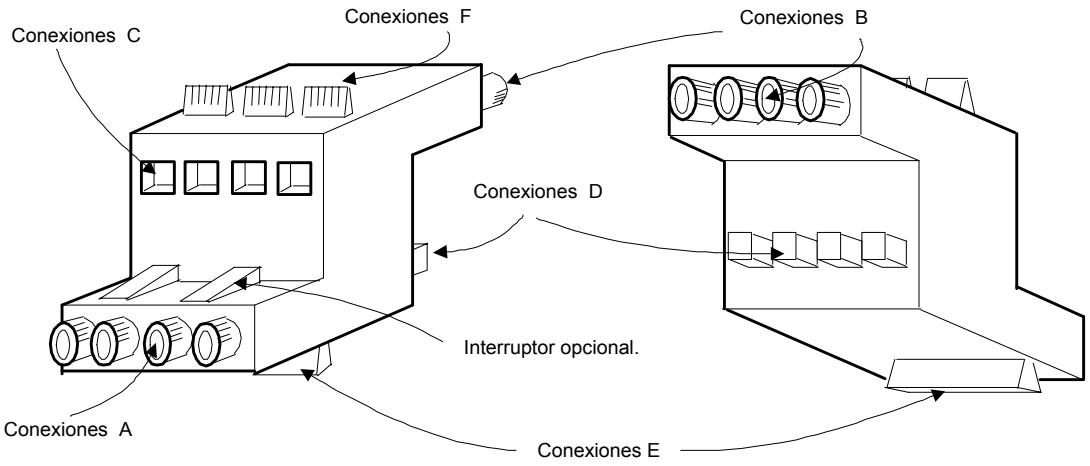


fig. 2.4 Vista exterior de un objeto

Tipos de conexiones y lo que permiten llevar a cabo
A: Recibir mensajes de otros objetos que desean algún servicio. B: Enviar mensajes a otros objetos C: Agregar el objeto a otro objeto. A través de ellas puede recibir y enviar mensajes. D: Agregar otros objetos E: Heredar información y funcionalidad a otro objeto. (Al objeto hijo) F: Obtener información y funcionalidad de otro objeto. (Del objeto padre) Interruptor: Dispositivo opcional que condiciona la aceptación del mensaje que llega.

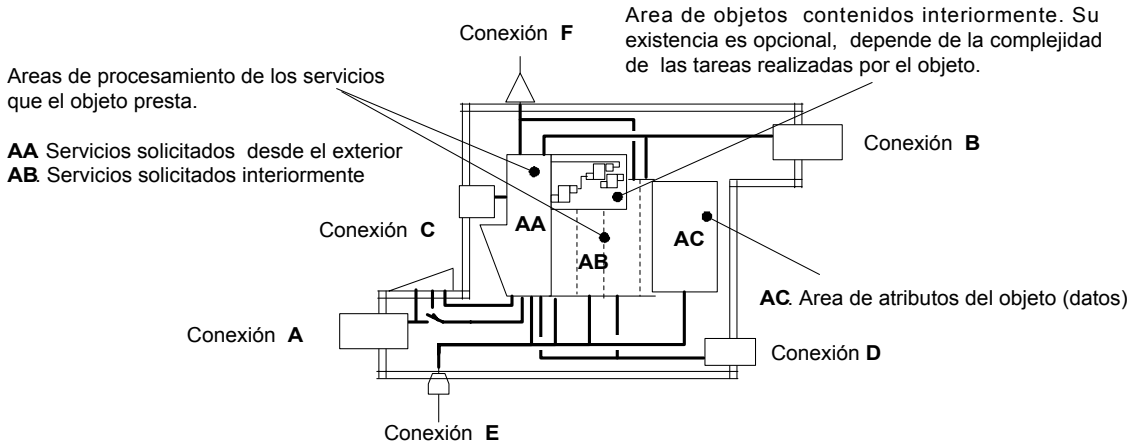


fig. 2.5 Vista interior de un objeto

Como se mencionó anteriormente el comportamiento del sistema se obtiene de las funciones que cada objeto lleva a cabo así como de la interacción entre los objetos, por lo que analizaremos brevemente los principales mecanismos mediante los cuales se asocian estos objetos.

Asociación de comunicación

Un objeto es diseñado para asumir varias responsabilidades, que permitan que otros objetos lo utilicen, solicitando sus servicios por medio del envío de mensajes. En la fig.2.6(a) se muestra la conexión A por donde se solicita el servicio, un interruptor (precondición, cuando exista) verifica que el mensaje pueda proseguir. En caso que sea denegado el paso, el interruptor manda un mensaje para que el objeto envíe la notificación correspondiente. Cuando la solicitud hecha lo requiera, por la conexión B se envía una respuesta, una solicitud de servicio a otro objeto o una notificación de fallo.

En esta forma de interacción la existencia de un objeto no depende del otro. El vínculo que une a los objetos eventualmente se romperá y cada uno puede seguir existiendo independientemente ya que la *asociación de comunicación* que se establece es de carácter transitorio, al denotar una relación de igual a igual o cliente/servidor.

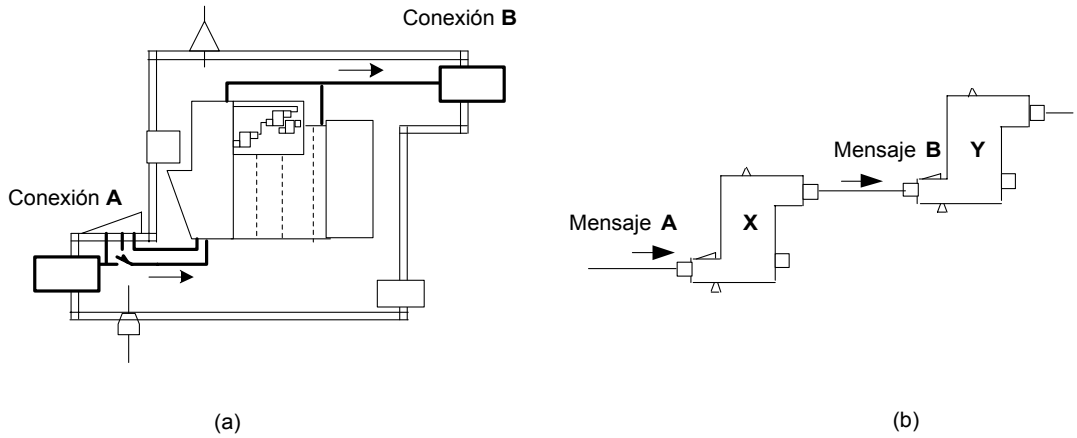


fig. 2.6 Un objeto usa los servicios de otro objeto

Agregación

Hay relaciones entre objetos que establecen una jerarquía conocida como **"todo/parte",** donde un objeto es agregado a otro objeto. En la fig. 2.7 (a) se muestran las conexiones por donde se realiza el vínculo en este tipo de interacción y en la parte (b) un ejemplo donde el objeto Y (la parte) se le agrega al objeto X (el todo). Este tipo de relación establece tanto una vía de acceso como una relación más estrecha que la anterior, en la que los tiempos de vida de ambos objetos pueden o no estar en íntima conexión. Existe la posibilidad que al destruirse el todo, se destruya también la parte o que siga existiendo.

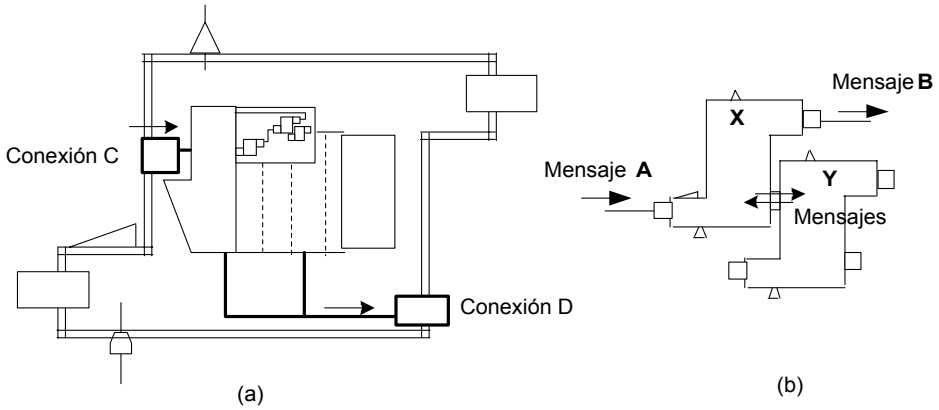


fig. 2.7 Un objeto se agrega a otro objeto

Composición

Al considerar de nuevo la jerarquía **todo/parte**, se establece ahora una relación conocida como composición, donde un objeto está compuesto de uno o varios objetos. Este tipo de relación se lleva a cabo a fin de dividir la funcionalidad del objeto cuando esta resulta demasiado compleja, cuando el objeto presenta características muy variadas, que dificultan su diseño o bien porque se descubre (en el dominio del problema) que conceptualmente existen objetos que se relacionan vía composición.

En la fig. 2.8 se presenta un ejemplo de esta relación, en ella se observa contención física que ilustra la idea de que los objetos contenidos no pueden existir independientemente del objeto que los contiene. El tiempo de vida de ellos se encuentra en íntima conexión; al destruirse el objeto contenedor se destruyen los objetos contenidos. Los objetos contenidos pueden comenzar a existir siempre y cuando el contenedor exista. Los objetos contenidos no pueden existir más allá (ni antes ni después) de la vida del objeto contenedor.

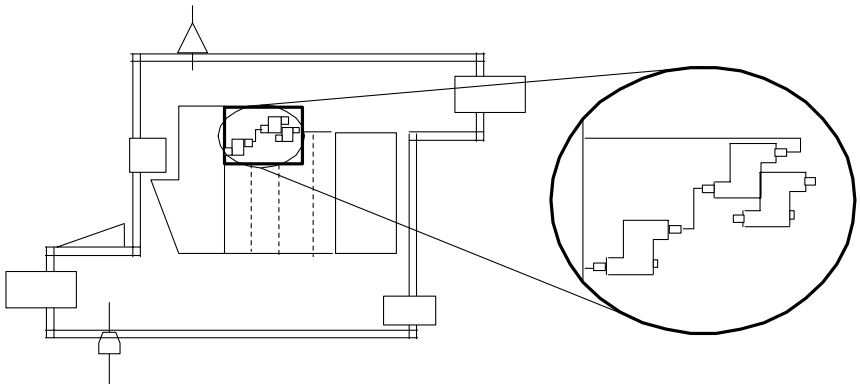


fig. 2.8 Un objeto se compone de otros objetos

Generalización/especialización (herencia)

La relación **"es un"** establece una jerarquía donde un objeto (denominado padre) hereda sus propiedades y comportamiento a otro objeto (denominado hijo). Aquí cambia la perspectiva de colaboración entre los objetos, orientándola hacia permitir extender la funcionalidad heredada al hijo a costa de romper el encapsulamiento del objeto padre.

Podemos observar en la fig. 2.9 la representación realizada para visualizar esta relación. En la parte (a) se han dibujado un objeto padre y un objeto hijo, el objeto hijo tiene en su interior únicamente el área para almacenar su propio estado, cuya estructura ha sido heredada, dependiendo su funcionalidad estrictamente del padre. La conexión F del hijo le permite penetrar las barreras del encapsulamiento por la conexión E y tener acceso al estado del padre, a sus áreas de procesamiento público y privado.

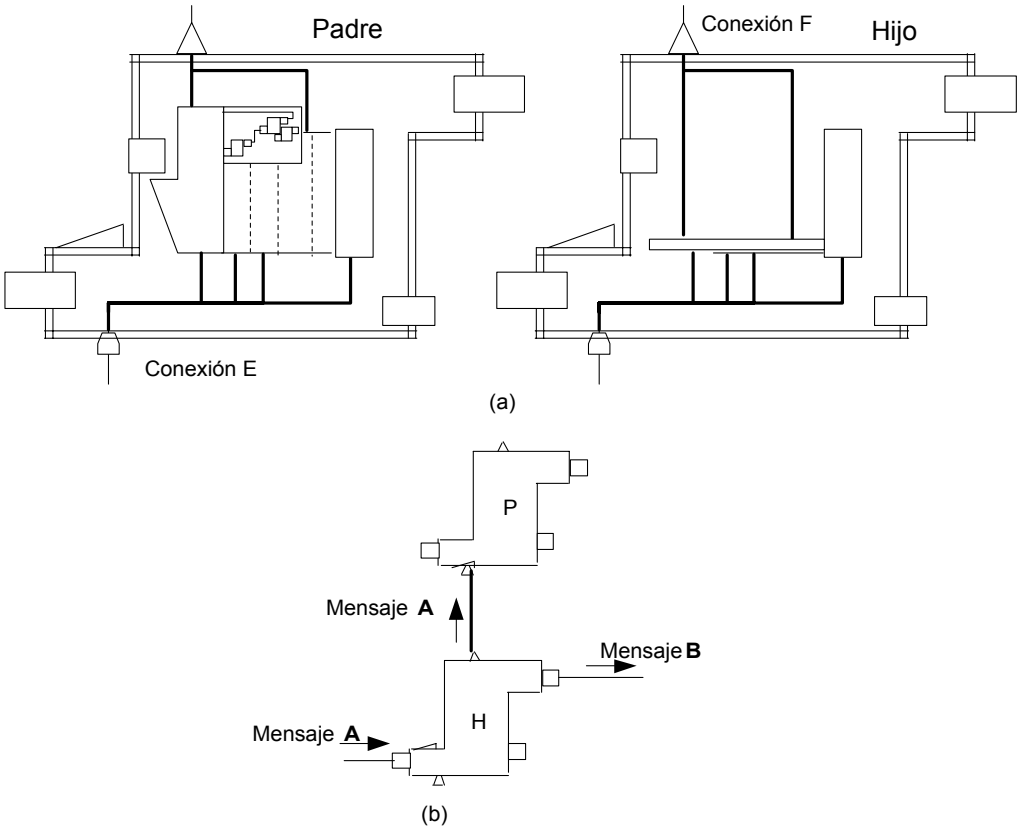


fig.2.9 Un objeto le indica a otro cómo realizar sus funciones

Como el objetivo de crear esta relación es extender la funcionalidad del objeto padre en el hijo, éste último puede perfectamente anular, sobreescribir o extender dicha funcionalidad, conteniendo sus propias áreas de procesamiento.

Si bien es cierto que esta relación busca fomentar la reutilización, es recomendado emplearla con sumo cuidado ya que establece una relación rígida. Al permitir penetrar el encapsulamiento, un cambio pobremente planeado en los padres pueden tener consecuencias indeseables en los hijos.

2.3 Beneficios del uso de los objetos

Los objetos poseen una serie de características que facilitan el desarrollo de sistemas. Entre ellas podemos mencionar:

- Separan la interfaz de la implementación
- Se corresponden mejor con sus contrapartes en el mundo real
- Pueden ser compuestos por otros objetos
- Evolucionan a lo largo del proceso de desarrollo

Como producto de estas características, se obtienen diversos beneficios del empleo de las técnicas de orientación a objetos. A continuación listamos los principales:

- **Alto grado de reutilización**: Las entidades en un modelo orientado a objetos pueden ser estructuradas con interfaces que permiten su reutilización por varias aplicaciones dentro del mismo dominio, ej. bibliotecas de clases. Al ser independientes y extensibles, estas entidades conducen de una forma natural a sistemas de software reutilizables, mantenibles y portables.

- **Mejor comunicación con los clientes:** Los objetos pueden describir entidades del mundo real que los clientes pueden comprender fácilmente.

- **Tiempo de desarrollo más corto:** El enfoque orientado a objetos se presta para desarrollar prototipos con conceptos que un cliente puede inspeccionar y aprobar antes que el equipo de desarrollo detecte errores que obliguen a revisiones, con las subsecuentes pérdidas de tiempo. Así mismo, los procesos de desarrollo proceden naturalmente de manera iterativa e incremental (ver cap. 4).

- **Facilidad de mantenimiento:** Con una correspondencia muy cercana entre los conceptos de los usuarios y las entidades en la implementación, los cambios en los requerimientos y las extensiones pueden ser aislados en unos cuantos objetos, además dichos cambios estarán más localizados (alta cohesión interna y bajo acoplamiento externo), dando como resultado una reducción en el código y las pruebas.

Para mayores detalles referente a los conceptos vistos en este capítulo se puede consultar el anexo donde se tratan los objetos, las clases y sus relaciones.

3 Modelaje

Cuando se pretende realizar un proyecto serio, el cual incorpora una complejidad significativa o una funcionalidad crítica, en cualquier rama de ingeniería se requiere la construcción de modelos a fin de reflejar en ellos la comprensión del problema y/o su respectiva solución.

Dentro de la ingeniería del software podemos observar dos aspectos importantes:

- Desde el enunciado de los requerimientos hasta la implementación de los sistemas se producen descripciones de modelos. Jacobson observa lo siguiente: "el desarrollo de sistemas se puede ver como una transformación gradual de una secuencia de modelos".

- Las aplicaciones resultantes implican una complejidad inherente que en muchas ocasiones excede la capacidad intelectual humana individual. Booch [Booch 1996] refiriéndose a las aplicaciones de dimensión industrial, apunta: "son aplicaciones que exhiben un conjunto muy rico de comportamientos, tienden a tener un ciclo de vida largo, llegando muchos usuarios a depender de su funcionamiento correcto, donde la característica distintiva de estas aplicaciones es que resulta sumamente difícil, si no imposible, para el desarrollador individual comprender las sutilezas de su diseño".

Según sean los métodos empleados para producir los modelos, estos pueden resultar en descripciones vagas e imprecisas, con una arquitectura sensible a los cambios o, por el contrario, precisas, carentes de ambigüedades, con la capacidad de atacar y controlar la complejidad, con lo que se mejora la calidad del modelaje. Estas son, entre otras, características que se pretenden conseguir utilizando los métodos orientados a objetos para modelar los sistemas de software.

Las técnicas de modelaje tradicional ponen énfasis en la descomposición funcional, enfocándose hacia la solución. Las técnicas de análisis estructurado originales (DeMarco/Yourdon) proponían la construcción en secuencia de cuatro modelos de análisis: modelo físico del sistema actual, modelo lógico del sistema actual, modelo lógico del nuevo sistema y modelo físico del nuevo sistema. A partir de éste último se hacía una transformación para obtener un diseño estructurado [Yourdon 1989]. Todos esos modelos son difíciles de mantener y su estructura es muy sensible a los cambios, por lo que la evolución de los sistemas resulta tediosa, laboriosa y onerosa. Las versiones más recientes [Yourdon 1989] del análisis estructurado consideran más bien la formulación de un modelo esencial [McMenamin 1984] que enfatiza el *qué,* sin compromisos relativos a la implantación, pero siempre bajo un enfoque de descomposición funcional.

En contraste, las técnicas de orientación a objetos se centran en los objetos, con énfasis en el aspecto estructural, lo que define relaciones e interacciones que establecen vías de enlace por donde se lleva a cabo la comunicación tanto de control como de información. La funcionalidad del sistema queda asignada, distribuida y encapsulada dentro de los objetos. Este enfoque hace énfasis en obtener una sólida comprensión del problema antes de intentar su solución.

3.1 Secuencia de modelos

Los modelos que se producen a lo largo del proceso de desarrollo se construyen con objetivos diferentes y por ello presentan diferentes grados de detalle. Al comienzo los modelos son bastante abstractos, enfocándose en cualidades externas del sistema; los modelos subsiguientes vienen a ser más detallados y prescriptivos pues describen cómo el sistema deberá ser construido y cómo se desea que funcione computacionalmente.

El modelo inicial es una descripción abstracta que se construye para comprender el problema antes de implementar la solución. El objetivo fundamental es modelar la semántica del problema en términos de los objetos relacionados. Cook [Cook 1994] denomina a éste el *modelo esencial*, cuyos elementos básicos son objetos (tipos de objetos[5]) y eventos. Los objetos modelarán cosas, las cuales pueden ser concretas o abstractas, transitorias o permanentes. Los eventos modelarán ocurrencias o acontecimientos (sucesos).

Posteriormente ese modelo se transforma y se hace corresponder a otro modelo, un modelo orientado a la solución del problema. Este modelo lo denominamos *modelo de especificación* [Cook 1994]. Su propósito es declarar *qué* hará el software, asignando entre los tipos de objetos las responsabilidades que determinan el comportamiento del sistema. Tal como el modelo esencial, este modelo trata con objetos y eventos.

