

Apuntes de Análisis y Diseño de Sistemas

Academia de Informática

Autores

**Paola Nayeli Cortez Herrera
Carlos de la Cruz Sosa**

**UPIITA
05/10/2011**

CONTENIDO

Unidad 1 Introducción a la Ingeniería de Software	5
1.1 Conceptos fundamentales del software (dato, información)	5
1.2 Definición del software.....	7
1.2.1 Características del software.....	9
1.2.2 Importancia del Software	11
1.2.3 Complejidad del diseño y desarrollo del software.	11
1.2.4 Crisis del software.....	14
1.2.5 Aspectos a considerar en el diseño del software	15
1.2.6 Claves en el diseño del software.	16
1.3 Sistemas de información.....	18
1.3.1Definición de los Sistemas de información.....	19
1.3.2Tipos de sistemas de información	22
1.4Análisis de requisitos.....	24
1.4.1Importancia del Análisis de requisitos.....	25
1.4.2 Obtención de los requisitos.	27
1.4.3 Interpretación y manejo de los requisitos.....	31
1.5Ciclos de Vida.	32
1.5.1Cascada.....	32
1.5.2Espiral	34
1.6Legislación del software.....	36
1.7Metodologías para el desarrollo del software.....	37
1.7.1Definición, tipos de metodologías y elementos.	37
1.7.2 Análisis de una metodología.....	38
1.7.3Desarrollo a pasos de una metodología.	39
Unidad 2 Fundamentos de Programación en C++	41

2.1 Estructura del Lenguaje.....	41
2.2 Funciones de entrada y salida en C++: cin<< y cout>>.....	42
2.3 Declaración de variables y constantes	43
2.4 Operadores	43
2.4.1 Operadores de asignación	43
2.4.2 Operadores matematicos:unarios y binarios.....	44
2.4.3 Operadores LÓgicos.....	45
2.4.4 Operadores relacionales.....	45
2.5 Sentencias de control.....	46
2.5.1 Sentencias condicionales: If – else, Switch	46
2.5.2 Sentencias de repetición: for, while, do – while.....	49
2.6 Arreglos unidimensionales y bidimensionales	51
2.6.1 Lectura y escritura en arreglos	54
2.6.2 Operaciones básicas con arreglos.....	54
2.6.3 Métodos de ordenamiento.....	55
2.7 Funciones (métodos).....	57
2.7.1 Tipos de funciones	58
2.7.2 Llamado a funciones	59
Unidad 3 Fundamentos del Paradigma Orientada a Objetos.....	61
3.1 Definición del Paradigma orientado a objetos.....	61
3.2 Definicion de clases y objetos en c++.....	62
3.2.1 Alcance de clases en C++	62
3.2.2 Atributos de clases en C++.....	63
3.2.3 Sobrecarga de clases en C++.....	66
3.3. Definición y creación de métodos.....	70
3.3.1 constructores en c++	70
3.3.2 sobrecarga de metodos en c++.....	71

3.3.3 sobrecarga de constructores en c++.....	72
3.4 Encapsulado	75
3.4.1 Encapsulado de clases	75
3.5. Herencia simple y múltiple.....	80
3.5.1. Definición de clases simples	82
3.5.2 Clases abstractas.....	84
3.6 polimorfismo	85
UNIDAD 4 UML	90
4.1 Introducción a UML.....	90
4.2 Herramientas del Modelado	91
4.2.1 Diagramas de flujo de datos (DFD)	92
4.2.2 Diagramas Entidad Relación	94
4.2.3 Diagramas de Transición de Estados	94
4.2.4 Diccionario de Datos	95
4.2.5 Especificación de Procesos	95
4.3 Diagramas básicos de UML	96
4.3.1 Diagramas de Casos de Uso	105
4.4 Diagramas de Actividades	110
4.4.1 Operaciones	112
4.4.2 Transiciones	113
4.5 Diagramas de Clases.....	114
4.5.1 Relaciones	115
4.5.2 Atributos	120
4.5.3 Operaciones	121
4.6 Diagramas de secuencia	122
4.6.1 Mensajes.....	124
4.7 Diagramas de Comunicación (Diagramas de Colaboración en UML 1.x)	125

4.8 Diagramas de Despliegue (Distribución)	128
4.8.1 Estados.....	129
4.8.2 Transiciones	131
4.9 Casos prácticos.....	134
Hotel	134
Clasificación de las manzanas	138
BIBLIOGRAFIA	143
Referencias a páginas web	143

UNIDAD 1 INTRODUCCIÓN A LA INGENIERÍA DE SOFTWARE

1.1 CONCEPTOS FUNDAMENTALES DEL SOFTWARE (DATO, INFORMACIÓN)

Se debe hacer una distinción entre datos e información, términos que en ocasiones se pueden llegar a confundir.