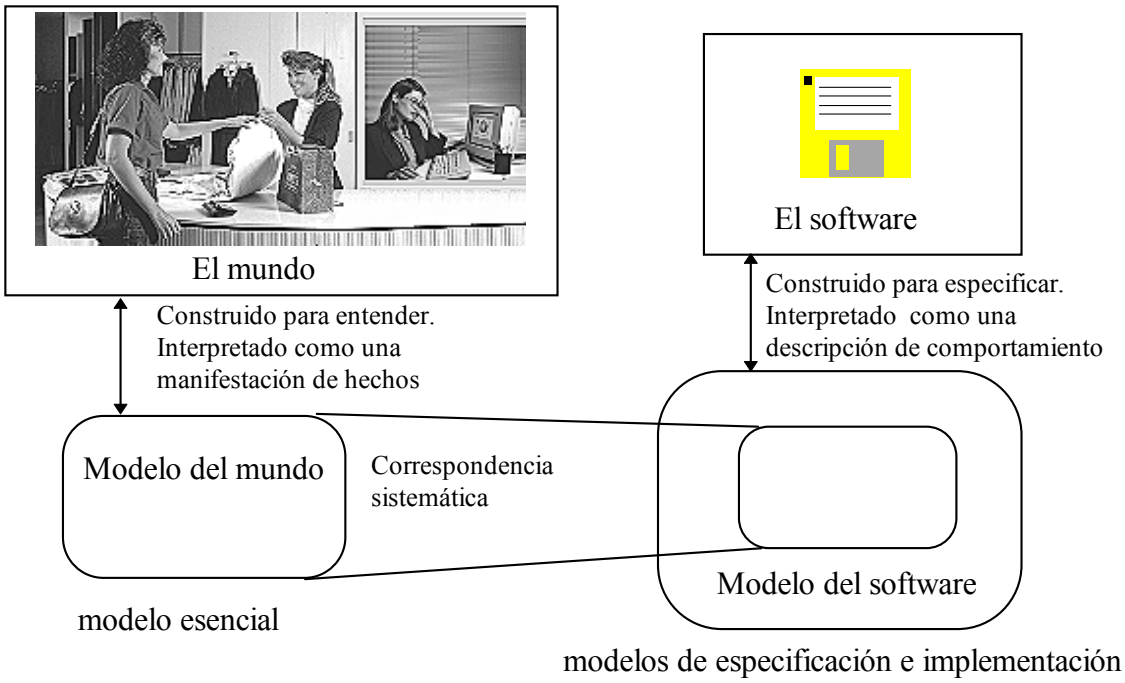


fig. 3.1 Diferentes modelos en el proceso de desarrollo

Se continúa el proceso de desarrollo transformando el modelo anterior hacia un modelo más elaborado que incluye decisiones de implementación. Este es denominado *modelo de*

[5] Algunos utilizan el término clase. Sin embargo, la mayor parte de las organizaciones normativas (como el OMG) consideran las clases como implementaciones de los tipos de objetos [Martin 1997]

implementación y trata con aspectos que establecen patrones de flujo de control dentro del software. Los bloques de construcción de este modelo consisten de objetos y mensajes. La interacción entre los objetos se describe en forma de mensajes enviados de un objeto a otro, presentándose la secuencia de dichos mensajes y el control de concurrencia.

3.2 Elementos del modelo de objetos

Las técnicas para establecer un modelo de objetos, marco de referencia de la orientación a objetos, según Booch, [Booch 1996] se basan en cuatro elementos fundamentales:

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía

Esto quiere decir que un modelo que carezca de estos elementos no es orientado a objetos. Existen tres elementos secundarios del modelo de objetos:

- Tipos (tipificación)
- Concurrencia
- Persistencia

Por secundarios quiere decirse que cada uno de ellos es una parte útil, pero no esencial, del modelo de objetos.

La importancia de estos elementos se manifiesta desde el inicio del proceso, donde se hace necesario identificar los fenómenos o entidades claves del dominio del problema, representándolos como abstracciones en el modelo inicial y los subsecuentes. Entendemos por *abstracción* aquello que nos permite concentrarnos en los aspectos esenciales (inherentes al fenómeno o entidad que está siendo modelado), ignorando sus propiedades accidentales.

El efecto de aplicar el concepto de *encapsulamiento* al objeto se manifiesta, por un lado, al ubicar dentro del objeto tanto sus características o propiedades como las operaciones que realiza, y por otro lado escondiendo los detalles de implementación de sus componentes.

Debido a que aún en las aplicaciones triviales puede haber muchas más abstracciones de las que sea posible comprender simultáneamente, la *modularidad* ofrece mecanismos para agrupar abstracciones relacionadas lógicamente. Esto, sin embargo aún no es suficiente [Booch 1996], pues frecuentemente un conjunto de abstracciones forma una jerarquía, y la identificación de esas jerarquías en el diseño simplifica en gran medida la comprensión del problema. La *jerarquía* es una clasificación u ordenamiento de las abstracciones. Las dos jerarquías más importantes es un sistema complejo son su estructura de clases (relación de herencia o jerarquía de clases) y su estructura de objetos (jerarquía de partes).

A medida que se desarrolla la jerarquía de clases, la estructura y comportamiento comunes a diferentes clases tenderá a migrar hacia superclases comunes. Por esta razón se habla a menudo de la herencia como una jerarquía de generalización/especialización. El ignorar las jerarquías de clase que existan puede conducir a diseños deformes y poco elegantes, apunta Booch.

3.3 Vistas

Es imposible captar los detalles sutiles de un sistema de software complejo en una sola vista. Se deben "comprender": la estructura taxonómica de las clases, los mecanismos de herencia utilizados, los comportamientos individuales de los objetos y el comportamiento dinámico del sistema en su conjunto.

El problema es análogo a tener el concepto de una pieza que se desea construir, tal como la que se presenta en la fig. 3.2 (a). Con la representación de la pieza en una sola vista no sería posible su construcción, ya que el constructor no sería capaz de comprenderla en su totalidad. Un caso semejante se muestra en la fig. 3.2 (b); por ser una pieza más compleja no basta tener dos vistas, para su total comprensión se hace necesario representarla por medio de tres vistas.

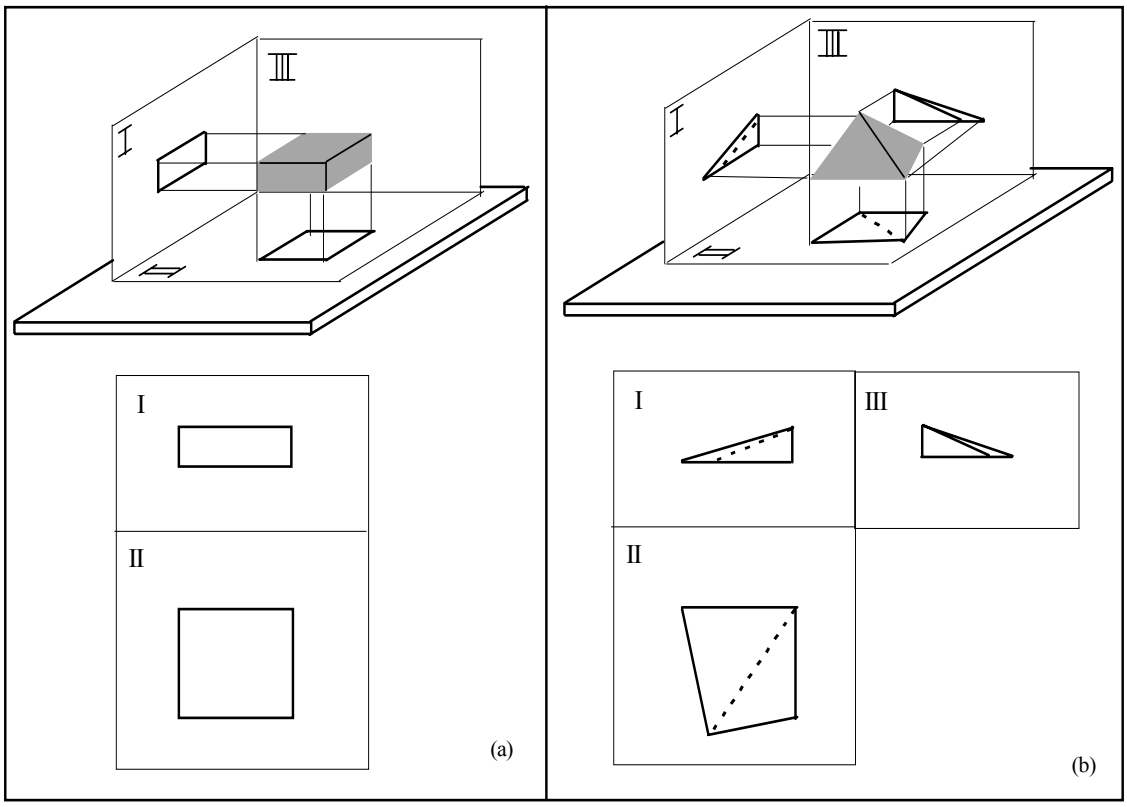


fig. 3.2 Necesidad de múltiples vistas

La analogía presentada nos sirve para valorar la necesidad de contar con diversas vistas (dependiendo de la complejidad) que describan apropiadamente el sistema de software, de tal forma que pueda ser implementado correctamente. La fig. 3.3 muestra la representación de un modelo usando múltiples vistas.

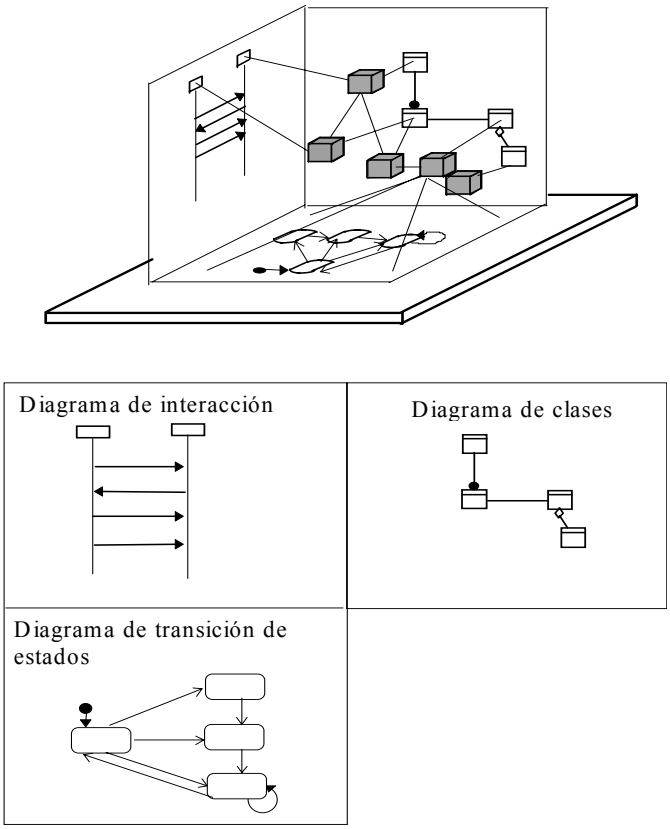


fig. 3.3 Modelo representado por múltiples vistas

Las diferentes vistas deben mostrarnos tres aspectos esenciales de un modelo descrito utilizando técnicas orientadas a objetos. Cada aspecto enfatiza algunas características del sistema y suprime otras, logrando cada uno presentar una visión particular del sistema (fig.3.4). Los aspectos esenciales son:

- Estructura
- Comportamiento
- Arquitectura

La *estructura* nos describe las partes estáticas del sistema e ignora el procesamiento. Se construye inicialmente a partir de los objetos, clases, asociaciones y formas de interacción presentes en el problema, evolucionando paulatinamente al ir incorporando elementos orientados a la solución e implementación. Los diferentes métodos orientados a objetos ofrecen diagramas de estructura estática (fig. 3.5), los cuales se inspiran en el modelaje *entidad - asociación (entidad - relación).*

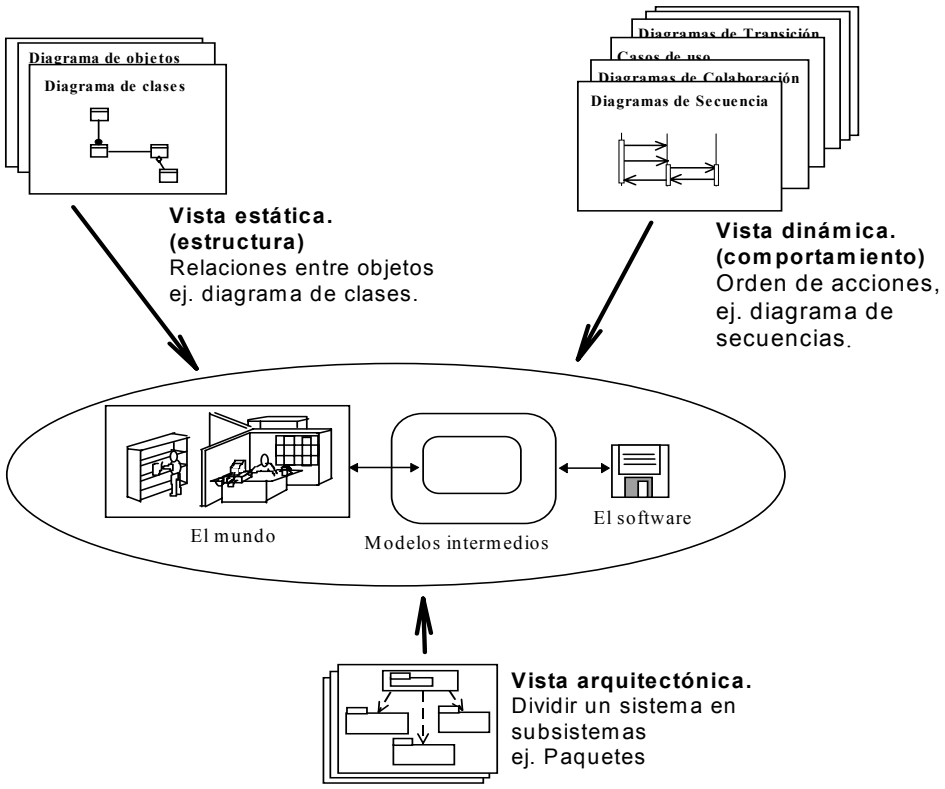


fig. 3.4 Tres vistas en un sistema

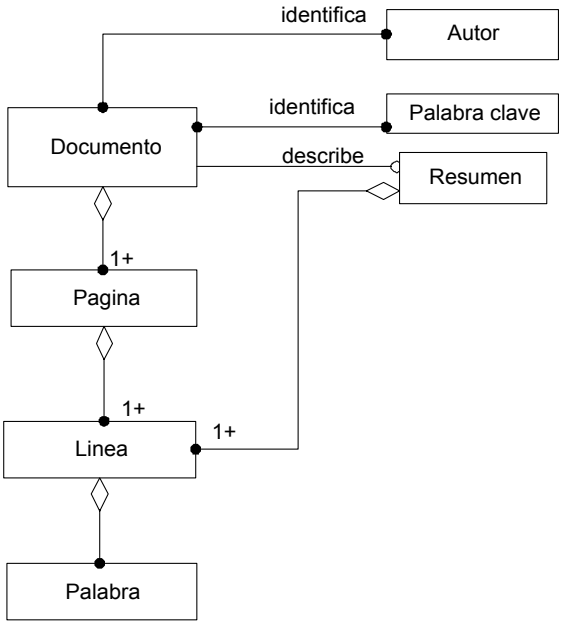


fig. 3.5 Diagrama de clases en OMT[6]

[6] Object Modeling Technique [Rumbaugh 1991]

El *comportamiento* describe la evolución dinámica del sistema y la forma como colaboran grupos de objetos para llevar a cabo acciones con un efecto esperado. Esta vista define los servicios útiles que ofrece el sistema a su entorno, por medio de la asignación de responsabilidades a las diferentes clases de objetos. Incluye cambios en el estado del sistema, secuencias de eventos, y entradas/ salidas externas. El modelo dinámico provee múltiples representaciones gráficas, las cuales ilustran diferentes perspectivas de la interacción de los objetos (fig. 3.6) tanto en el análisis como en el diseño, variando los niveles de abstracción con que se presentan.

La *arquitectura* nos muestra la descomposición de un sistema en subsistemas, permitiendo agrupaciones basadas en los objetos (como las *categorías de clases* del método de Booch fig.3.7). Puede distinguirse entre arquitectura interna y externa. La primera se refiere a la forma como se organiza el sistema internamente. La segunda se refiere a la forma como es percibido el sistema por los usuarios; para ella son útiles los casos de uso de Jacobson [Jacobson 1994].

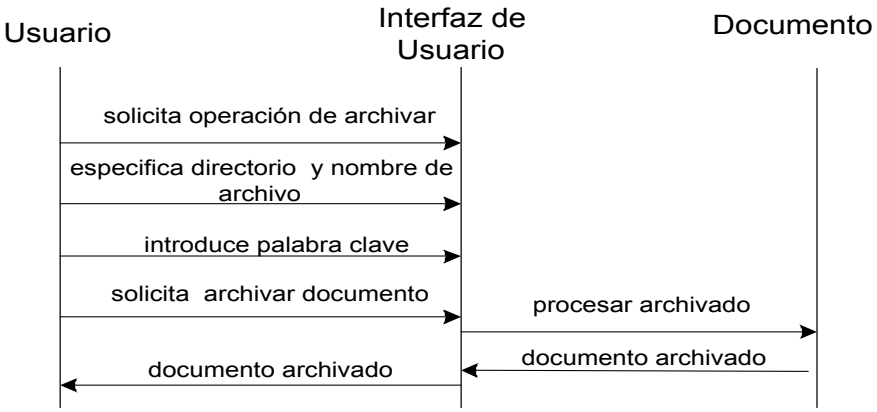


fig. 3.6 Rastreo de eventos en OMT

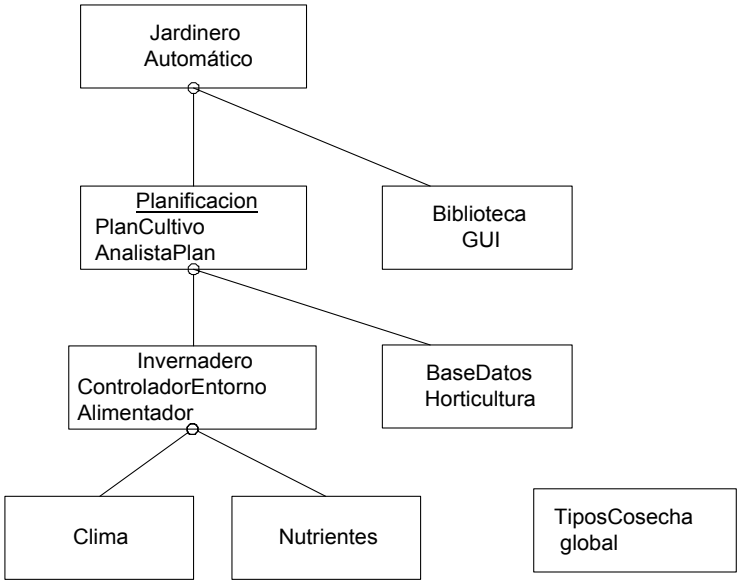


fig. 3.7 Categoría de clases (Método de Booch)

Dentro de los múltiples métodos para la construcción de modelos orientados a objetos, existen diversos enfoques basados en cuál aspecto se toma como punto de partida para guiar el proceso de desarrollo. Por estar éste tema ligado al análisis y diseño se abordará en el capítulo siguiente.

4 Análisis y diseño orientados a objetos

Las aplicaciones informáticas abarcan un amplio conjunto de categorías y es difícil establecer compartimientos nítidamente separados. Las aplicaciones tienden a crecer en tamaño y complejidad, y por ello son más difíciles de desarrollar. Su funcionalidad se desplaza del procesamiento de transacciones a la simulación de sistemas; de interfaces basadas en texto a basadas en gráficos y sistemas multimedios, de aplicaciones monousuario a multiusuario.

Estas exigencias tienden a complicarse por la importancia de contar con aplicaciones portátiles entre un gran número de plataformas (que cambian muy rápidamente) y la necesidad de facilitar el aprendizaje y entendimiento de diferentes tipos de usuarios.

La necesidad de desarrollar software confiable en menos tiempo exige un enfoque sistemático para el diseño de sistemas. Existen diversos enfoques y todos ellos apuntan a producir buenos sistemas. Pero, ¿a qué llamamos "buenos sistemas"? Como menciona Jacobson [Jacobson 1994]: "La pregunta ¿qué es un buen sistema? puede ser contestada en parte desde un punto de vista externo y en parte desde un punto de vista interno. El punto de vista externo proviene de quienes usan el sistema. Ellos desean un sistema que entregue resultados correctos rápidamente, que sea confiable, eficiente, fácil de aprender y usar. El punto de vista interno proviene de quienes desarrollan o dan mantenimiento al sistema. Desde dicho punto de vista se desea que el sistema sea: fácil de modificar y extender, fácil de entender, contenga partes reutilizables, fácil de probar, compatible con otros sistemas, portátil, poderoso y fácil de construir."

4.1 Modelos de procesos

En la búsqueda por diseñar buenos sistemas, han sido propuestos diversos enfoques o modelos de procesos [Goldberg 1995], los cuales establecen un orden para llevar a cabo un conjunto de actividades que guiarán el desarrollo de los sistemas. Podemos ver el modelo de proceso como una colección de máximas, estrategias, actividades, métodos y tareas que son organizados a fin de lograr un conjunto de metas y objetivos. Este conjunto de elementos facilitarán al gestor controlar el proceso de desarrollo del software y suministrar las bases para construir productivamente software de alta calidad. Jacobson, Booch y Rumbaugh definen proceso así: "El conjunto total de actividades necesarias para transformar los requerimientos de un cliente en un conjunto consistente de artefactos que representan un producto de software y - en momentos posteriores - transformar cambios en esos requerimientos en nuevas versiones del producto de software"[Jacobson 1999].

Entre los diferentes modelos de procesos propuestos, podemos mencionar:

- El modelo de cascada (o ciclo de vida clásico)
- El modelo de espiral
- El ciclo de vida orientado a objetos

Todo modelo de proceso consta de un conjunto de actividades que permiten cumplir con tareas que poseen un propósito específico. En general, las posibles actividades de estos modelos de proceso son: análisis, diseño, implementación, documentación, prueba y mantenimiento. Este

escenario de desarrollo sucede independientemente del modelo empleado; se puede caracterizar como muy turbulento al inicio, estabilizándose a medida que se avanza en el desarrollo.

El modelo de cascada (fig. 4.1) describe el flujo del proceso de desarrollo. El trabajo comienza creando la especificación de requerimientos del sistema. A partir de ella se lleva a cabo una descripción lógica del sistema que corresponde a la etapa de análisis, se prosigue con la etapa de diseño por medio de la cual la descripción en la etapa anterior se traduce en una representación del software de alto nivel donde la condición idealizada será gradualmente reemplazada por los requerimientos del ambiente de implementación que se ha de escoger. Sigue la etapa de implementación, posteriormente los diferentes módulos son probados individualmente e integrados.

Cuando la última prueba de integración ha sido completada el sistema puede ser probado y entregado para comenzar su evolución (etapa de mantenimiento).

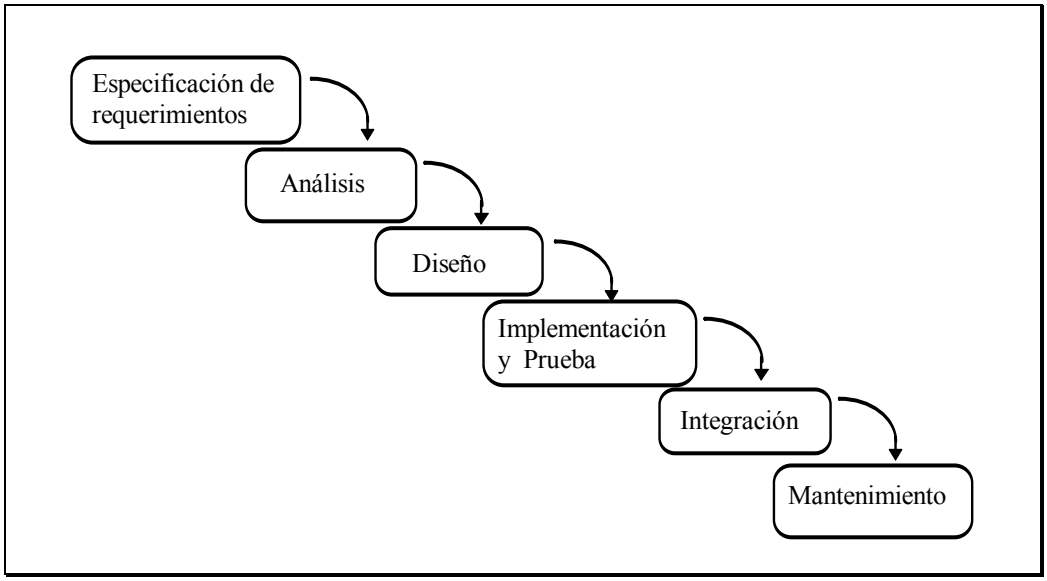


fig. 4.1 Modelo de cascada o ciclo de vida clásico

Inicialmente la idea fue que cada etapa debería completarse antes de comenzar la siguiente. Este principio fue eliminado rápidamente, permitiendo así que una fase comenzara antes que la anterior fuera terminada. El modelo de cascada ha tenido un gran efecto en la ingeniería del software, aunque nunca se proyectó su utilización literal cuando se introdujo.

El mayor problema, sin embargo, en la gran mayoría de los casos fue la fase de mantenimiento, lo que en realidad acarreaba nuevos requerimientos, análisis, diseño, etc. Se desarrollaron otros modelos con el fin de describir esos nuevos hechos; uno de los más populares es el modelo de espiral (fig.4.2) [Jacobson 1994].

El modelo de espiral puede ser descrito como el desarrollo de un producto para formar nuevas versiones. Con cada iteración alrededor de la espiral se van construyendo nuevas versiones del software cada vez más completas. Este modelo es un enfoque más realista para el desarrollo del software y de sistemas a gran escala. Utiliza un enfoque "evolutivo" para la ingeniería del

software permitiendo al desarrollador y al cliente entender y reaccionar a los riesgos en cada nivel evolutivo. Mantiene el enfoque sistemático correspondiente a los pasos sugeridos por el ciclo de vida clásico, pero incorporándola dentro de un marco de trabajo iterativo que refleja de forma más precisa el mundo real [Pressman 1993].

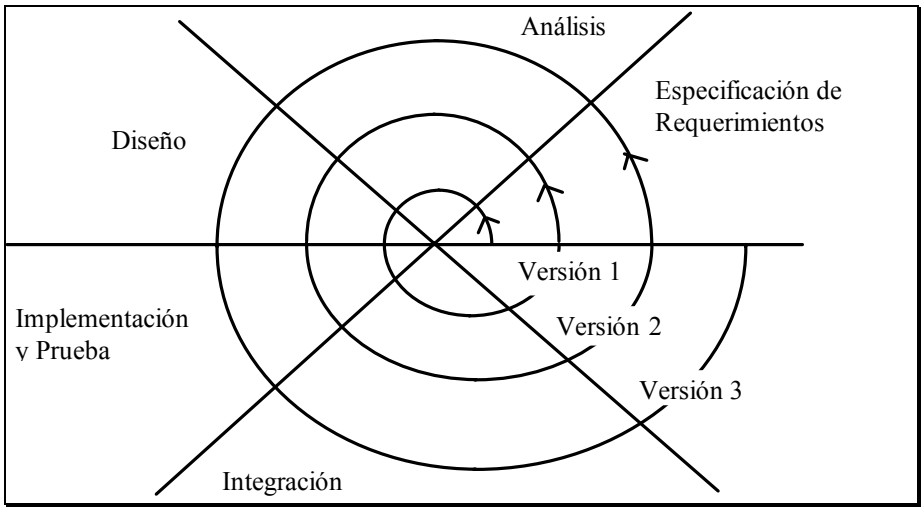


fig. 4.2 Modelo de espiral

El ciclo de vida orientado a objetos presentado en la fig. 4.3 refleja un esquema de desarrollo en el que una base de conocimiento representada por bibliotecas de componentes reutilizables y de fácil conexión impactan las diferentes fases de producción de modelos. La reutilización de clases y el proceso de generalización es un proceso iterativo el cual influye tanto el análisis como el diseño[Nerson 1992].

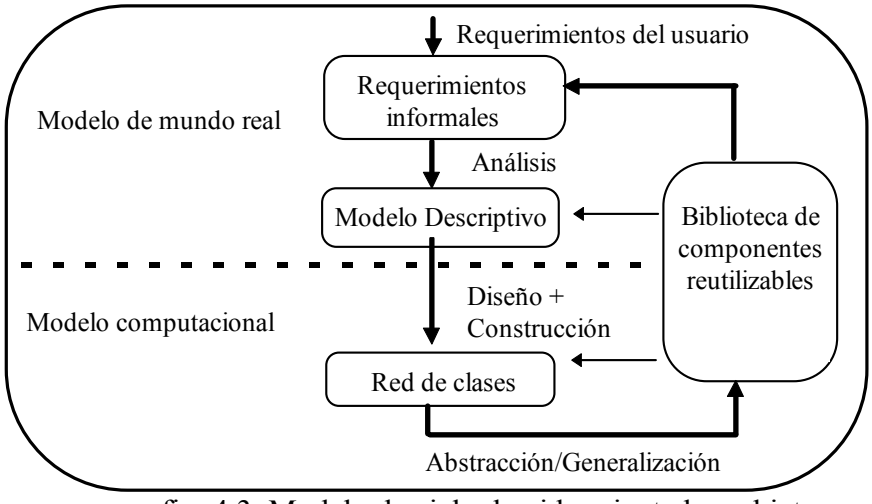


fig. 4.3 Modelo de ciclo de vida orientado a objetos

Los modelos se construyen mediante un proceso iterativo e incremental. Es iterativo en el sentido que conlleva el refinamiento sucesivo de una arquitectura orientada a objetos, al aplicar la experiencia y resultados de cada versión a la siguiente iteración del análisis y diseño. Es incremental ya que cada pasada por un ciclo de análisis, diseño, construcción lleva a refinar

gradualmente los modelos, convergiendo gradualmente hacia una solución que se encuentra fundamentada y cumple con los requerimientos reales del usuario final, siendo además comprensible, confiable y adaptable. Además, el ciclo de vida orientado a objetos, incorpora las ideas del modelo espiral: los incrementos se definen con base en el análisis de riesgos y así se producen versiones del producto de software.

4.2 ¿En qué consisten el análisis y diseño?

La fase de análisis se enfoca en **qué** debe hacer el sistema, más que en el **cómo** lo llevará a cabo. Un método de análisis consiste de conceptos, técnicas y pasos para construir un modelo del problema. La fig. 4.4 ilustra el proceso para conducir el análisis.

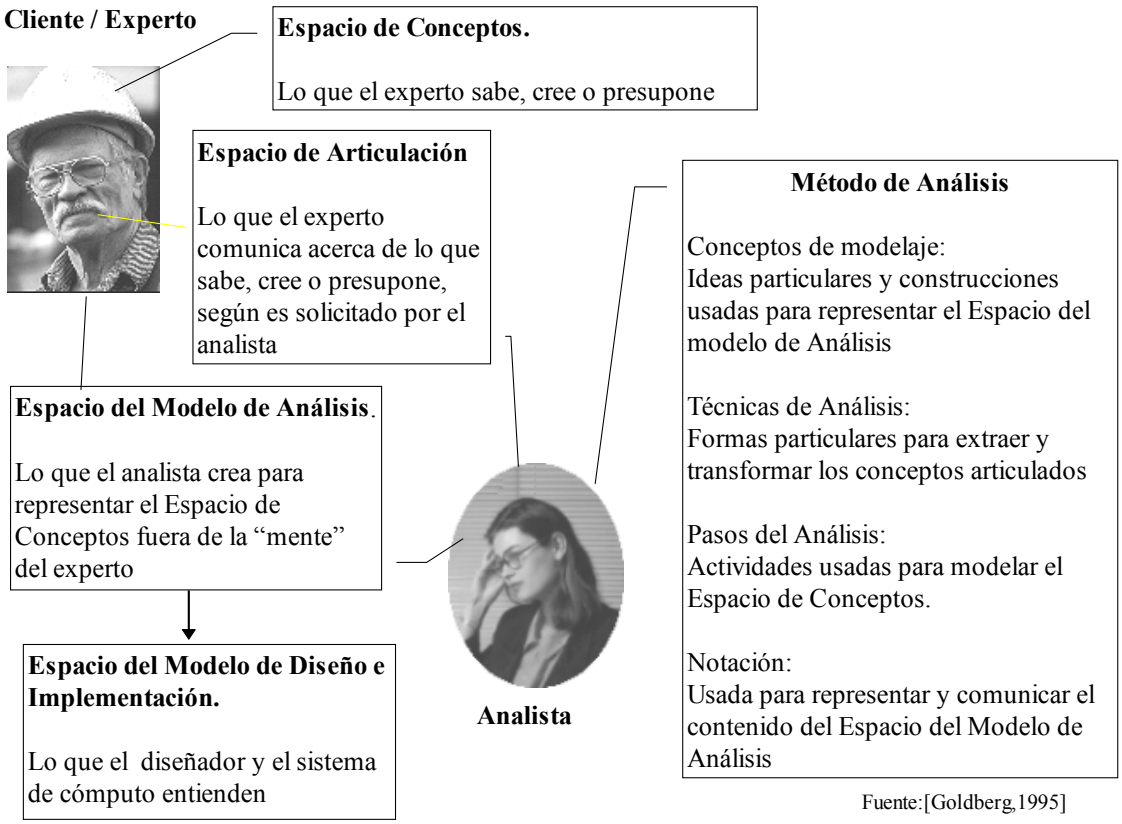


fig. 4.4 Proceso de Análisis

De acuerdo con la fig. 4.4 el análisis involucra a dos individuos (roles o papeles) a través de tres espacios de información: *Conceptos, Articulación* y *Modelo* [Goldberg 1995]. La primera persona es un usuario o cliente, experto quien se supone entiende el problema por ser resuelto.

Suponemos que el problema es bastante complejo y es mantenido en el Espacio de Conceptos, es decir en la mente del experto. El entendimiento del problema por parte del experto es amplio y muy asociativo. Cuando es el mismo quien desarrolla el sistema, la principal manifestación de los conceptos fuera de la mente del experto probablemente sea la solución directa.

El reto del análisis comienza cuando el experto debe comunicar los conceptos a alguien más, en este caso un analista, como lo describe la figura. Puesto que los conceptos son a menudo muy ricos y amplios, generalmente no es posible para el experto comunicar adecuadamente su completo entendimiento en una única expresión. Como resultado, estos expertos se ven forzados a proveer múltiples explicaciones, verbales o escritas. El analista induce y sugiere para poder captar dichas explicaciones (Espacio de Articulación) y reúne todas las partes juntándolas en una representación coherente (Espacio del Modelo de Análisis).

El análisis es particularmente difícil cuando el experto es incapaz de articular lo que sabe, lo que podría interpretarse como: lo que es bien entendido por el cerebro es pobremente expresado por la boca.

Los conceptos de modelado, técnicas de análisis y pasos de análisis representan en la fig. 4.4 el método de análisis que guía la forma en que el analista manipula el Espacio de Articulación y del Modelo de Análisis.

Las técnicas y pasos de análisis dirigen y guían al analista en la transformación de la información proveída por el experto en una representación del Modelo de Análisis en el Espacio del Modelo de Análisis. Se puede pensar del Espacio de Articulación como el espacio apropiado para el encuentro entre el experto y el analista.

Un objetivo fundamental del análisis es minimizar la diferencia entre el Espacio de Conceptos y el Espacio del Modelo de Análisis**.** Lo ideal sería que cada aspecto en el Modelo de Conceptos del experto pudiera tener una correspondencia directa en el Modelo de Análisis. El experto revisaría el Modelo de Análisis a fin de determinar si describe con precisión su propio entendimiento del problema. Si la distancia entre el Modelo de Análisis y el entendimiento del experto es muy grande, será bastante difícil, si no imposible, para el experto verificar su precisión. Consecuentemente, uno de los principales requerimientos para cualquier Modelo de Análisis es que sea entendible por el experto; el grado con que esto sea posible está directamente relacionado con la naturaleza de los conceptos de modelado y la notación usada para representar los diferentes aspectos del Modelo de Análisis.

El diseño consiste en dos actividades: diseño arquitectónico y diseño detallado. En el diseño arquitectónico se toman decisiones estratégicas acerca de cómo la funcionalidad del sistema se distribuye entre componentes independientes, cómo se relacionan dichos componentes, y cómo se transfiere el control de componente a componente. A menudo se incluye una especificación de cómo los usuarios dan y reciben información, y cómo el sistema se comunica con otros sistemas. El diseño detallado conduce a decisiones tácticas, tales como la escogencia de algoritmos, estructuras de datos y protocolos de comunicación. La fig. 4.6 indica que el proceso continúa del análisis al diseño y la implementación. Los desarrolladores cambian el modelo de análisis agregando restricciones de implementación, tal como la escogencia de alguna plataforma (herramienta o lenguaje de desarrollo, bibliotecas de implementación), el estilo de presentación para el usuario, o el sistema manejador de bases de datos.

4.3 ¿Por qué análisis y diseño orientados a objetos?

A medida que la complejidad del software se incrementa se ponen de manifiesto las debilidades que poseen y los problemas que acarrean los métodos tradicionales para el desarrollo de software. La falta de un modelo unificado que guíe e integre las diferentes fases impide transiciones armoniosas; por el contrario, estas transiciones consumen tiempo y a menudo reducen la calidad del producto final. La información desarrollada en el proceso de análisis sirve como entrada a la fase de diseño, pero en una terminología y notación completamente diferentes, las cuales representan otra perspectiva a partir de la fase de diseño. Por esta razón, tales enfoques incluyen lo que pueden ser fronteras innecesarias entre el análisis y el diseño [Korson 1990]. Estas fronteras son el resultado de un cambio del dominio del problema (en el análisis) al dominio de la solución (en el diseño), lo que da lugar a pérdida de información y aparición de incomprensiones (fig. 4.5).

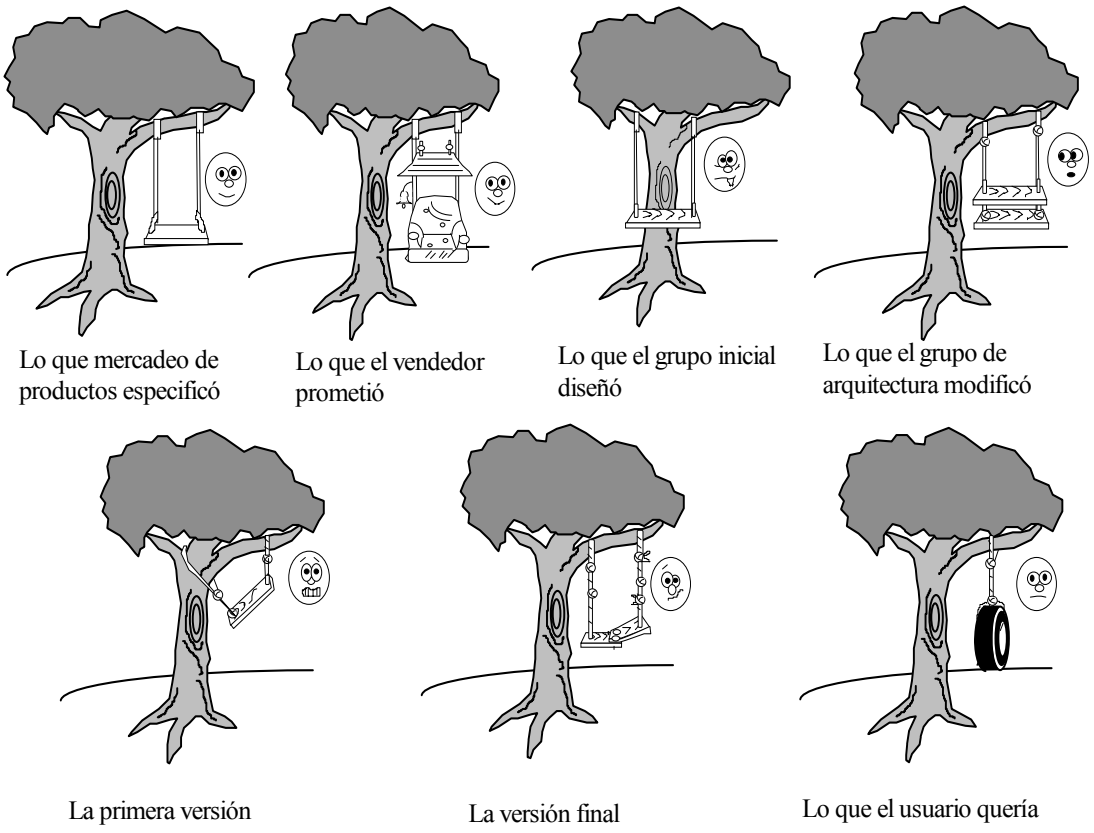


fig. 4.5 Problemas en la transición de una fase a otra

Por el tipo de descomposición que se lleva a cabo en los enfoques tradicionales, la estructura conceptual del problema, plasmada en el modelo de análisis, se realiza mediante herramientas como diagramas de flujo de datos, tablas de decisión y español estructurado. Estos no permiten que se establezca una comunicación fluida entre el analista y el usuario (experto), debido a que "no hablan el mismo lenguaje", de tal forma que si el usuario no ha sido capacitado para entender el modelo, no será capaz de verificar con precisión su propio entendimiento. Como se mencionó anteriormente, existirá una gran brecha entre el espacio de conceptos del experto y el espacio del modelo de análisis.