Los datos reflejan hechos recogidos en la organización y que están todavía sin procesar, mientras que la información se obtiene una vez que estos hechos se procesa, agregan y presentan de la manera adecuada para que puedan ser útiles a alguien dentro de la organización, por lo que de este modo estos datos organizados y procesados presentan un mayor valor que en su estado original.

Los datos quedan perfectamente identificados por elementos simbólicos (letras y números), que reflejan valores o resultados de mediciones.

Sin embargo, la información son “datos dotados de relevancia y propósito”, como señala Peter Drucker, que permiten reducir la incertidumbre de quien los recibe. La figura 1.1 representa la forma en como los datos son convertidos en información.

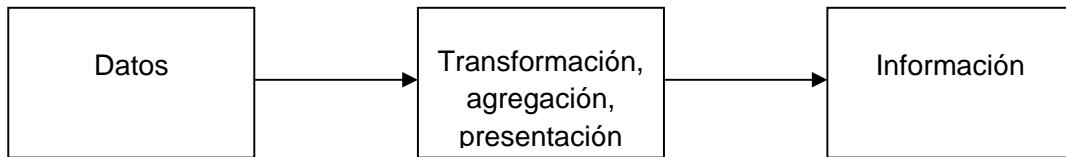


Figura 0.1

El proceso de transformación de los datos en información.

Características que debe cumplir la información

La información será útil para la organización en la medida en que facilite la toma de decisiones y, para ello, ha de cumplir una serie de requisitos, entre los que cabe citar:

- **Exactitud:** la información ha de ser precisa y libre de errores.
- **Completitud:** la información debe contener todos aquellos hechos que pudieran ser importantes para la persona que la va a utilizar.
- **Economicidad:** el coste en que se debe incurrir para obtener la información debería ser menor que el beneficio proporcionado por ésta a la organización.

- **Confianza:** para dar crédito a la información obtenida, se ha de garantizar tanto la calidad de los datos utilizados, como la de las fuentes de información.
- **Relevancia:** la información ha de ser útil para la toma de decisiones. En este sentido, conviene evitar todos aquellos hechos que sean superfluos o que no aporten ningún valor.
- **Nivel de detalle:** la información debería presentar el nivel de detalle indicado a la decisión que se destina. Se debe proporcionar con la presentación y el formato adecuados, para que resulte sencilla y fácil de manejar.
- **Oportunidad:** se debe entregar la información a la persona que corresponde y en el momento en que ésta la necesita para poder tomar una decisión.
- **Verificabilidad:** la información ha de poder ser contrastada y comprobada en todo momento.

No se debe olvidar que el exceso de información también puede ser causa de problemas y suponer un obstáculo en vez de una ayuda en la toma de decisiones. En este sentido conviene indicar que acotar las necesidades de información de cualquier organización es un proceso que debería ser continuado o sistemático.

Aportar datos a una organización es simple: hoy son muchas las organizaciones que cuentan con un nivel elevado de informatización en sus distintos procesos, y generan bases de datos que registran muchas de las operaciones que se realizan (ver tabla 1.1). A partir de estas bases de datos no resulta compleja la generación de informes o estadísticas e inclusive la aplicación de tecnologías avanzadas para el análisis de los datos. Sin embargo, proporcionar a las distintas personas de la organización la información realmente necesaria para su función, exige un conocimiento importante de los procesos que se desarrolla, así como de la función desarrollada por cada usuario, e incluso del perfil y preferencias de la persona que utilizará la información.