Por otro lado, los enfoques tradicionales hacen poco énfasis en la reutilización, ya que cada sistema tiende a ser construido a partir de cero. Eso, aunado a una estructuración inestable, trae como consecuencia que los costos de mantenimiento representen el mayor porcentaje del costo total de los sistemas. Sumado a esto, tenemos que los sistemas basados en tales enfoques son susceptibles de modificaciones proporcionalmente importantes; al respecto expresa Jacobson [Jacobson 1994]: "los sistemas de software construidos con los métodos basados en funciones y datos han probado ser como casas hechas de naipes, pequeños cambios han tenido consecuencias significativas[7]."

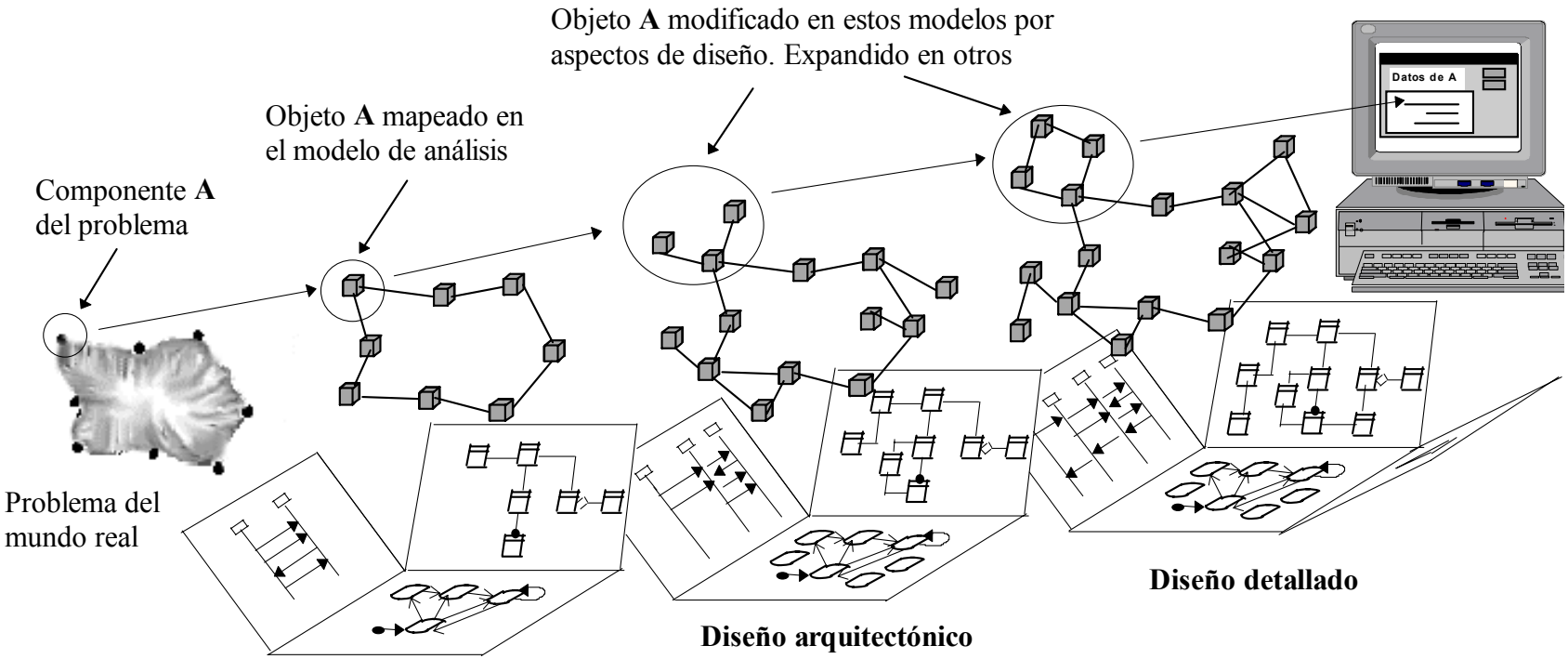
Bajo el enfoque orientado a objetos no se lleva a cabo una descomposición funcional, sino que se plantea una descomposición alternativa, mediante la cual se identifican las abstracciones claves del dominio del problema, las cuales son representadas por objetos. Estos objetos son los componentes esenciales de la estructura del sistema. Mediante las relaciones que se crean entre ellos se establece la red de comunicación por la que circula la información (de control y datos) con la cual se consigue la funcionalidad que debe proveer el sistema. Esta funcionalidad queda asignada, distribuida y encapsulada en los objetos.

Entre los beneficios que se pueden obtener al llevar a cabo el proceso de desarrollo de sistemas por medio de este enfoque se encuentran:

- El modelo resultante de cada fase consiste siempre de objetos y sus interrelaciones. El análisis y el diseño tienen mucho en común: los mismos conceptos orientados a objetos, técnicas y notaciones usados en el análisis se aplican igualmente en el diseño, por lo que la transición de una fase a otra es tan natural, que a veces es difícil especificar el punto final del análisis y el inicial del diseño (fig. 4.6).

- Los modelos que construimos en el análisis orientado a objetos reflejan la realidad de modo más natural que los del análisis tradicional de sistemas [Martin 1994]. Se dice que es natural porque las piezas que componen la estructura del sistema se corresponden muy cercanamente con los conceptos del mundo real, los cuales modelan. Esto permite una comunicación con los usuarios en términos comprensibles, que puede incluso ayudarles a ser creativos con respecto de sus necesidades.

[7] Los autores consideran, en su experiencia, que los sistemas estructurados alrededor de los datos tienden a ser más estables que los estructurados alrededor de las funciones.



Fase de análisis

Modelo simplificado que capta la semántica del dominio del problema en términos de los objetos relacionados y su representación por medio de diagramas que muestran su estructura y comportamiento.

Fase de diseño

Modelos que representan el dominio de la solución. Afectados por aspectos de especificación e implementación.

fig. 5.6 Modelos en el análisis y diseño.

- La capacidad de la orientación a objetos de representar conceptos con un alto grado de abstracción puede conducir a crear sistemas ensamblables, como en las demás ingenierías, utilizando bloques preconstruidos (conceptos), muchos de los cuales estarán disponibles en bibliotecas de componentes o derivarse a partir de ellos, lo cual potencia la reutilización [Sánchez 1995].

- El proceso que conduce a la construcción de arquitecturas orientadas a objetos tiende a ser iterativo e incremental. En cada iteración se agrega nueva información o detalle, lo cual provee retroalimentación que conduce al refinamiento de los modelos producidos. Una o varias iteraciones pueden ser tratadas como un incremento del sistema, es decir, un paso en la evolución del sistema, tendiendo a converger hacia la solución que cumpla con los requerimientos de los usuarios.

4.4 Análisis y diseño orientados a objetos

Las fronteras entre análisis y diseño con objetos son difusas, a pesar de que el objetivo principal de ambos es diferente. Esto se debe esencialmente a que se aplican los mismos conceptos, técnicas y notaciones tanto en el análisis como en el diseño; lo que varía son los dominios a que pertenecen los objetos sobre los cuales se aplican. Mientras en el análisis el enfoque va para los objetos en el dominio del problema, en el diseño se dirige hacia los objetos en el dominio de la solución, o como escribe Booch: “En el análisis se persigue modelar el mundo describiendo las clases y objetos que forman el vocabulario del dominio del problema, y en el diseño se inventan las abstracciones y mecanismos que proporcionan el comportamiento que este modelo requiere”.

La falta de una clara distinción entre el análisis y el diseño conduce a los diversos autores a dirigir el énfasis de las descripciones de sus métodos hacia una o todas las actividades que componen el proceso de desarrollo. Algunos se enfocan en el análisis, otros en el diseño, los hay quienes enfatizan el prototipaje y los que toman en cuenta todas las fases, también denominado *enfoque multimodelo*. Así observamos en métodos como el de Coad-Yourdon se introducen en la fase de análisis vistas que autores como Booch utilizan en el diseño.

Como apunta Fowler [Fowler 1995], las ventajas y desventajas que comprenden estas decisiones son variadas, pero en general se pueden mencionar las siguientes:

La ventaja del enfoque multimodelo es que al separar modelos detallados para el análisis y diseño, se pueden llevar a cabo diferentes tipos de decisiones en cada uno de los modelos. Aspectos conceptuales pueden ser manejados en el modelo de análisis, mientras que aspectos de implementación se separan en el modelo de diseño. Esto desliga la complejidad propia del dominio del problema de la complejidad que introduce la tecnología. Sin embargo, como estos modelos no son independientes sino que están íntimamente ligados, cambios que se presenten en uno de los modelos se deberán reflejar en sus contrapartes. Esto puede conducir a profundos y complicados procesos para controlar los cambios en diferentes lugares dando como resultado una seria sobrecarga. Si los cambios no se manejan apropiadamente aparecerán incompatibilidades y discrepancias entre los diversos modelos.

Una alternativa es enfatizar la etapa de diseño. La justificación de esta alternativa radica en sopesar la sobrecarga de mantener el modelo de análisis contra el valor del modelo mismo, particularmente cuando el modelo de análisis, de alguna forma, guarda una estrecha relación con el software. Esto no significa que desaparece el modelo de análisis sino que se lleva a cabo un bosquejo y a partir de éste se elabora un modelo detallado de diseño. Esto produce un modelo de diseño que se encuentra muy cercano al código, y donde la notación es próxima a los conceptos en los lenguajes de programación.

Otra alternativa viable es poner énfasis en el análisis. Lo más importante de este enfoque es que al centrar su atención en el modelo inicial, se resalta la estructura conceptual del sistema. Este modelo se transforma posteriormente a código y puesto que no se mantiene un modelo de diseño separado, este puede ser generado por el modelo de análisis y reglas de transformación. El principal problema con este enfoque radica en la necesidad de reglas de transformación. Pobres reglas de transformación conducirán a crear un software ineficiente y falto de elegancia, difícil de entender. A esto se le puede agregar lo difícil que resulta eliminar del modelo aspectos de implementación. Esto se debe a que existen decisiones de diseño que tienen que documentarse por algún lado, ya sea en un documento separado, en el código o haciendo anotaciones en el modelo de análisis (lo cual no es deseable porque viene a contaminar dicho modelo).

Existe otra corriente de pensamiento que sostiene que los modelos de análisis y diseño muy elaborados son deseables solamente en aquellas situaciones que por su complejidad así lo demanden, por lo que es mejor enfocar el proceso de desarrollo en construir un bosquejo de la estructura global del sistema y utilizar prototipos para trabajar sobre los detalles. Esto conduce a modelos de análisis y diseño muy simples, que son transformados en prototipos donde se toman en cuenta los detalles, muchos de los cuales serán definidos y probados en la implementación.

La principal desventaja de esta corriente radica en el hecho que el modelo se encuentra limitado al lenguaje utilizado para construir el prototipo. Puede ser difícil obtener una visión global del modelo y posiblemente sea más complicado realizar cambios en ese modelo que en uno conceptual en el papel.

En la mayoría de los métodos se usan los diferentes enfoques, unos promueven el multimodelo, otros hacen una mezcla dándole más peso al análisis o al diseño o, aunque hay quienes promueven el prototipado, no lo emplean en su forma pura, sino apoyando los otros modelos.

4.4.1 Consideraciones en cada fase

La esencia de la fase de análisis se centra en poder describir **qué** debe hacer el sistema. Con ese objetivo en mente y a partir de ciertas heurísticas[4], se llevan a cabo los siguientes pasos:

- Abstraer los aspectos esenciales del dominio de la aplicación y representarlos en un modelo.
- El modelo contiene lo encontrado en el dominio de la aplicación: objetos, clases, asociaciones, eventos e interacciones.
- Se construyen los modelos estructural, de comportamiento y arquitectura.

[4] Ver adelante estas heurísticas

En la fase de diseño, cambia nuestra atención, centrándonos en **cómo** el sistema resolverá el problema planteado. Para ello se realizan las siguientes acciones:

- Se toman decisiones de representación (datos) y manipulación (algoritmos) y se añaden detalles al modelo, para describir y optimizar una eventual implementación.
- El modelo de diseño expresa los objetos del dominio de la aplicación en términos de los objetos del dominio computacional.
- Sirven de apoyo las tarjetas CRC[5], el refinamiento y la descomposición.

En la fase de implementación, el modelo de diseño se expresa en términos de lenguajes de programación, bibliotecas de operaciones, bibliotecas de componentes, bases de datos, protocolos de comunicación, hardware etc.

4.4.2 Heurísticas que guían el proceso

La mayoría de los métodos orientados a objetos poseen heurísticas que apoyan el proceso de desarrollo. El modelaje es guiado por algún aspecto que se utiliza como fuente primaria para descubrir las clases y objetos. Estas heurísticas varían según se enfoque el modelaje hacia los datos del modelo, hacia los eventos o a los escenarios. Diferentes desarrolladores poseen preferencias hacia dónde enfocar la creación de modelos, sin embargo, esto también depende del dominio de aplicación del sistema por ser construido. Es conveniente conocer acerca de los diversos enfoques, pues un modelador experimentado producirá modelos más robustos al considerar más integralmente las múltiples facetas de un sistema.

4.4.2.1 Enfoque guiado por los datos del modelo

Respecto del enfoque del modelaje hacia los datos del modelo, diversos autores lo mencionan de distintas formas. Booch se refiere a este enfoque como “enfoque clásico” porque derivan sobre todo de los principios de la categorización clásica. Goldberg lo ilustra como “resaltar los requerimientos” debido a que las clases y objetos se derivan de los requerimientos del dominio del problema. Lo que indica este enfoque es que después de leer las especificaciones de requerimientos se subrayen nombres, verbos y adjetivos. Se supone que los nombres corresponderán a objetos del sistema, los verbos a servicios que los objetos deberán prestar y los adjetivos serán atributos de los objetos.

Diversos autores presentan diversas fuentes para encontrar clases y objetos. Booch lista las fuentes propuestas por los autores correspondientes (Tabla 4.1).

Este enfoque conduce el proceso de análisis a través del desarrollo de la vista estructural; es decir este es un enfoque conducido por la estructura estática del sistema.
Después de haber identificado los tipos de objetos, los subtipos, supertipos, asociaciones, objetos compuestos y atributos, se construye la vista estructural modelando estos elementos. Una vez identificadas las operaciones que resultan ser las generadoras de eventos que provocan los

[5] CRC (Class, Responsibility, Collaborators). Ver fig. 4.12.

cambios de estados en los objetos, se desarrollan diagramas de estados para cada clase con un comportamiento no trivial.

Shlaer - Mellor	**Ross**	**Coad-Yourdon**
Cosas tangibles	Personas	Estructuras
Papeles (roles)	Lugares	Otros sistemas
Eventos	Cosas	Dispositivos
Interacciones	Organizaciones	Eventos recordados
	Conceptos	Papeles desempeñados
	Eventos	Posiciones
		Unidades de organización

Tabla 4.1 Fuente para encontrar objetos y clases

Cuando el modelo ha crecido hasta contener un gran número de abstracciones, se identifican agrupamientos de clases tales que permitan su organización, a fin de establecer la arquitectura lógica del sistema

Los modelos creados son refinados orientándolos hacia una solución técnica mediante un proceso iterativo. Se inventan las abstracciones, mecanismos y estructuras colaborativas que proporcionan la infraestructura técnica que satisfaga el comportamiento que el modelo requiere.

fig. 4.7 Entidades susceptibles de representar clases y objetos

Algunos problemas se presentan bajo este enfoque. Goldberg expone: “dos hechos debilitan esta técnica. Primero se supone que existe una especificación completa, formal y correcta. Esto no es totalmente cierto, especialmente para grandes sistemas. Por el contrario, muchas veces la ausencia de una clara especificación de requerimientos es el motivo por el cual se escoge la

orientación a objetos como paradigma de desarrollo. Segundo, subrayar nombres y verbos en algunos documentos probablemente no siempre dará resultados de diseño satisfactorio debido a que muchos objetos son abstractos y pudieran no aparecer de forma explícita en las descripciones escritas. Bajo esta técnica se ofrecen pocas pautas sobre cómo identificar objetos que representan políticas intangibles, sincronizadores y coordinadores que gobiernan una situación técnica o de negocio.

Mientras este enfoque se centra en cosas tangibles del dominio del problema, dándole peso a la vista estructural, existen otros enfoques que se centran en el comportamiento como fuente primaria de clases y objetos.

4.4.2.2 Enfoque guiado por escenarios

Un escenario es una secuencia de acciones que toma lugar en el dominio del problema bajo consideración y que tiene como objetivo proveer algún resultado o valor tangible para el usuario. Construir escenarios ayuda tanto a determinar el comportamiento que se espera brinde el sistema como a asignar responsabilidades a los diferentes objetos para conseguir tal comportamiento.

Los escenarios pueden presentarse de dos formas: como escenarios específicos o como escenarios generales [Fowler 1995]. Un escenario específico podría expresar: "El hotel Flamingo de Oro ordenó un pedido de 112 botellas de ron, 50 botellas de tequila y 80 botellas de whisky", en este caso el escenario se refiere a objetos individuales. Un escenario general en cambio se refiere a tipos: "Un cliente ordenó un pedido. Un pedido consistirá de un número de líneas para producto, cada una de la cuales tiene un producto y una cantidad." En cada caso el escenario describe, en un cierto nivel de detalle, un uso real del sistema.

Los métodos precursores de este enfoque son OBA (Object Behavior Analysis) de ParcPlace Systems [Rubin1992] y Objectory de Jacobson [Jacobson1994]. OBA utiliza escenarios específicos y denomina "**guiones**[6] " a su técnica para construir los escenarios. Los principios de los guiones se fundamentan en el concepto de que una forma en que las personas almacenan su comprensión de un evento en el Espacio de Conceptos es por medio de memorias de episodios [Goldberg 1995]. Memoria de episodios es una secuencia de eventos que forman una historia o episodio. Al captar la historia descrita por el experto, el analista determina los roles o responsabilidades requeridas para representar el conocimiento del experto. El analista describe esta historia por medio de guiones. Un guión en OBA consiste de un conjunto de contratos. Cada contrato es un acuerdo entre dos roles, el iniciador y el participante. El iniciador es responsable por ejecutar una acción, y el participante es responsable por proveer el servicio. En la fig.4.8 se presenta un fragmento de un guión.

Durante el diseño, se les asigna a los objetos los papeles que desempeñarán y las responsabilidades a su cargo, con el fin de obtener el comportamiento que el sistema deberá proveer. Los resultados deberán describirse en alguna de las notaciones conocidas.

Nombre del Guión:

[6] Scripting en la documentación en inglés.

	Autor: Versión: Precondición: existe (Hoja de cálculo), mostrada (Hoja de cálculo) Postcondición: modificada (Hoja de cálculo) Rastreo: Modificación		
Iniciador	**Acción**	**Participante**	**Servicio**
Usuario	Seleccione D1	Hoja de Cálculo	Seleccione una celda
Usuario	Escriba el texto NUEVO	D1	escriba el contenido
Usuario	Establezca estilo a negrilla	D1	establezca estilo a negrilla
Usuario	Seleccione A2	Hoja de Cálculo	seleccione una celda
Usuario	Escriba el texto NOMBRE	A2	escriba el contenido
	(repita seleccione y escriba texto)	B2, C2, D2, A3 hasta A10	
Usuario	Seleccione fila 2	Hoja de Cálculo	seleccione una fila

fig. 4.8 Fragmento de un guión para modificar una hoja de cálculo

Objectory tiene objetivos similares, pero por su enfoque hacia los escenarios generales se concentra en tipos de objetos. Las secuencias de acciones son captadas como *casos de uso*[7], y los objetos que residen fuera del sistema se denominan *actores* (fig.4.10). Objectory identifica tres tipos de objetos: interfaz, entidad y control. Los objetos de interfaz traducen las acciones de un actor a eventos y los eventos del sistema a resultados que un actor pueda entender. Los objetos entidad modelan la información del sistema. Los objetos de control llevan a cabo aquellos servicios que no son fácilmente alojables en los otros dos tipos (fig.4.11). Esta clasificación facilita el planteamiento de modelos de análisis más robustos y evolucionables.

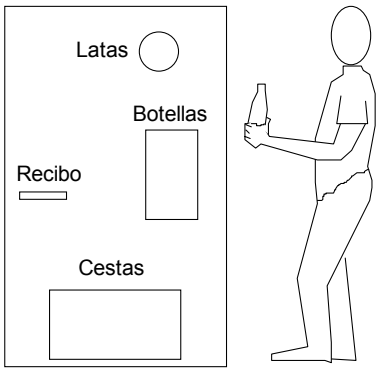


fig. 4.9 Máquina de reciclaje

En la fig. 4.9 se presenta el ejemplo de una máquina que recupera envases retornables en un supermercado (tomado de [Jacobson 1994]). El actor principal es el cliente que deposita los envases. La máquina, después de recibirlos, le entrega un recibo con el importe que le corresponde. En la fig.4.10 se muestran los casos de uso relativos al ejemplo. Se presenta también una descripción del caso de uso "Retornar artículo" y un diagrama de objetos, con los respectivos objetos de control, entidad e interfaz que representan la estructura de los componentes..

[7] Use case en la documentación en inglés.

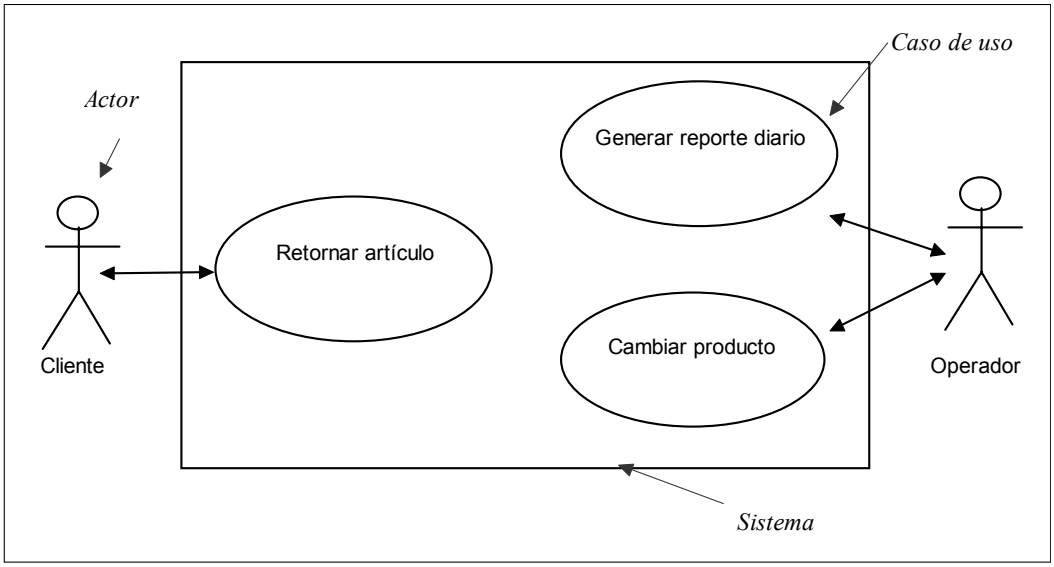


fig. 4.10 Casos de uso de la máquina de artículos retornables

caso de uso: **Retornar artículo**

Cuando el cliente retorna un artículo, éste es medido por el sistema.
La medida es usada para determinar que clase de artículo ha sido depositado.
Si es aceptado, el total de artículos para ese cliente y el total de artículos de cada tipo para ese día se incrementan.
Si el artículo no es aceptado, un mensaje NO VALIDO se ilumina en el panel.
Cuando el cliente presiona el botón de recibos, la impresora imprime la fecha.
El total del cliente es calculado. y la siguiente información se imprime en el recibo:

- Nombre
- Número de artículos retornados
- Valor de depósitos
- Total para cada tipo

Finalmente, se imprime el importe que el cliente debe recibir.

El análisis de los casos de uso se puede utilizar a partir del análisis de requerimientos, momento en que los participantes (usuarios finales, expertos del dominio, desarrolladores) enumeran los escenarios fundamentales para el funcionamiento del sistema. Estos escenarios en conjunto describen las funciones del sistema. A medida que se va analizando cada escenario, se deben identificar los objetos que participan, las responsabilidades de cada objeto, y cómo estos objetos colaboran entre sí, en términos de las operaciones que invoca uno sobre otro. Continuando con el proceso de desarrollo, estos escenarios se expanden para considerar condiciones excepcionales así como comportamientos secundarios del sistema. Los resultados de estos escenarios se van formalizando en los diagramas de clases, de interacción o de eventos.

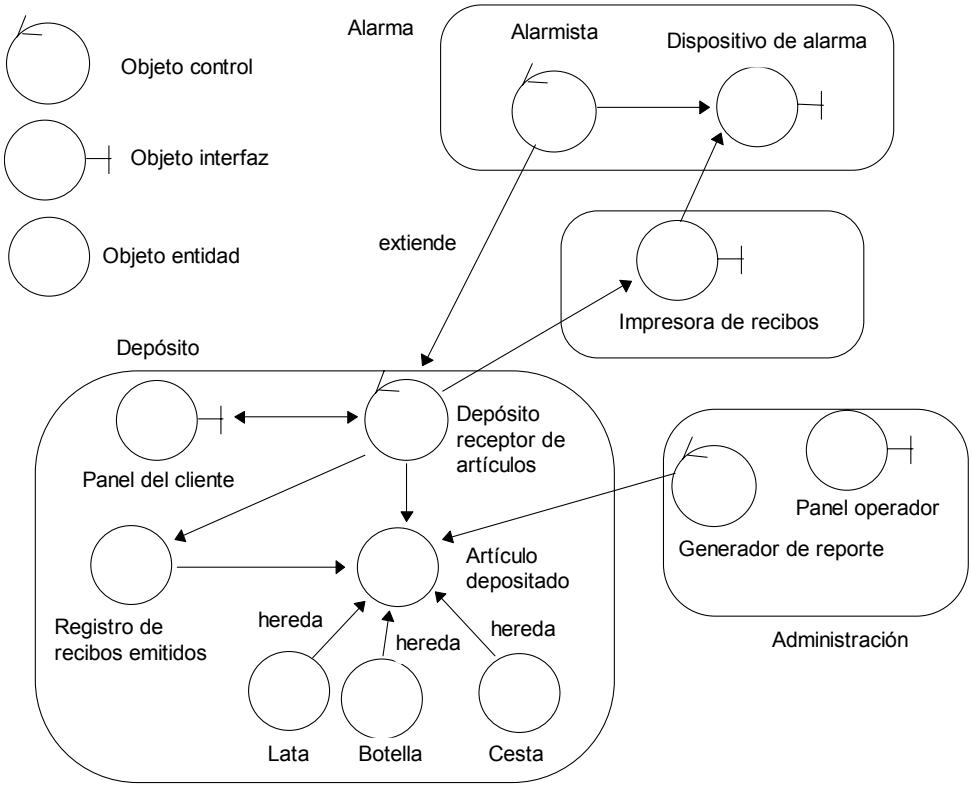


fig. 4.11 Diagrama de objetos

Por su importancia los escenarios han sido incorporados en diferentes métodos (OMT, Booch) y en la notación UML. Estos se tratarán un poco más en el informe referente a UML.

Se mencionan en este enfoque las **tarjetas CRC** porque, como menciona Booch, han resultado ser una herramienta efectiva para analizar escenarios. Ayudan a disponer un cierto orden y organización entre las clases que se relacionan, además ayudan en la comunicación entre el equipo de desarrollo. Una tarjeta CRC está compuesta de 3 partes (fig. 4.12): el nombre de la clase, sus responsabilidades y las clases que colaboran con ella [Budd 1994]. Usualmente sus dimensiones son las de una ficha de cartulina (10 x 15 cms).

Cada tarjeta identifica una clase que resulte relevante en el escenario. A medida que se avanza en el análisis del escenario se pueden asignar nuevas responsabilidades a una clase ya existente, agrupar determinadas responsabilidades y crear una nueva clase, o dividir las responsabilidades de una clase en otras y asignarlas a una clase diferente.

Refiriéndose a esta herramienta Booch expresa: "Las tarjetas CRC pueden disponerse espacialmente para representar patrones de colaboración. Desde el punto de vista de la semántica dinámica del escenario, las tarjetas se disponen para mostrar el flujo de mensajes entre instancias prototípicas de cada clase; desde el punto de vista de la semántica estática del escenario, las tarjetas se colocan para representar jerarquías de generalización/especialización o de agregación entre las clases."

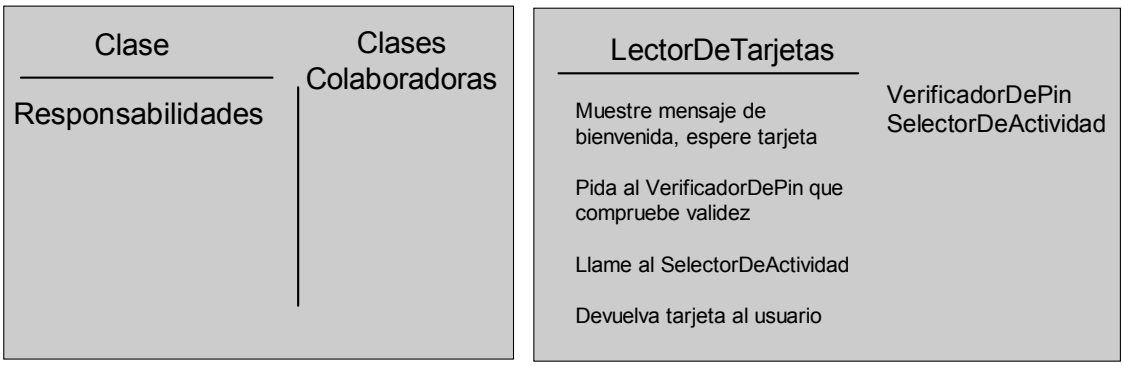


fig. 4.12 Partes de una tarjeta CRC y ejemplo de su uso

4.4.2.3 Enfoque guiado por eventos

Las operaciones son procesos que son llevados a cabo por los objetos. Cuando las operaciones tienen éxito, ocurren los eventos. Los eventos implican cambios de estado en los objetos involucrados.

Utilizar los eventos como punto de arranque para el análisis es un enfoque que ha sido extendido por Martin y Odell [Martin 1997]. Bajo este enfoque se listan los objetivos del sistema y se definen formalmente los eventos con los cuales se consiguen esos objetivos. Mediante un análisis de esos eventos y sus causas se identifican los tipos de objetos involucrados, con los cuales se crean y refinan los diagramas correspondientes. En la fig. 4.13 (tomada de [Martin 1997]) se presenta una secuencia de eventos, donde se puede inferir que para que se pueda preparar y consumir un tazón de cereal, debería de existir un tipo de objeto tazón de cereal o un tipo más general. Martin y Odell escriben: "solamente los elementos estructurales identificados en el análisis de los eventos deberían quedar definidos para el dominio del problema. Cualquier elemento que no pueda ser identificado, no deberá ser definido."

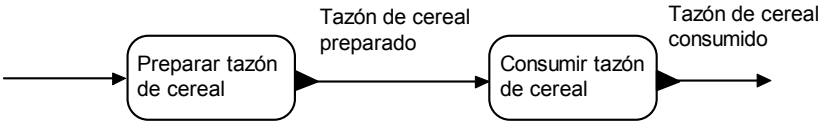


fig. 4.13 Secuencia de eventos en la notación de Martin y Odell

El análisis de eventos es útil para elaborar diagramas de eventos e identificar elementos estructurales para su mapeo en la vista estática del sistema. Este análisis se puede enfocar de dos maneras: dirigido a metas y dirigido a eventos. El primer caso es muy útil para analizar el comportamiento que no está muy bien comprendido. Al definir los objetivos de un proceso, deberá decidirse una forma de llegar a estos objetivos, lo que a su vez se convertirá en nuevos objetivos. Para las situaciones activadas por la capacidad de respuesta del negocio o en las que el comportamiento está razonablemente bien entendido, resulta útil un análisis dirigido a eventos. A continuación se describen brevemente los pasos para los dos tipos de análisis.

Análisis dirigido por metas

- Definir el alcance del análisis
  El analista deberá empezar con el examen de qué es lo que debe y no modelar.

- Aclare el tipo de evento
  Especificar pre-estados y post-estados de los eventos.

- Generalice el tipo de evento
  Identificar si el evento cabe en un tipo de evento más general.

- Defina condiciones de operación
  Considerar las operaciones así como sus condiciones de activación.

- Identifique las causas de la operación
  Se identifican las reglas de activación para el diagrama de eventos.

- Depure los resultados del ciclo
  Se generaliza, especializa e incluso se puede eliminar tipos de objetos y sus mapeos.

En la fig. 4.14 se presenta en forma esquemática los pasos necesarios para llevar a cabo el análisis de eventos dirigido por metas. Se observan los pasos donde se identifican tipos de objetos y las relaciones que se establecen entre ellos. Esto permite ir construyendo el diagrama que representa la vista estática del sistema.

Análisis dirigido por eventos

- Definir el enfoque de análisis
  Idéntico al anterior, pero se hace con el evento de inicio en vez del evento objetivo.

- Aclarar el tipo de evento
  Igual que el dirigido a metas.

- Generalizar el tipo de eventos
  Igual.

- Defina operaciones activadas
  Similar, pero en el dirigido por metas la operación que se estaba definiendo era la que causó el evento. Aquí se definen operaciones, identificándose cada una sobre la base de ser activadas por dicho evento.

- Identifique efectos operacionales
  En el dirigido por metas se identifican los eventos de activación, aquí lo identificado es el tipo de evento resultante de la operación.

- Depure los resultados del ciclo

Igual, pero la elección para continuar el ciclo depende de si el evento objetivo ha sido o no alcanzado.

Este enfoque permite a los analistas considerar el comportamiento general del sistema sin tenerlo que dividir en escenarios separadamente. Sin embargo, esto requiere el uso de una técnica de análisis del comportamiento basado en eventos [Fowler 1995].

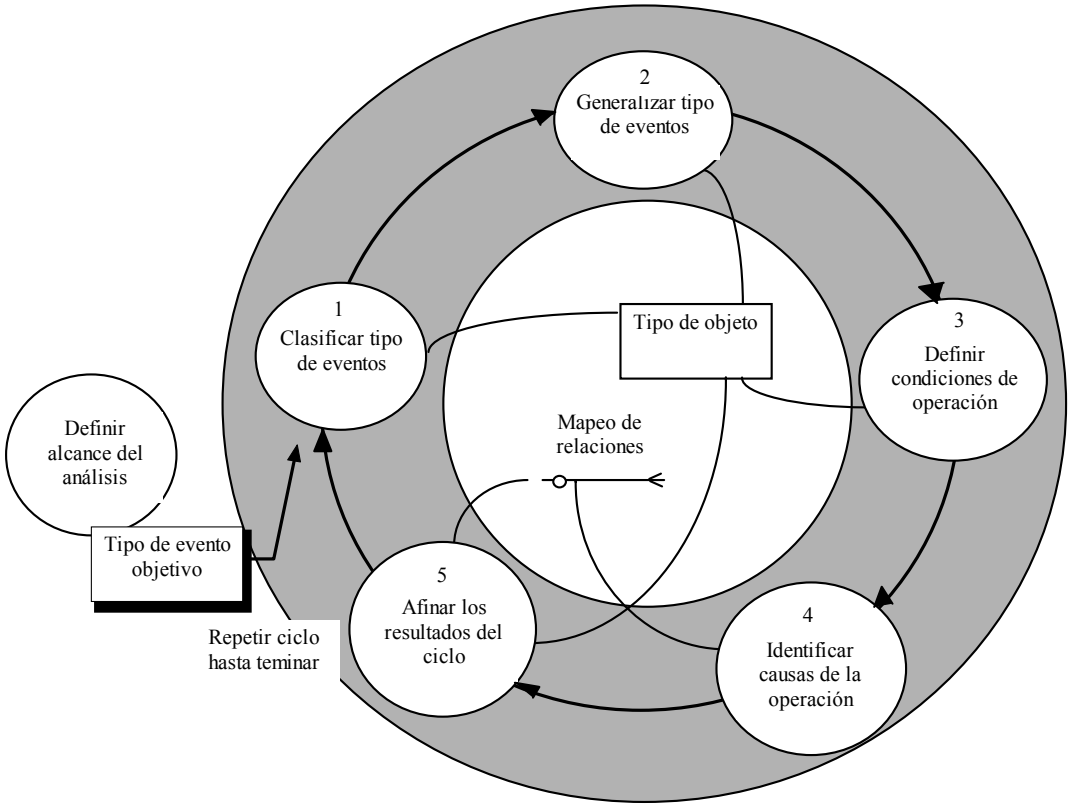


fig.4.14 Gráfico de pasos para análisis de eventos dirigido por metas

Un enfoque adicional que vale la pena comentar, citado por Fowler, es el guiado por patrones, promovido por Peter Coad. En este enfoque se comienza identificando ciertas variedades de tipos (contenedores, transacciones, dispositivos). Se continúa el proceso utilizando un conjunto de patrones que ayudan a construir el modelo donde el patrón actúa como una pregunta dirigida - ¿se aplica este patrón en esta situación?, si es así, ¿cómo? -. Cada patrón también viene con responsabilidades típicas, las cuales al ser usadas como preguntas dirigidas, colaboran en la construcción del modelo.

4.4.3 Vistazo a algunos métodos orientados a objetos

Para poder construir las múltiples vistas que describen un sistema desde su planteamiento hasta su implementación, se dispone de una variedad de métodos que proveen los elementos necesarios

(conceptos, técnicas, pasos, notaciones). Entre los principales métodos orientados a objetos tenemos: Booch, Coad-Yourdon, Fusion, Jacobson, Martin-Odell, OMT y Shlaer-Mellor.

Además de los métodos citados, ha tomado un lugar importante entre los desarrolladores una notación (Unified Modeling Language, UML) para el modelaje de sistemas orientados a objetos a tal punto que actualmente se ha convertido en un estándar[8]. Aunque el UML no forma parte de un método específico, provee elementos de modelaje necesarios para representar apropiadamente las vistas mencionadas en el capítulo anterior. Un resumen de esta notación se presenta en un informe separado.

A pesar de la gran variedad de métodos, existen semejanzas entre ellos, sobre todo en el modelaje del aspecto estático que como se mencionó anteriormente se inspiran en el modelo Entidad - Relación. El autor Devis [Devis 1997] encuentra tanta similitud entre los diversos métodos que al respecto expresa: "los métodos orientados a objetos son como los políticos, conocido uno, conocidos todos". Sin embargo existen diferencias respecto de cuál aspecto se le da mayor énfasis para realizar el modelaje.

Fowler [Fowler 1995] propone algunas recomendaciones para escoger, dependiendo del área en que desee hacer énfasis: estructural, comportamiento, arquitectónico o de heurísticas.

En el aspecto estructural hay poco que escoger ya que la mayoría poseen semejanzas. Sin embargo el método de Odell [Martin 1994] y el de Rumbaugh (OMT) [Rumbaugh 1991] son los más sofisticados. Ambos emplean clasificación múltiple y dinámica. Mientras agregan alguna complejidad en la implementación, ayudan a simplificar la etapa de análisis. Si las reglas son valiosas observe el de Odell y el de Graham.