Nivel Organizacional	Necesidades de Información
Estrategia	Alto nivel de agregación Información muy vinculada a los objetivos globales Alto valor de la información del entorno
Control de Gestión	Nivel de agregación medio Posibilidad de “bajar al detalle” Análisis, tendencias y comparativa
Operativo	Alto nivel de detalle Necesidad corto plazo

Tabla 1.1 Clasificación de los niveles organizacionales y las necesidades de la información

En definitiva, la información y le conocimiento que acumulan las organizaciones deberán ser considerados como un recurso más, al mismo nivel que el capital, los bienes e instalaciones o el personal. En consecuencia, es necesario gestionarlo y explotarlo adecuadamente, para que pueda contribuir a la consecución de las metas y objetivos fijados por la organización.

Del mismo modo que existe un proceso para lograr información a partir de los datos, tal como se ha descrito, también existe un proceso por el que la información es transformada en conocimiento a partir de la experiencia (figura 1.2.).

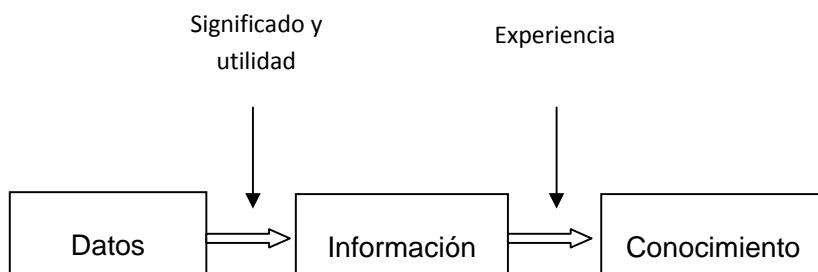


Figura 1.2 El proceso de transformación de los datos en información

Tal y como exponen Davenport, Long y Beers (1998), “*a diferencias de los datos, el conocimiento se produce de manera invisible en la mente humana, y sólo un adecuado clima empresarial puede convencer a la gente para crear, revelar, compartir y utilizar ese conocimiento. A causa del factor humano del conocimiento, es deseable contar con una estructura flexible y que fomente el desarrollo, y son muy importantes las motivaciones para crear, compartir y utilizar los conocimientos. Los datos y la información se transmiten constantemente por medios electrónicos, pero el conocimiento parece viajar más a gusto a través de una red humana*”.

1.2 DEFINICIÓN DEL SOFTWARE.

En la actualidad, el software tiene un papel dual. Es a la vez, un producto y un vehículo mediante el cual se entrega un producto. Como producto, ofrece la potencia de cómputo presentada como hardware de una computadora o, de manera más amplia, por una red de computadoras accesible mediante hardware local. Sin importar el lugar en que resida el software, ya sea en un celular o dentro de una computadora central, éste es un transformador de información; realiza la producción, el manejo, la adquisición, la modificación, el despliegue o la transmisión de la información que puede ser tan simple como un solo bit o tan compleja como una presentación multimedia. En su papel de vehículo para la entrega de un producto, el software actúa como la base para el control de

la computadora (sistemas operativos), la comunicación de información (redes), y la creación y el control de otros programas (utilerías de software y ambientes).

El software entrega el producto más importante de nuestro tiempo: *información*. Transforma los datos personales (por ejemplo, las transacciones financieras de un individuo) de forma que los datos sean más útiles en un contexto local; maneja información de negocios para mejorar la competitividad; proporciona una vía para las redes de información alrededor del mundo (Internet) y proporciona los medios para adquirir información en todas sus formas.

El software se ha convertido en el elemento clave de la evolución de los sistemas y productos basados en computadoras, así como en una de las tecnologías más importantes en el ámbito mundial. En los pasados 50 años, el software ha evolucionado desde ser una herramienta para la solución de problemas especializados y el análisis de información, hasta convertirse en una industria por sí mismo. Todavía se tienen problemas al desarrollar software de alta calidad a tiempo y dentro del presupuesto. El software – programas, datos y documentos – se dirige a un amplio espectro de tecnologías y áreas de aplicación. En la actualidad el software evoluciona de acuerdo con un conjunto de leyes que han permanecido inalteradas a lo largo de 30 años. La intención de la ingeniería del software es proporcionar un marco general para construir software con una calidad mucho mayor.