En cuanto al comportamiento, es conveniente escoger entre aquellos métodos que resaltan el modelaje basado en eventos, estados o interacciones. Cabe notar que si las técnicas basadas en estado deben ser utilizadas, es conveniente hacerlo en conjunción con las técnicas basadas en interacción. Los diagramas de estado de Harel (empleados por Booch y Rumbaugh) resultan ser los más sobresalientes, siendo los diagramas de interacción (Jacobson y Booch) una muy buena fuente para modelar la interacción. Por otro lado, el método de Odell es la única fuente que da suficiente consideración al modelaje de eventos.

La descomposición funcional apoya el aspecto arquitectónico. Sin embargo en la actualidad esta se considera incompatible con el desarrollo orientado a objetos y no debería ser utilizada. Algo semejante es el diagrama de flujo de objetos de Odell, que puede ser útil para el modelaje estratégico y aproximar los flujos de trabajo. Booch provee el mejor compromiso entre simplicidad y dominio, pero se pueden considerar también los contratos de Wirfs-Brock si fuera necesario. Para modelar la arquitectura externa, los casos de uso de Jacobson proporcionan los mejores resultados.

[8] El OMG (Object Management Group) estandarizó la notación UML a fines de 1997

4.4.3.1 Características importantes

A continuación se presentan sinópticamente algunos de los principales métodos de análisis y diseño vigentes a fines del siglo veinte.

Shlaer-Mellor

En este método las clases modelan la vista estructural, la cual provee asociaciones y subtipos. Cada clase tiene asociado un diagrama de estados donde se describe su ciclo de vida. Las acciones definidas en los diagramas de estado se describen en seudocódigo o en un diagrama de flujo de datos. Las conexiones de mensajes entre clases se dividen en eventos asincrónicos y consultas sincrónicas, y son definidas como paso inicial en el modelaje de la vista arquitectónica. Estos modelos, basados en mensajes, son usados de nuevo para describir subsistemas.

Shlaer-Mellor	
Etapas	Descripción
1. Desarrollar un modelo de información	El modelo consiste de un conjunto de objetos, atributos, asociaciones y construcción de objetos por medio de relaciones es-un.
2. Definir los ciclos de vida de los objetos	Analizar el ciclo de vida de cada objeto y formalizarlo mediante un conjunto de estados, eventos, reglas de transición y acciones. Definir temporizadores (timers), para generar eventos futuros. **Se utiliza el modelo de estado.**
3. Definir las dinámicas de las asociaciones	Desarrollar un modelo de estado para las asociaciones entre objetos que evolucionan en el tiempo. Para cada asociación dinámica se define, en el modelo de información, un objeto asociativo.
4. Definir las dinámicas del sistema	Producir un modelo de tiempo y control para el sistema.
5.Desarrollar los modelos de procesos	Crear un diagrama de flujo de datos para cada acción. El diagrama muestra los procesos de cada acción, y los flujos de los datos entre los procesos.
6. Definir dominios y subsistemas	Descomponer el problema en dominios conceptualmente distintos. Se identifican cuatro tipos de dominios: de aplicación, de servicio, de arquitectura y de implementación.

Modelos	Descripción
1. Modelo de información	Permite identificar las entidades conceptuales del mundo real. Las entidades y asociaciones se representan con ayuda de un modelo estilo entidad-asociación.
2. Modelo de estado	Representa el comportamiento de cada entidad y enlace.
3. Modelo de procesos	Define las acciones asociadas a cada estado del modelo de estado mediante un diagrama de flujo de datos tradicional.

Coad-Yourdon

En este método la atención se centra en la fase de análisis y no hay una transición clara de la fase de análisis a la de diseño. Cinco etapas de desarrollo componen el método: objetos y clases, estructuras, temas, atributos y servicios. Cuatro componentes se toman en cuenta en cada etapa: Interacción con el usuario, dominio del problema, manejo de tareas, manejo de datos.

Coad-Yourdon	
Etapas	Descripción
1. Definir objetos y clases	La definición de las clases de objetos puede hacerse buscando en: estructuras, otros sistemas, periféricos, eventos (sucesos), cosas por registrar, papeles desempeñados, procedimientos operacionales, unidades organizacionales.
2. Definir estructuras	Se identifican dos tipos de estructuras: herencias y composiciones. Se definen buscando las asociaciones entre clases y se representan como: Generalización/Especialización y Compuesto/Componente (agregación/ descomposición).
3. Definir áreas temáticas	Dividir esquema de la aplicación en temas (áreas temáticas), para obtener subesquemas más simples y manejables. Las clases de objetos raíz de una composición son áreas potenciales. Refinar las áreas temáticas para reducir la interdependencia entre temas.
4. Definir atributos	Identificar los atributos que caracterizan cada clase de objetos. Las características o propiedades atómicas de los objetos son sus posibles atributos. Identificar las asociaciones entre clases de objetos. Determinar la cardinalidad de las asociaciones.
5. Definir servicios	Determinar operaciones que definen el comportamiento de los objetos. Establecer mensajes entre objetos. Identificar todos los servicios que provee cada clase ya sea para sí o para otras clases.
Modelos	Descripción
Modelo de objetos	Contempla principalmente las jerarquías Generalización/Especialización y Compuesto/Componente. Esta basado en entidad - asociación (la notación para cardinalidades difiere de todos los demás métodos).

Booch

La filosofía del método es orientada al diseño, y provee construcciones ligadas al C++. La notación refleja esta característica. Booch divide su notación en: conceptos esenciales y avanzados. Los diagramas de clases representan la vista estructural y algunos aspectos de arquitectura con la noción de categorías de clases. Los diagramas de objetos presentan instancias y los mensajes que se intercambian. El comportamiento se describe mediante los diagramas de estado de Harel en conjunción con técnicas basadas en interacción. Se lleva a cabo una clara separación entre arquitectura lógica (usando categorías de clases) y arquitectura física (usando subsistemas).

Booch	
Etapas	Descripción
1. Identificar clases y objetos	Identificar las entidades fundamentales del sistema: clases y objetos relevantes. Establecer mecanismos que definan el comportamiento requerido de los objetos para realizar cierta función.
2. Identificar la semántica de clases y objetos	Establecer los significados de las clases y los objetos identificados en la etapa anterior. Técnica útil: escribir escenarios que describan el ciclo de vida de cada objeto desde su creación hasta su destrucción, enfatizando sus comportamientos característicos.
3. Identificar asociaciones entre clases y objetos	Establecer interacciones de clases y objetos: herencia entre clases, patrones de colaboración entre objetos. Descubrir modelos que permitan reorganizar y simplificar las estructuras de las clases y conjuntos de objetos que colaboran. Tomar decisiones de visibilidad/ocultamiento entre clases y objetos.
4. Implementar clases y objetos	Tomar decisiones referentes a la representación de clases y objetos. Asignar clases y objetos a módulos. Asignar programas a procesadores.
Modelos	Descripción
1. Modelo lógico	Se contemplan los aspectos estático, dinámico y de arquitectura desde un punto de vista conceptual. Para ello se utilizan los diagramas de objetos, diagrama de categorías de clases, diagramas de transición de estados y de interacción.
2. Modelo físico	También se contemplan los aspectos estático, dinámico y de arquitectura desde la perspectiva de implementación.. Se dispone de diagramas de módulos y diagramas de procesos.

Object Modeling Technique (OMT, "Rumbaugh")

El método OMT, como se publicó originalmente, usa tres técnicas: diagramas de objetos para la vista estructural, los diagramas de estados de Harel para el comportamiento y diagramas de flujos de datos para una vista funcional. La vista estructural posee una notación muy rica y compleja. Rumbaugh fue uno de los primeros en usar los diagramas de estado de Harel. Sin embargo, las técnicas basadas en estado necesitan algo más para manejar el comportamiento entre objetos y aunque se provee una técnica basada en interacción, prácticamente quedan en un segundo plano.

Sin embargo se han realizado modificaciones al método enfocando esos aspectos, las cuales han aparecido en múltiples artículos[9].

OMT	
Etapas	Descripción
1. Análisis de objetos	Escribir un enunciado del problema. Construir un modelo de objetos. Desarrollar un modelo dinámico. Construir un modelo funcional. Verificar, iterar, simplificar y refinar los tres modelos.
2. Diseño del sistema	Primera etapa del diseño. Se toman decisiones de alto nivel acerca de la estructura general del sistema y su arquitectura.
3. Diseño de objetos	Orientado hacia modelos computacionales requeridos para la implementación práctica. Construye modelos de objetos, dinámico y funcional refinados.
4. Implementación	El diseño se expresa en forma ejecutable: lenguajes de programación, bases de datos, redes de comunicación, hardware, etc.
Modelos	Descripción
1. Modelo de objetos	Descripción de la estructura de los objetos del sistema. Usa diagramas estilo entidad - asociación con una notación muy rica.
2. Modelo dinámico	Escenarios, statecharts de Harel, diagrama de flujo de eventos. Descripción de los aspectos de un sistema concernientes al control, incluyendo tiempo, secuenciación de operaciones e interacción de objetos.
3. Modelo funcional	Descripción de los aspectos del sistema que transforman valores mediante funciones, restricciones, correspondencias y dependencias funcionales. Antes se usaban diagramas de flujo de datos, en OMT-2 se usan diagramas de interacción de objetos.

Object-oriented Software Engineering (OOSE, "Jacobson", Objectory)

OOSE que se sustenta en el método Objectory (desarrollado por Object Systems en Suecia bajo la dirección de Ivar Jacobson). OOSE consiste en un proceso que se desarrolla a través de múltiples etapas, cada una de las cuales entrega un resultado. Uno de los aspectos importantes del método es su enfoque en los escenarios (denominados *casos de uso*) los cuales desempeñan un papel esencial en el proceso de desarrollo. Otra característica distintiva en este método es la clasificación de los objetos del análisis en objetos de interfaz, control y entidad. Esto promueve un estilo para separar y a la vez integrar el procesamiento y la interfaz del usuario en el diseño del sistema.

[9] Rumbaugh, J. "OMT: The dynamic model," Journal of Oriented Programming,7,9 (1995), pp 6-12
Rumbaugh, J. "OMT: The functional model," Journal of Oriented Programming,8,1 (1995), pp 10-14
Rumbaugh, J. "OMT: The object model," Journal of Oriented Programming,7,8 (1995), pp 21-27

OOSE, Jacobson	
Etapas	Descripción
1. Análisis	Analizar y especificar requerimientos, sin tomar en cuenta el ambiente de implementación. Modelar los objetos del dominio del problema. Utiliza casos de uso: secuencia de acciones ejecutadas por un actor en un diálogo con el sistema para proveer algún valor tangible para el actor.
2. Construcción	Consiste de dos fases: diseño e implementación. Se construye un modelo en cada una de ellas. Para cada caso de uso concreto se dibuja un diagrama de interacción que muestra cómo funcionan los objetos, lo que describe la comunicación entre los diferentes bloques.
3. Prueba	Realizar pruebas de integración del sistema: comunicación entre bloques y las interfaces descrita en los requerimientos. Documentar evaluación. Los casos de uso son importantes en esta fase.
Modelos	Descripción
1. Modelo de requerimientos	Permite especificar la funcionalidad del sistema. Se utilizan los casos de uso y descripción de interfaces: vistas lógicas del usuario del sistema (sin detalles; información esencial y estilo de interacción).
2. Modelo de análisis	Describir el sistema utilizando tres variedades de objetos: objetos de interfaz, de entidad y de control.
3. Modelo de diseño	Permite refinar el modelo de análisis y está compuesto por bloques que constituyen el diseño de los objetos.
4. Modelo de implementación	Permite realizar la implementación del sistema en código fuente.
5. Modelo de evaluación.	Presenta la especificación de las pruebas que comprobarán la idoneidad del sistema y sus casos de uso.

A continuación presentamos una tabla que resume el énfasis que dan los principales métodos de desarrollo orientados a objetos a las diversas fases de modelaje. (++: mayor énfasis, +: razonable énfasis, -: menor énfasis, --: muy poco énfasis, en blanco: no cubierto)

Método	Requerimientos	Análisis	Diseño	Implementación
Booch		-	++	++
Coad-Yourdon		++	+	+
Fusion	+	++	++	++
Jacobson	++	++	++	+
Martin-Odell	++	++	+	-
OMT		++	+	+
Shlaer-Mellor		++	-	--

Tabla 4.2 Fase del modelaje que se enfatiza

La siguiente tabla resume el apoyo que tienen los métodos dominantes de la década de 1990.

Método	Cursos	Libros	Herramientas	Lenguaje
Booch	√	√	√	Independiente
Coad-Yourdon	√	√	√	Independiente
Fusion	√	√	√	Independiente
Jacobson	√	√	√	Independiente
Martin-Odell	√	√	√	Independiente
OMT	√	√	√	Independiente
Shlaer-Mellor	√	√	√	Independiente

Tabla 4.3 Apoyos a los métodos

En esta tabla se presentan las áreas de desarrollo donde los diversos métodos muestran fortaleza o debilidad.

Método	Sistemas de información	Software de base	Sistemas de control
Booch	-	++	++
Coad-Yourdon	++	+	-
Fusion	-	++	++
Jacobson	++	++	++
Martin-Odell	++	+	-
OMT	+	++	++
Shlaer-Mellor	+	+	+

Tabla 4.4 Principales dominios de aplicación de los métodos

5 Perspectivas

5.1 Estandarización

Entre 1987 y 1995 ha surgido una enorme variedad de métodos para hacer análisis y diseño de sistemas. Incluso se habló de una "guerra de los métodos" ("*method wars*"), entre los principales proponentes de métodos con alguna etiqueta de "objetos". El resultado de esto fue una notable confusión entre quienes deseaban desarrollar software bajo el paradigma de objetos, pues no contaban con conceptos ni notaciones estandarizadas para expresar sus modelos de análisis o diseño, ni con herramientas que facilitasen la documentación o la generación (parcial o total) de implementaciones.

Con la llegada de Rumbaugh y Jacobson (entre 1995 y 1996) a la compañía Rational, donde ya estaba Booch, se comenzó un esfuerzo por "unificar" los conceptos, notaciones y métodos de estos tres autores. Por iniciativa del Object Management Group (OMG)[10], en 1996 se comenzó un esfuerzo por lograr un estándar para notaciones de modelaje que permitiesen el apoyo por herramientas de muchos proveedores.

El proceso seguido por el OMG se basa en tecnología pre-existente. Primero se convoca a presentar información sobre el estado del arte de posibles estándares de base (RFI: *request for information*) y luego se convoca a la presentación de propuestas (RFP: *request for proposal*). Muchas organizaciones enviaron información sobre su tecnología (notación, conceptos, herramientas, métodos), entre ellas: Rational, IBM, Oracle, ICL, Hewlett-Packard, Softeam, EIA CDIF[11] Division, Finsiel, Boeing, Ericsson. Hubo varias propuestas base para la estandarización, convocada por OMG para inicios de 1997; entre las principales estuvieron: Rational (UML), consorcio OPEN (OML, proceso OPEN), IBM y ObjectTime. La propuesta de notación de Rational (UML) contó con el respaldo de muchas empresas informáticas, notablemente: Oracle, Microsoft, Digital, Texas Instruments, Hewlett-Packard e i-Logix. El UML fue votado como el estándar de base y luego fue modificado para acoger algunas otras propuestas. Se han publicado libros de referencia y guía sobre UML [Rumbaugh 1999, Booch 1999].

Además del UML, en este esfuerzo de estandarización se utilizó el Object Constraint Language (OCL)[12] para restringir aspectos de la definición sintáctica del UML. OCL se puede utilizar como una notación complementaria al UML, para especificar restricciones que no pueden ser expresadas en notación gráfica (precondiciones y postcondiciones de operaciones, relaciones de integridad entre elementos estructurales, reglas de negocio expresables como condiciones, etc.).

[10] El OMG es la organización que promueve estándares en tecnología de objetos. Entre los más notables están OMA (Object Management Architecture), CORBA (Common Object Request Broker Architecture) y UML (Unified Modeling Language).

[11] CDIF: CASE Data Interchange Format, un formato estandarizado para el intercambio de datos entre herramientas CASE. El intercambio se hace mediante una dimensión "espacial" y una "lógica". La dimensión espacial muestra la colocación de los elementos de notación en un plano bidimensional que puede ser grafidado. La dimensión lógica muestra la interrelación entre los elementos de diseño; ésta es la que potencialmente permite cosas como generación de código, control de calidad, métricas, manejo de configuración, etc.

[12] Lenguaje para restricción de objetos.

El OMG adoptó al UML y al OCL como lenguajes estandarizados en noviembre de 1997. El UML cuenta con un metamodelo semejante al de CDIF, por lo que se espera que se convierta en la notación de uso común para representar visualmente modelos de objetos. Muchos proveedores de herramientas han adoptado ya la notación y muchos más lo harán. Esto facilitará el trabajo de los desarrolladores de sistemas[13] y se espera que en el futuro será posible el intercambio de modelos entre herramientas de distintos proveedores.

En cuanto a los métodos, no se ha estandarizado ninguno aún. Es improbable que esto suceda, por cuanto es inverosímil que llegue a inventarse un método general para resolver cualquier clase de problema. El consorcio OPEN ha documentado un proceso de desarrollo, así como una variedad de técnicas útiles en distintas actividades del desarrollo de sistemas. Rational Software y los autores principales del UML (Booch, Jacobson y Rumbaugh) han propuesto un proceso unificado para el desarrollo de sistemas [Jacobson 1999, Kruchten 1999] que se prevé tendrá fuerte impacto en el ambiente profesional y empresarial.

5.2 Elección de método y proceso[14]

Una de las panaceas informáticas más veneradas en los últimos años es la de las "metodologías" o, más apropiadamente, los métodos. Hay quienes piensan que es posible, siguiendo un método particular, resolver cualquier problema de desarrollo de sistemas que se les vaya a presentar. No hay tal. La búsqueda de métodos universales está condenada al fracaso, a pesar de lo que digan los vendedores y su gurú favorito.

Los métodos para el desarrollo de sistemas han surgido como un embellecimiento y abstracción de los métodos de programación. Este ha sido, históricamente, un proceso inductivo. Los métodos de programación surgieron a partir de técnicas inventadas para lenguajes de programación particulares, que fueron generalizables a familias de lenguajes de programación que seguían (más o menos) el mismo paradigma (imperativo-procedimental, orientado a objetos, funcional, etc.). El ciclo ha sido, aproximadamente éste:

1. Aparece un lenguaje de programación que tiene cierto éxito.
2. Se proponen técnicas para organizar programas en ese lenguaje.
3. Las técnicas (de programación) se generalizan a lenguajes semejantes.
4. Aparecen técnicas y notación para representar más abstractamente los *diseños*, siguiendo el mismo paradigma.
5. Aparecen técnicas para pasar de los diseños a estructuras de programación.
6. Aparecen técnicas y notación para representar más abstractamente los modelos de *análisis*, siguiendo el mismo paradigma.
7. Aparecen técnicas para pasar los modelos de análisis a modelos de diseño.
8. Aparecen consultores y desarrolladores expertos en el paradigma.
9. Aparecen libros de texto (para todos los niveles: programación, diseño, análisis).

[13] Ejemplos de herramientas que apoyan parcial o totalmente al UML, disponibles en el medio centroamericano son. Rose de Rational Visual Modeler (incluido en Visual Studio) de Microsoft y Object Engineering Workbench de Innovative Software.

[14] El contenido de esta sección se basa en [Trejos 1998].

10. Aparecen herramientas para apoyar las diversas notaciones del paradigma. Estas pasan por varias generaciones y van desde diagramadores con poca semántica hasta generadores de aplicaciones que apoyan un ciclo de vida para el desarrollo de sistemas.

Un método plantea maneras explícitas de estructurar las acciones y pensamientos del desarrollador. Un buen método indica al desarrollador *qué* hacer, *cómo* hacerlo, *cuándo* hacerlo y *porqué* se hace. Un método de análisis y diseño es un enfoque coherente para la descripción de sistemas.

No deben confundirse lenguajes (como UML) con métodos. Los métodos son mucho más que notación, pues usualmente también comprenden elementos como técnicas, heurísticas, proceso y patrones.