Definiciones:

- El software de computadora es el producto que los ingenieros de software construyen y después mantienen en el largo plazo. Incluye los programas que se ejecutan dentro de una computadora de cualquier tamaño y arquitectura, el contenido que se presenta conforme los programas se ejecutan y los documentos, tanto físicos como virtuales, que engloban todas las formas de medios electrónicos. (Pressman)
- El software no son sólo programas, sino todos los documentos asociados y la configuración de datos que se necesitan para hacer que estos programas operen de manera correcta. Por lo general, un sistema de software consiste en diversos programas independientes, archivos de configuración que se utilizan para ejecutar estos programas, un sistema de documentación describe la estructura del sistema, la documentación para el usuario que explica cómo utilizar el sistema y sitios web que permitan a los usuarios descargar la información de productos recientes. (Sommerville)
- Programas informáticos y la documentación asociada.

Los productos software que pueden ser desarrollados para un cliente en particular o para el mercado en general.

Y son clasificados en:

- *Genéricos*.- desarrollados para ser vendidos a un amplio rango de clientes diferentes
- *Particulares* (por encargo) .- desarrollado para un cliente individual de acuerdo a sus necesidades

En 1970 menos del uno por ciento de las personas podrían haber definido lo que significaba “software de computadora”. En la actualidad, la mayoría de los profesionales y muchos miembros del público creen que entienden el software. Pero, ¿en realidad lo hacen?

Una definición de software en un libro de texto pudo tener la siguiente forma: *el software se forma con 1) las instrucciones (programas de computadora) que al ejecutarse proporcionan las características, funciones y el grado de desempeño deseados.; 2) las estructuras de datos que permiten que los programas manipulen información de manera adecuada; y 3) los documentos que describen la operación y el uso de los programas.*

No existe duda de que se pueden encontrar definiciones más completas. Pero se requiere más que una definición formal.

Para entender el software (y la ingeniería de software), es importante examinar las características que lo hacen diferente de otras cosas que construye el ser humano. El software es un elemento lógico, en un lugar físico, de un sistema. Por lo tanto, el software tiene características muy diferentes a las del hardware.

1.2.1 CARACTERÍSTICAS DEL SOFTWARE.

El software tiene características muy diferentes a las del hardware, entre ellas están:

1. *El software se desarrolla o se construye; no se manufactura en el sentido clásico.*
A pesar de que existen similitudes entre el desarrollo del software y la manufactura del hardware, las dos actividades son diferentes en lo fundamental. En ambas, la alta calidad se alcanza por medio del buen diseño, pero la fase de manufactura del hardware puede incluir problemas de calidad inexistentes (o que son fáciles de corregir) en el software. Ambas actividades requieren la construcción de un “producto”, pero los enfoques son diferentes. Los costos del software se concentran en la ingeniería. Esto significa que los proyectos de software no se pueden manejar como si fueran proyectos de manufactura.
2. *El software no se “desgasta”*
El hardware tiene un número considerable alto de fallas al inicio de su vida (a menudo éstas se atribuyen a defectos de diseño o manufactura). Después, los

defectos se corrigen y la tasa de fallas baja hasta un nivel estable (se desea que éste sea muy bajo) por algún período. Sin embargo, conforme pasa el tiempo, la falla se eleva de nuevo conforme los componentes del hardware sufren los efectos acumulativos del polvo, la vibración, el abuso, las temperaturas extremas y muchos otros males ambientales. Expresado en forma más simple, el hardware comienza a desgastarse.

El software es inmune a los males ambientales que desgastan el hardware. Los defectos sin descubrir causan tasas de falla altas en las primeras etapas de vida de un programa. Sin embargo, los errores se corrigen (en el mejor de los casos sin agregar otros errores). Esto concluye que el software no se desgasta, pero sí se deteriora.

Durante su vida, el software experimenta cambios. Conforme estos ocurren se presenta la posibilidad de introducir errores, lo que puede ocasionar fallas. Por lo cual el software se deteriora debido a los cambios.

Otro aspecto del desgaste ilustra la diferencia entre el hardware y el software. Cuando un componente del hardware se desgasta se sustituye con un repuesto. Pero en el software no existen repuestos. Cualquier falla del software implica un error en el diseño o en el proceso mediante el cual se pasó del diseño al código máquina ejecutable. Por lo tanto, el mantenimiento del software implica de manera considerable una complejidad mayor que el del hardware.