La decisión de adoptar un método no es simple. He aquí algunas recomendaciones:

- Estudie y compare varios métodos. Estos son usualmente documentados en libros; busque los que contengan ejemplos ilustrativos de tamaño no trivial.
- Evalúe la aplicabilidad de los métodos al tipo de sistema que desarrolla su organización. Escoja un método de base.
- Inicie un proyecto piloto que sea: de tamaño razonable (no muy grande), útil para su organización, interesante técnicamente, representativo de aplicaciones en su organización y no crítico en cuanto a tiempo. El riesgo debe ser riesgo reducido y solo atribuible a aspectos metodológicos.
- Forme un equipo de proyecto con personal entusiasta, receptivo y competente.
- Capacite a su personal. Promueva que estudien otros métodos alternativos.
- Considere la combinación de técnicas de diversos métodos. Utilice sólo una notación, como el UML, que seguramente tendrá correspondencia con las notaciones de los métodos. Puede considerar utilizar una herramienta CASE liviana para diagramar.
- Procure aprovechar conocimientos existentes en la organización y las similitudes que estos tengan con los métodos considerados.
- Incentive que su personal aplique principios de diseño que busquen calidad, reutilización y mantenibilidad.
- Considere buscar consultoría o mentoría externas.
- Dé seguimiento al proyecto y evalúelo.
- Aprenda de la experiencia.
- Considere ahora dar pautas de desarrollo con el método, evite ser burocrático. Ahora es el momento para considerar herramientas CASE más potentes. No introduzca herramientas CASE sin antes contar con una base metodológica.
- Defina estrategias y principios por utilizar en futuros proyectos.

Tenga presente que, independientemente del paradigma de su método favorito, la abstracción y la descomposición (estructuración) son las dos principales herramientas intelectuales con que contamos para manejar la complejidad inherente a cualquier proyecto informático no trivial.

Nuestra recomendación es que los interesados estudien el *proceso* propuesto por Rational [Kruchten 1999, Jacobson 1999], pero lo complementen con técnicas para análisis y diseño

desarrollados previamente por otros metodólogos (ver *5.3 Recomendaciones bibliográficas*). El proceso unificado de Rational está bien definido y los materiales de estudio son razonablemente asequibles, esto puede servir de base para que una organización defina su propio proceso de desarrollo.

5.3 Recomendaciones bibliográficas

El método más influyente en lo conceptual, y quizás el más sólido, es el OOSE de Jacobson [Jacobson 1994], que es muy útil para empezar el desarrollo de sistemas a partir de una formulación de los casos de uso. El método de Jacobson puede ser complementado por otros. Para comenzar a plantear modelos de objetos en el análisis un enfoque sencillo es el de Coad y Yourdon [Coad 1991], que tiene muchas heurísticas útiles para el descubrimiento de objetos. [Booch 1996] contiene gran diversidad de técnicas útiles en el diseño. [Rumbaugh 1991] presenta también muchas técnicas útiles para análisis, diseño e implementación; de particular utilidad son las guías para llevar modelos estructurales hacia bases de datos relacionales.

[Coad 1997] presenta un conjunto de estrategias y heurísticas para el desarrollo de modelos de objetos, que complementa otros planteamientos. En [Martin 1997] se presentan técnicas y procesos útiles para análisis y diseño; las técnicas para trabajo en grupo allí expuestas pueden trabajarse con cualquier otro método de análisis. [Coleman 1994] documenta el método Fusion, que contiene buenas ideas para el análisis y el diseño, y sigue un proceso bastante riguroso.

Para madurar mejor la aplicación rigurosa de la orientación a objetos, vale la pena estudiar y comprender en profundidad sus *principios*. [Meyer 1996] presenta de manera magistral y lúcida los principios del desarrollo integral de software de acuerdo con el paradigma de objetos; en particular, estúdiese con atención su capítulo de “diseño por contrato”. La idea de desarrollar rigurosamente sistemas de software siguiendo la noción de contratos es expuesta ampliamente en [Waldén 1995]. Cook y Daniels han escrito una de las mejores referencias en cuanto a modelaje riguroso orientado a objetos [Cook 1994].

Desde 1993 ha surgido con fuerza la idea de utilizar patrones para resolver problemas particulares de diseño. El libro clásico sobre patrones de diseño es [Gamma 1994]. [Fowler 1997] presenta una diversidad de patrones útiles para el análisis de sistemas.

El libro más recomendado para administración de proyectos con tecnología de objetos es [Goldberg 1995]. [Webster 1997] presenta recomendaciones para reconocer, resolver y evitar problemas relativos al desarrollo de software con el paradigma de objetos.

En los proyectos de desarrollo de sistemas es conveniente considerar el entorno de procesos en que se usarán los sistemas y hacer una reingeniería de estos cuando sea conveniente. Una interesante aplicación de los principios de la orientación a objetos a la ingeniería organizacional (reingeniería de negocios) puede encontrarse en [Jacobson 1994a].

Anexo. Objetos y clases

Entre los más importantes conceptos de la tecnología orientada a objetos se encuentran la creación y descripción de las clases y los objetos, bloques básicos de construcción. Estos determinan cuán reutilizables pueden resultar los componentes de software cuando las entidades creadas durante la fase de análisis son transformadas por la fase de diseño en entidades del programa.

1 Objetos

Desde la perspectiva de la cognición humana, un *objeto* es cualquiera de las siguientes cosas [Booch 1996]:

- Una cosa tangible y/o visible
- Algo que puede comprenderse intelectualmente
- Algo hacia lo que se dirige un pensamiento o acción

Se añade la idea de que un objeto modela alguna parte de la realidad y es, por tanto, algo que existe en el tiempo y el espacio. En software, el término objeto se aplicó formalmente en primer lugar en el lenguaje Símula; los objetos existían en los programas en Símula típicamente para simular algún aspecto de la realidad.

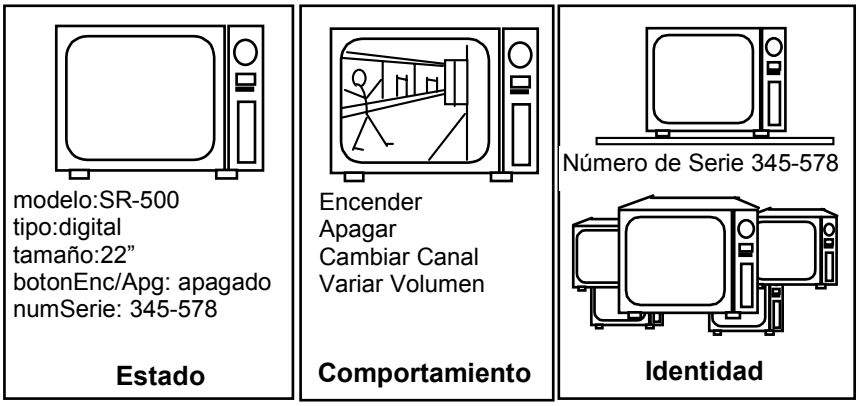


fig. 1 Características de un objeto

Los objetos del mundo real no son las únicas variedades de objetos de interés en el desarrollo del software. Otros tipos importantes de objetos son aquellos que resultan como invenciones del proceso de diseño y cuyas colaboraciones con otros objetos sirven como mecanismos para desempeñar algún comportamiento de nivel superior. Booch (citando a Smith y Tockey) escribe: “un objeto representa un elemento, unidad o entidad individual e identificable, ya sea real o abstracta, con un papel bien definido en el dominio del problema”, y añade: “nuestra experiencia sugiere la siguiente definición:”

"Un objeto tiene estado, identidad y comportamiento (fig.1); la estructura y comportamiento de objetos similares están definidos en su clase común; los términos instancia (fig. 2) y objeto son intercambiables."

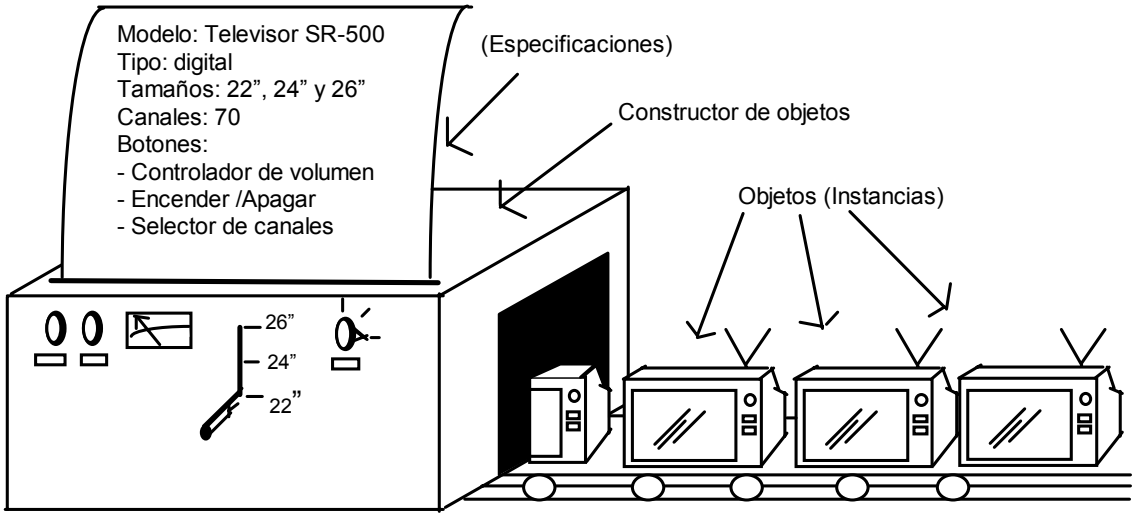


fig. 2 Conceptos de clases (especificaciones) y objetos

Los objetos se pueden reunir en conjuntos, que tengan la misma estructura y comportamiento. Dichos conjuntos se conocen como clases de objetos. Las *clases* describen cómo se estructuran internamente los objetos y cuáles son los servicios que proveen.

Toda la información en un sistema está almacenada dentro de los objetos y puede ser manipulada solamente solicitándole a los objetos ejecutar aquellas operaciones definidas para tal fin. Las operaciones que cada objeto puede llevar a cabo definen su comportamiento, los clientes únicamente pueden solicitar aquellas operaciones que se definen en la interfaz.

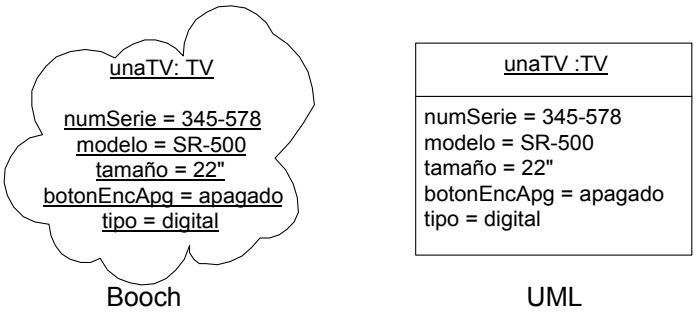


fig.3 Representación de un objeto con diferentes notaciones

1.1 Estado

El *estado* de un objeto es el resultado acumulado de su comportamiento [Booch 1996]. Por ejemplo, cuando se instala un teléfono por primera vez, se encuentra en estado ocioso, lo que significa que no hay ningún comportamiento anterior de interés especial. Cuando levantamos el teléfono, se dice que está descolgado y en estado de marcar; en ese estado no se espera que suene el timbre, se tiene la posibilidad de iniciar una conversación con alguna persona en otro teléfono.

Si el teléfono está colgado y suena el timbre, al levantarlo, se encontrará es estado de recepción, dándonos la posibilidad de hablar con la persona que inició la comunicación.

En cualquier momento concreto, el estado de un objeto abarca todas sus propiedades (habitualmente inmutables), junto con los valores actuales (habitualmente mutables) de cada una esas propiedades. Se dice que son habitualmente mutables porque en algunos casos pueden ser inmutables, tal como el número de serie del televisor, el cual no cambia después de darle su valor (usualmente cuando se crea el objeto).

Un *evento* es un cambio en el estado de un objeto. El evento provoca el envío de un mensaje a un objeto. El *mensaje* es una solicitud de servicio que ocasiona la alteración de algunos de los valores asociados a las propiedades de los objetos. Supongamos que definimos una propiedad para el objeto televisor, llamada botonEncendidoApagado con la posibilidad de almacenar uno de dos valores: "encendido" o "apagado". Pensemos que inicialmente se encuentra en el estado "apagado"; ejecutar la operación "encender" provocará un cambio de estado, el cambio de estado se reflejará en el cambio del valor del atributo botonEncendidoApagado.

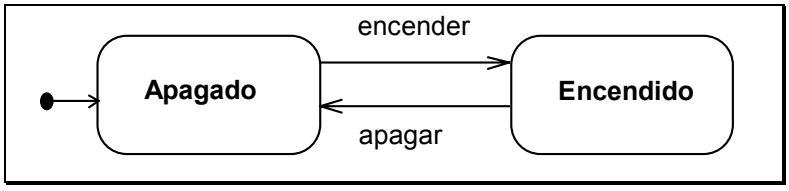


fig. 4 Estados del objeto televisor

Los estados se representan por lo general en los diagramas de transición de estados, donde se muestran los eventos que provocan la transición de un estado a otro, y las acciones que resultan de ese cambio de estado (fig. 4). Cada diagrama de transición de estados representa una vista del comportamiento de una sola clase durante su ciclo de vida.

1.2 Identidad

Cada objeto posee una *identidad* que lo distingue de otros objetos; la identidad se representa computacionalmente vía un identificador interno que denominaremos *oid* y que no es revelado al usuario. Al identificar cada objeto en forma única, se pueden tener dos objetos con características idénticas y considerarlos como objetos diferentes. Los oids permiten que los objetos tengan una existencia independiente de los valores contenidos en sus atributos.

1.3 Comportamiento

Ningún objeto existe de forma aislada. Los objetos reciben solicitudes para realizar acciones, y ellos mismos actúan sobre otros objetos. Puede decirse que el *comportamiento* es cómo actúa un objeto en términos de sus cambios de estado y paso de mensajes. Podemos pensar que el comportamiento de un objeto representa su actividad visible y comprobable exteriormente [Booch 1996].

En el mundo real, las personas y las cosas exhiben comportamientos específicos. Por ejemplo, un carro acelera, desacelera, frena, etc. En el mundo del software, los sistemas y sus componentes también exhiben comportamientos. El comportamiento de un objeto dentro de un sistema es descrito a través de un conjunto de operaciones (métodos). Una operación es una acción que un objeto debe llevar a cabo, esta acción puede provocar una reacción en otro objeto enviándole un mensaje.

Dentro de las operaciones que una clase ofrece como servicio a sus clientes, se ha observado que existen típicamente cinco tipos [Booch 1996]. Los tres tipos más comunes de operaciones son:

Modificador: Una operación que altera el estado de un objeto.

Selector: Una operación que accede al estado de un objeto, pero no lo altera.

.

Iterador: Una operación que permite acceder a todas las partes de un objeto en algún orden establecido.

Hay otros dos tipos de operaciones habituales; representan la infraestructura necesaria para crear y destruir instancias de una clase:

Constructor: Una operación que crea un objeto y/o inicializa su estado.

Destructor: Una operación que libera el estado de un objeto y/o destruye el objeto.

El comportamiento que cada objeto exhibe contribuye al comportamiento del sistema, el cual se manifiesta como un todo por la interacción de los objetos que lo componen. Como participante de esa interacción, un objeto puede desempeñar uno de tres papeles:

Actor: Un objeto que puede operar sobre otros objetos pero nunca se opera sobre él por parte otros objetos; en algunos contextos, los términos objeto activo y actor son equivalentes.

Servidor: Un objeto que nunca opera sobre otros, solo otros operan sobre él.

Agente: Un objeto que puede operar sobre otros y además otros objetos pueden operar sobre él; un agente se crea normalmente para realizar algún trabajo en nombre de un actor u otro agente.

1.4 Colaboración y paso de mensajes

La colaboración puede ser definida como un servicio ofrecido por un objeto para ser usado por otros objetos. El concepto de colaboración se encuentra estrechamente ligado al concepto de paso de mensajes, el cual es una descripción de cómo se comunican los componentes del software [Nielsen 1995]. La llamada a procedimientos en un lenguaje procedimental y el envío de un mensaje en los orientados a objetos tienen semejanzas, sin embargo existen dos diferencias bien claras:

1. Todo mensaje posee un receptor, es decir el mensaje enviado tiene que llegar a alguien especificado explícitamente.
2. La interpretación del mensaje (el método usado para responder el mensaje) depende del receptor y puede variar con diferentes receptores.

2 Clases

Clases y objetos se encuentran íntimamente relacionados (en la mayoría de los lenguajes de programación orientados a objetos no puede hablarse de un objeto sin atención a su clase). Mientras que un objeto individual es una entidad concreta que desempeña algún papel en el sistema global, la clase capta la estructura y comportamiento comunes a todos los objetos relacionados y nunca existe en forma concreta[15] (fig.5).

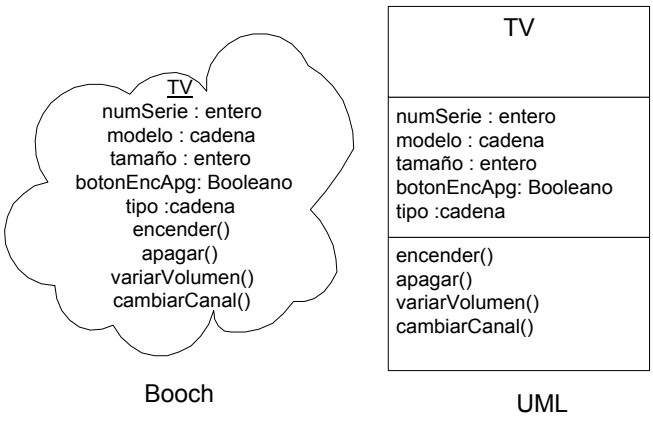


fig. 5 Representación de clases

Un conjunto general de características para la definición de una clase incluye:

1. **Una clase es una abstracción de alto nivel.**
   La abstracción es útil para describir un conjunto de características comunes a entidades encontradas en el análisis y diseño. La clase televisor, por ejemplo, puede ser usada para describir atributos (características) que son comunes a todas las instancias específicas de televisor, tales como modelo, color, tamaño, tipo.
2. **Una clase es una colección de objetos.**
   Cada clase representa una colección de objetos con atributos y operaciones comunes.
3. **Un objeto es una instancia específica de una clase.**
   Esto representa la estrecha relación que existe entre clases y objetos. Un objeto es un ejemplar de una clase particular, y tiene un conjunto de atributos y operaciones asociadas con esa clase.

4. **Una clase puede encontrarse en una jerarquía.**

[15] Esto es parcialmente cierto, pues en lenguajes como Smalltalk existen objetos que representan clases

En un sistema pueden existir gran cantidad de abstracciones diferentes en constante interacción, colaborando unas con otras, a tal grado que resulte difícil comprenderlas simultáneamente. Afortunadamente se puede establecer una clasificación u ordenación de estas abstracciones dando lugar a dos variedades principales de jerarquías: *la jerarquía de clases*, en la que una clase (subclase) hereda de una o más clases (superclases) su estructura y comportamiento, y *la jerarquía de partes* que describe relaciones de agregación.

2.1 Interfaz e implementación

Una clase sirve como una especie de contrato que vincula una abstracción y todos sus clientes [Booch 1996]. La abstracción que desempeña el papel de receptor es el objeto a quien se le envía el mensaje. Si el receptor acepta el mensaje, acepta la responsabilidad de llevar a cabo la acción solicitada.

Esta visión de la programación como un contrato lleva a distinguir entre la visión externa y la visión interna de una clase. *La interfaz* de una clase proporciona su visión externa y por tanto enfatiza la abstracción a la vez que oculta su representación y los secretos de su comportamiento. Esta interfaz se compone de todas las operaciones que representan los servicios que los clientes pueden solicitar a los objetos de esa clase. Por contraste, *la implementación* de una clase es su visión interna, engloba los secretos de su comportamiento, y consiste principalmente de la implementación de las operaciones que establecen sus responsabilidades.

2.2 Relaciones entre clases

Las clases, al igual que los objetos, no existen aisladamente. Antes bien, para un dominio de problema específico, las abstracciones claves suelen estar relacionadas por vías muy diversas e interesantes, formando la estructura de clases del diseño.

Existen tres tipos básicos de relaciones entre clases [Booch 1996]. La primera es la *generalización/especialización*, que denota una relación **"es un"** (is a). Por ejemplo una rosa es un tipo de flor, lo que quiere decir que una rosa es una subclase especializada de una clase más general, la de las flores. La segunda es la relación **"***todo/parte***"** (part of). Así un pétalo no es un tipo de flor, es una parte de una flor. La tercera es la asociación que denota alguna dependencia semántica entre clases de otro modo independientes como, por ejemplo, entre insectos y flores.

2.2.1 Asociación.

Una *asociación* es una conexión entre clases, una conexión semántica entre objetos de las clases involucradas en la asociación. De los tres tipos de relaciones mencionados en el párrafo anterior, las asociaciones son las más débiles semánticamente. Usualmente la identificación de asociaciones entre clases es una actividad de análisis y de diseño inicial, momento en el cual se comienza a descubrir las dependencias generales entre las abstracciones. A medida que se continúa el diseño y la implementación, se refinarán a menudo estas asociaciones débiles orientándolas hacia una de las otras relaciones de clases más concretas [Booch 1996].

En la fig. 6 se muestra una relación de asociación. Mediante la creación de asociaciones, se llega a plasmar quiénes son los participantes en una relación semántica, sus papeles y su *cardinalidad,* es decir el número de participantes en la relación.

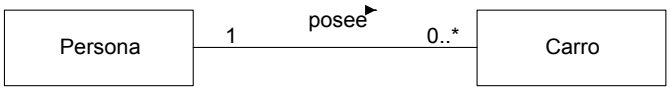


fig.6 Relación de asociación en UML

En la fig.6 se muestra una asociación *uno -a- cero o muchos*, lo que significa que para cada instancia de la clase Persona existen cero o más instancias de la clase Carro y por cada Carro, existe exactamente una Persona. Esta multiplicidad denota la cardinalidad de la asociación. En la práctica, existen tres tipos habituales de cardinalidad en una asociación: uno a uno, uno a muchos y muchos a muchos. Estas cardinalidades pueden restringirse con más precisión en lenguajes como UML.

2.2.2 Generalización/Especialización (Herencia)

Se puede considerar la generalización/especialización como una forma de organizar las clases en una jerarquía de conceptos. El anidar clases para formar entidades especializadas es uno de los aspectos importantes en la descomposición de sistemas en componentes pequeños y manejables.

La generalización es una relación entre una clase general y otra específica. La clase específica, llamada *subclase,* hereda todo lo que ha sido definido e implementado (atributos, operaciones, asociaciones) en la clase general, llamada la *superclase*. Una subclase habitualmente extiende o restringe la información o el comportamiento existente en su superclase. Una subclase que extiende su superclase se dice que utiliza **herencia por extensión.** En contraste, una subclase que restringe el comportamiento de su superclase se dice que usa **herencia por restricción**.

Las superclases representan abstracciones generalizadas, porque en ellas se factorizan las estructuras y comportamiento comunes a clases relacionadas, mientras que las subclases representan especializaciones donde se ubican los elementos que serán añadidos o modificados.

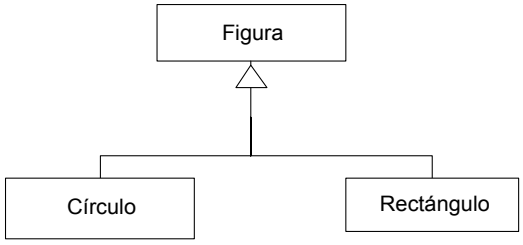


fig.7 Relación de herencia en UML

Cuando una clase Y hereda de una clase X, la clase Y tiene, por herencia, todas las características de X (fig.8). Entonces se dice que Y "es una" X. Y es indudablemente más que una X, pero además de todo lo extra que pudiera ser, es también una X [Korson 1990].

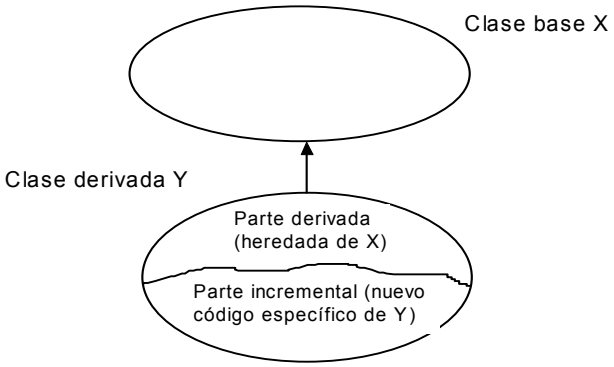


fig. 8 Y es una X

En la jerarquía de clases, algunas clases tendrán instancias y otras no. Se espera tener instancias de las clases más especializadas (*clases concretas*), por ejemplo la clase Rectángulo o Círculo. Sin embargo, no se esperan instancias de clases intermedias, más generales (*clases abstractas*), como la clase Figura. Una clase abstracta tiene operaciones abstractas, es decir aquellas operaciones que no han sido implementadas en la clase donde fueron definidas. Una clase que hereda de una clase abstracta debe implementar las operaciones abstractas o será a su vez otra clase abstracta. Este mecanismo permite que cada clase concreta implemente la misma operación de acuerdo con las responsabilidades particulares de cada una y sea utilizada de formas diferentes (fig.9).

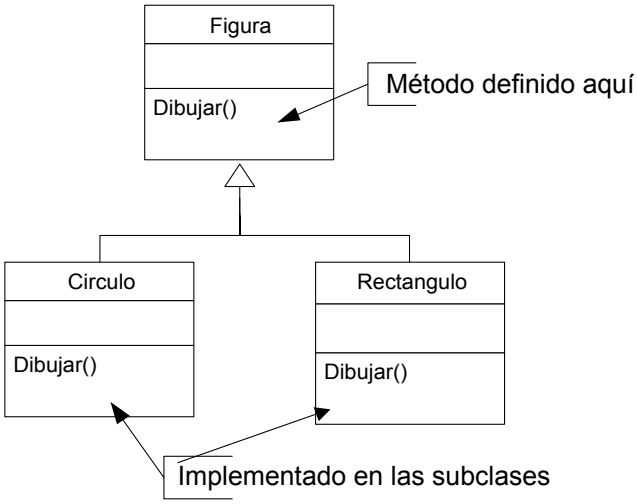


fig.9 Clase abstracta y concretas

Cuando una subclase tiene exactamente una superclase se tiene *herencia simple*. Cuando una subclase se deriva de varias superclases se está en presencia de *herencia múltiple*; aunque no es muy frecuente el uso de la herencia múltiple, existirán casos donde permite crear un mejor diseño. Como menciona Booch: "En nuestra experiencia, se ha encontrado que la herencia múltiple es como un paracaídas: no siempre se necesita, pero cuando así ocurre, uno está verdaderamente feliz de tenerla a mano."

Polimorfismo

El polimorfismo significa que el objeto que envía el mensaje no necesita conocer la clase a que pertenece el objeto receptor. Un mensaje puede ser interpretado de diferentes formas, dependiendo de la clase receptora. Por tanto la instancia que recibe el mensaje es quien determina la interpretación y no la instancia que lo emite. Expresa Jacobson [Jacobson 1994] "A menudo se dice que el polimorfismo significa que una operación puede ser implementada en diferentes formas en diferentes clases; esto solamente es una consecuencia del hecho de que un mismo mensaje pueda ser interpretado de diferente forma dependiendo de la clase a que pertenece el objeto al que se dirige el mensaje y no polimorfismo en sí mismo" y continúa "Polimorfismo y asociación dinámica son a menudo confundidos. Asociación dinámica significa que los estímulos (mensajes) no están asociados a una cierta operación en la clase de la instancia receptora sino hasta que es enviado."

El polimorfismo es una característica importante para lograr la flexibilidad de un sistema ya que al no importarle al emisor (cliente) cómo el receptor (servidor) lleva a cabo una operación, sino solamente que hay alguien que mediante su comportamiento puede satisfacer su solicitud, las modificaciones que se lleven a cabo en el receptor no afectarán al emisor.

La herencia y el polimorfismo son mecanismos de alguna forma ligados. Mientras que la herencia permite que diferentes clases compartan el mismo código, lo que conduce a una reducción del tamaño del código y a un incremento en la funcionalidad, el polimorfismo permite que este código compartido sea adaptado para que se ajuste a las circunstancias específicas de cada clase individual.

2.2.3 Agregación.

La agregación denota una forma fuerte de asociación, en la cual la instancia de una clase (el agregado) está formado por componentes. El agregado es semánticamente un objeto que se trata como una unidad en muchas operaciones, aun cuando conste de varios objetos.

Las diversas clases relacionadas por medio de la descripción de agregación de los objetos asociados establecen una jerarquía todo/parte. Un ejemplo de agregado es un televisor el cual está compuesto de parlante, tubo de imagen, transformador, etc. Los términos claves para identificar agregados son; **"consiste de", "es parte de", "tiene un"**, opuesto a la relación **"es un"** para una estructura jerárquica de herencia.

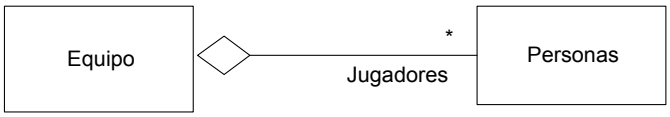


fig. 10 Relación de Agregación

Es importante estar claro de la estrecha relación que existe entre el agregado y sus partes [Webster 1995], ya que se impone una responsabilidad al agregado, de crear y eliminar sus componentes, ya sea llevándola a cabo directamente o delegando en otros objetos dicha responsabilidad.

Existe otra relación que es una forma de agregación, denominada *composición.* Esta es una relación todo/parte mucho más fuerte, donde las partes "viven" en el interior del todo, lo que implica que ambos objetos se encuentran en íntima conexión. Cuando se crea el todo se crean sus partes, de igual forma al destruirse el todo se destruyen sus partes. Esta relación aparece representada en la fig.11.

fig.11 Relación de composición

La agregación y composición representan relaciones todo/parte. Sin embargo la composición, al denotar contención física, no permite que las partes del objeto puedan ser compartidas por otros objetos. Unicamente a través del objeto contenedor se puede acceder a los servicios que prestan las partes. Por el contrario, la agregación al no ser tan restrictiva, permite lo que Booch denomina compartición estructural, donde un objeto puede ser referenciado por otros objetos [Booch 1996], con lo cual se consigue un mejor modelaje de "partes compartidas". En este caso los tiempos de vida de las partes no dependen del tiempo de vida del "todo"; pueden ser partes temporalmente y luego disociarse (sin morir ni perder identidad)(fig.12).

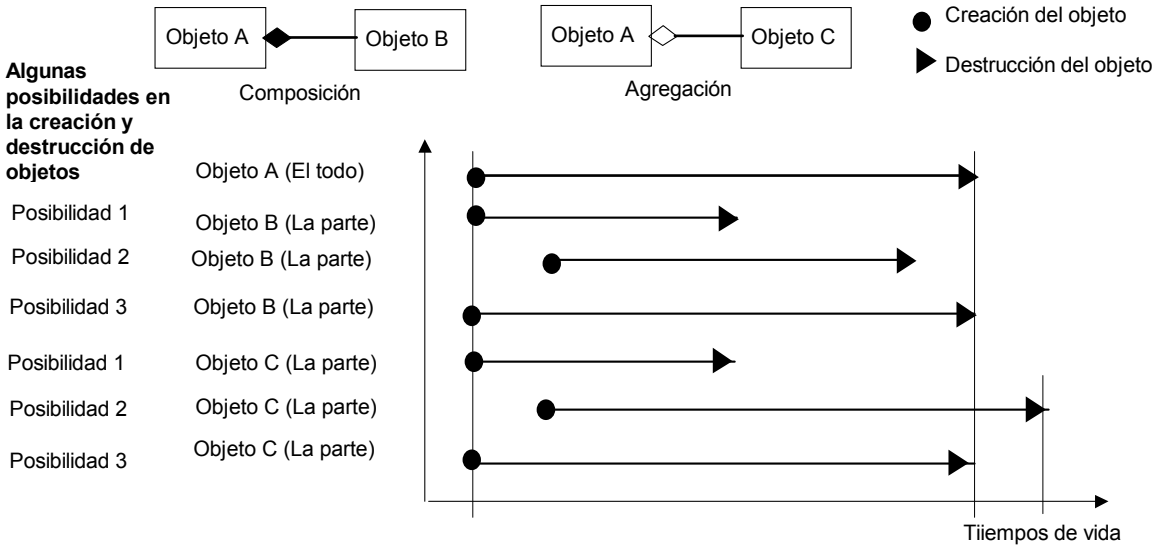


fig.12 Diferentes posibilidades de tiempos de vida, para
un objeto B y otro C, ligados a un objeto A por composición y agregación

2.2.4 Uso

Mientras que una asociación denota una conexión semántica, una relación de *uso* es un posible refinamiento de una asociación, por la que se establece cuál abstracción es el cliente y cuál abstracción es el servidor que proporciona ciertos servicios (fig.13).

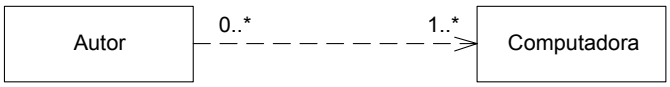


fig.13 Relación de uso

Bibliografía

[Booch 1996] Booch, G. Análisis y Diseño Orientado a Objetos con aplicaciones. Addison-Wesley/Diaz de Santos, 1996.

[Booch 1999] Booch, G.; Jacobson, I.; Rumbaugh, J. UML user guide. Addison Wesley Longman, 1999.

[Budd 1994] Budd, T. Programación orientada a objetos. Addison-Wesley Iberoamericana, 1994.

[Coad 1991] Coad, P.; Yourdon, E. Object-oriented analysis. 2da ed. Yourdon Press / Prentice-Hall, 1991.

[Coad 1997] Coad, P. Object models, strategies and patterns. Prentice-Hall, 1997.

[Cook 1994] Cook, S.; Daniels, J. Designing Object Systems. Prentice Hall, 1994.

[Chauvet 1997] Chauvet, Jean-Marie, Corba, ActiveX y JavaBeans. Eyrolles, 1997.

[Coleman 1994] Coleman, D. et al. Object-oriented software development: the Fusion method. Prentice-Hall, 1994.

[Deutsch 1991] Deutsch, L. Object-oriented software technology. IEEE Computer, vol 24, no. 9, Sept. 1991, pp. 112,113

[Devis 1997] Devis, R. C++, STL, Plantillas, Excepciones, Roles y Objetos. Editorial Paraninfo, 1997.

[El-Rewini 1995] El-Rewini, H.; Hamilton, S. Object Technology. IEEE Computer, vol.28, no. 10, Oct. 95, pp. 58-59

[Fichman 1992] Fichman, R. Object Oriented and Conventional Analysis and Design Methodologies. IEEE Computer, vol. 25, no. 10, Oct. 92, pp. 22-39

[Fowler 1995] Fowler, M. A comparison of Object-Oriented Analysis and Design Methods OOPSLA 1995.

[Fowler 1997] Fowler, M. Analysis patterns: reusable object models. Addison-Wesley, 1997.

[Gamma 1994] Gamma, E. et al. Design patterns: elements of reusable object-oriented software. Prentice-Hall, 1994.

[Garceau 1993] Garceau, L.; Jancura, E.; Kneiss, J. Object Oriented Analysis And Design: A New Approach to Systems Development. Journal of Systems Managament, no. 144, Enero 1993, pp. 25-32

[Goldberg 1989] Goldberg, A.; Robson D. Smalltalk-80 The Language. Addison-Wesley , 1989.

[Goldberg 1995] Goldberg, A.; Rubin, K. Succeeding with Objects. Addison Wesley, 1995

[Jacobson 1994] Jacobson, I. et al. Object Oriented Software Engineering. Addison - Wesley, 1994.

[Jacobson 1994a] Jacobson, I.; Ericsson, M.; Jacobson, A. The object advantage; business process reengineering. ACM Press / Addison - Wesley, 1994.

[Jacobson 1993] Jacobson. I. Is Object Technology Software's Industrial Platform?. IEEE Software, vol. 19, no. 1, Enero 93, pp. 24-30

[Jacobson 1999] Jacobson, I.; Booch, G.; Rumbaugh, J. The Unified software development process. Addison Wesley Longman, 1999

[Khan 1995] Khan.E. Object-Oriented Programming for Structured Procedural Programmers. IEEE Computer, vol. 28 no. 10, Oct. 1995, pp. 48-57

[Korson 1990] Korson, T.; McGregor J. Understanding Object-Oriented: A Unifying Paradigm. Communications of the ACM, vol. 33 no. 9, Sept. 1990, pp. 40-60

[Kozaczynski 1993] Kozaczynski, W.; Combelles, A. What it Takes to Make OO Work. IEEE Software, vol. 19, no. 1, Enero 93, pp. 20-23

[Kruchten 1999] Kruchten, Ph. The Rational Unified process. Addison Wesley Longman, 1999.

[Martin 1994] Martin, J.; Odell, J. Analisis y Diseño Orientado a Objetos. Prentice Hall Hispanoamericana S.A., 1994

[Martin 1997] Martin, J; Odell, J. Métodos orientados a objetos, consideraciones prácticas. Prentice Hall, 1997.

[McGregor 1994] McGregor, J.; Korson, T. Integrated Object-Oriented Testing and Development Processes. Communications of the ACM, vol.37, no. 9, Sept. 1994, pp. 30-38

[McMenamin 1984] McMenamin, S.; Palmer, J. Essential systems analysis. Yourdon Press, 1984.

[Meyer 1988] Meyer, B. Object-oriented software construction. Prentice-Hall, Hemel Hempstead, 1988.

[Meyer 1992] Meyer, B. Applying Design by Contract. IEEE Computer, vol. 25 no. 10, Oct. 1992, pp. 40-51

[Meyer 1996] Meyer, B. Object-oriented software construction. 2da. ed. Prentice-Hall, 1996.

[Monarchi 1992] Monarchi, D.; Puhr, G. A Research Typology for Object Oriented Analysis and Design. Communications of the ACM, vol.35, no. 9, Sept. 1992, pp. 35-47

[Nerson 1992] Nerson, Jean-Marc. Applying Object Oriented Analysis and Design. Communications of the ACM, vol. 35, no. 9, Sept. 92, pp. 63-74

[Nielsen 1995] Nielsen, K. Software Development with C++. AP Professional, 1995

[Pittman 1993] Pittman, M. Lessons Learned in Managing Object Oriented Development. IEEE Software, vol. 19, no. 1, Enero 93, pp. 43-53

[Pressman 1993] Pressman R. Ingeniería de Software, Un enfoque práctico. McGrawHill, 1994

[Radin 1996] Radin. G. Object technology in perspective. IBM Systems Journal, vol. 35, no.2, 1996, pp 124, 127

[Rine 1992] Rine, D. Object Oriented Computing. IEEE Computer, vol. 25, no. 10, Oct. 92, pp. 6-10

[Rubin 1992] Rubin, K.; Goldberg, A. Object Behavior Analysis. Communications of the ACM, vol. 35, no. 9, Sept. 1992, pp. 48-62

[Rumbaugh 1991] Rumbaugh, J. et al. Object-Oriented Modeling and Design. Prentice Hall, 1991.

[Rumbaugh 1999] Rumbaugh, J.; Booch, G.; Jacobson, I. UML reference guide. Addison Wesley Longman, 1999.

[Sánchez 1995] Sánchez, J. Reutilización de Software. Seminario de Investigación II. Instituto Tecnológico de Costa Rica, 1995.

[Sethi 1992] Sethi R. Lenguajes de Programación, Conceptos y Constructores. Addison-Wesley Iberoamericana, 1992

[Stroustroup 1993] Stroustrup, B. El Lenguaje de programación C++. Addison-Wesley Iberoamericana, 1993

[Trejos 1998] Trejos, I. Orden y método. Computer World América Central, junio 1998.

[Ungar 1992] Ungar, D.; Smith, R. Object, Message, and Performance: How They Coexist in Self. IEEE Computer, vol. 25, no. 10, Oct. 92, p.53-63

[Voss 1994] Voss. G. Programación Orientada a Objetos. McGrawHill, 1994

[Waldén 1995] Waldén, K.; Nerson, J.-M. Seamless object-oriented software architecture: analysis and design of reliable systems. Prentice-Hall, 1995.

[Webster 1995] Webster B. Pitfalls of Object-Oriented Development. M&T Books, 1995.

[Winblad 1990] Winblad, A.; Edwards, S.; King D. Software orientado a objetos. Addison Wesley/Diaz de Santos, 1990.

[Yourdon 1979] Yourdon, E.; Constantine, L. Structured design. Prentice Hall, 1979.

[Yourdon 1989] Yourdon, E. Análisis estructurado moderno. Prentice Hall Hispanoamericana, 1989 y 1993.