3. A pesar de que la industria tiene una tendencia hacia la construcción por componentes, la mayoría del software aún se construye a la medida.

Considérese la forma en que se diseña y construye un hardware de control para un producto de cómputo. El ingeniero de diseño dibuja un esquema simple del sistema de circuitos digital, realiza algunos análisis fundamentales para asegurarse de que el diseño realizará las funciones apropiadas y después busca en los catálogos de componentes digitales cada circuito integrado de acuerdo con un numero de parte, una función definida y validada, una interfaz bien definida y un conjunto estandarizado de directrices de integración. Una vez seleccionado cada componente, puede solicitarse para después ensamblarlo.

Cuando una disciplina de ingeniería evoluciona se crea una colección de diseños estándar de componentes. Los tornillos y los circuitos integrados son solo dos ejemplos de los miles de componentes estándar que utilizan los ingenieros mecánicos y eléctricos al diseñar sistemas nuevos. Los componentes reutilizables se han creado para que el ingeniero se pueda concentrar en los elementos que en realidad son innovadores en el diseño, es decir, en las partes que representan algo nuevo. En el mundo del hardware, la reutilización de componentes es una parte natural del proceso de ingeniería. En el ámbito del software, dicha actividad apenas se ha comenzado a extender.

1.2.2 IMPORTANCIA DEL SOFTWARE

Actualmente casi todos los países dependen de complejos sistemas informáticos. Infraestructuras nacionales y utilidades dependen de sistemas informáticos, y la mayor parte de los productos eléctricos incluyen una computadora y software de control. La fabricación industrial y distribución está completamente informatizada, como el sistema financiero. Por lo tanto, producir software costeable es esencial para el funcionamiento de la economía nacional e internacional.

La ingeniería del software es una disciplina de la ingeniería cuya meta es el desarrollo costeable de sistemas de software. Éste es abstracto e intangible. No está restringido por materiales. O gobernado por leyes físicas o por procesos de manufactura. De alguna forma, esto simplifica la ingeniería del software ya que no existen limitaciones físicas del potencial del software. Sin embargo, esta falta de restricciones naturales significa que el software puede llegar a ser extremadamente complejo y, por lo tanto, muy difícil de entender.

Se puede afirmar que se han hecho enormes progresos desde 1968 y que el desarrollo de esta ingeniería ha mejorado considerablemente nuestro software. Se comprenden mucho mejor las actividades involucradas en el desarrollo de software. Se han desarrollado métodos efectivos de especificación, diseño e implementación del software. Las nuevas notaciones y herramientas reducen el esfuerzo requerido para producir sistemas grandes y complejos.

Ahora se sabe que no hay un enfoque <<ideal>> a la ingeniería de software. La amplia diversidad de diferentes tipos de sistemas y organizaciones que usan estos sistemas significa que necesitamos una diversidad de enfoques al desarrollo de software. Sin embargo, las nociones fundamentales de procesos y la organización del sistema son la base de todas estas técnicas, y éstas son la esencia de la ingeniería del software.

Los ingenieros de software pueden estar orgullosos de sus logros. Sin software complejo no habríamos explorado el espacio, no se tendría Internet y telecomunicaciones modernas, y todas las formas de viajar serían más peligrosas y caras. Dicha ingeniería ha hecho entonces enormes contribuciones, y no cabe duda de que, en cuanto la disciplina madure, su contribución en el siglo XXI será aún más grande.

1.2.3 COMPLEJIDAD DEL DISEÑO Y DESARROLLO DEL SOFTWARE.

Como sugiere Brooks, <<la complejidad del software es una propiedad esencial, no accidental>>. Se observa que esta complejidad inherente se deriva de cuatro elementos: la complejidad del dominio del problema, la necesidad de gestionar el proceso de desarrollo, la flexibilidad que se puede alcanzar a través del software y los problemas que plantea la caracterización del comportamiento de sistemas discretos.

La complejidad del dominio del problema. Los problemas que se intentan resolver con el software conllevan a menudo a elementos de complejidad ineludible, de requisitos que compiten entre sí y que quizás incluso en ocasiones se contradicen. Considérese los requisitos para el sistema electrónico de un avión multimotor, un sistema de conmutación para teléfonos celulares o un robot autónomo. La funcionalidad pura de tales sistemas es difícil incluso de comprender, pero añádase además todos los requerimientos no funcionales, tales como facilidad de uso, rendimiento, coste, capacidad de supervivencia y fiabilidad, que a menudo están implícitos. (Ver figura 1.3)

La forma habitual de expresar requisitos hoy en día es mediante grandes cantidades de texto y ocasionalmente acompañadas de unos pocos dibujos. Estos documentos son difíciles de comprender, debido a estar abiertos a diversas interpretaciones.

Una complicación adicional es que los requisitos de un sistema de software cambian frecuentemente durante su desarrollo. La observación de productos de las primeras fases como documentación de diseño y prototipos y la posterior utilización de un sistema son factores que llevan a los usuarios a comprender y articular mejor sus capacidades reales.

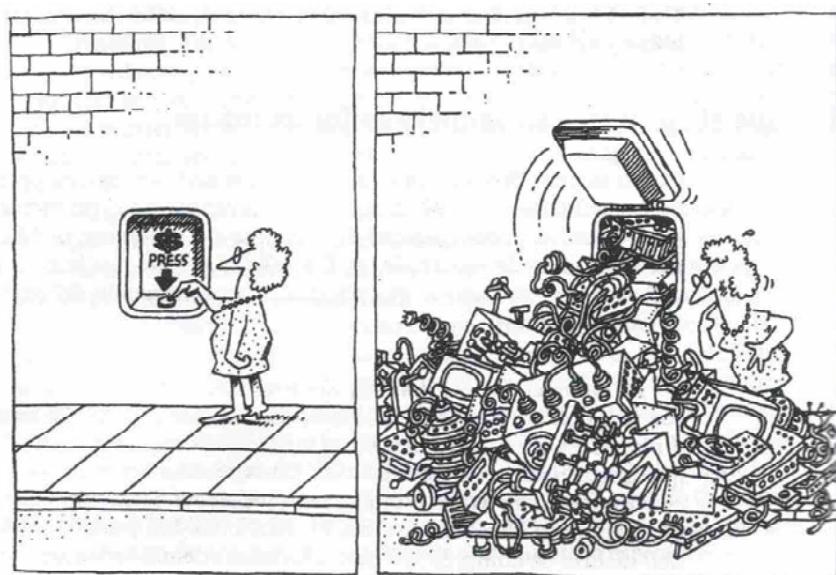


Figura 1.3 La tarea del equipo de desarrollo de software es ofrecer ilusión de simplicidad

Ya que un sistema grande es una inversión considerable es inadmisible desechar un sistema existente cada vez que los requerimientos cambian.

Esto da por resultado la aclaración de 3 conceptos:

Mantenimiento del software: Se considera mantenimiento cuando se originan errores.

Evolución: se produce cuando se responde a requerimientos que cambian y;

Conservación: cuando se siguen empleando medios extraordinarios para mantener en operación un elemento de software anticuado y decadente.

Desafortunadamente, la realidad sugiere que un porcentaje exagerado de los recursos de desarrollo de software se emplean en conservación del mismo.

La dificultad de gestionar el proceso de desarrollo. La tarea fundamental del equipo de desarrollo de software es dar vida a una ilusión de simplicidad. Ciertamente, el tamaño no es una gran virtud para un sistema de software. Se hace lo posible por escribir menos código mediante la invención de mecanismos ingeniosos y potentes que dan la ilusión de simplicidad, así como la reutilización de marcos estructurales de diseño y código ya existentes.

Un código grande exige la utilización de un equipo de desarrolladores, y de forma ideal se utiliza un equipo tan pequeño como sea posible. Sin embargo, da igual el tamaño, siempre hay retos considerables asociados con el desarrollo en equipo. Un mayor número de miembros implica una comunicación más compleja y por tanto una coordinación más difícil, particularmente si el equipo está disperso geográficamente, y esta situación no es nada excepcional en proyectos muy grandes. Con un equipo de desarrolladores, el reto clave de la dirección es siempre mantener una unidad e integridad en el diseño.

La flexibilidad que se puede alcanzar a través del software. El software ofrece la flexibilidad máxima por lo que un desarrollador puede expresar casi cualquier clase de abstracción. Esta flexibilidad resulta ser una propiedad que seduce increíblemente, ya que también empuja al programador a construir por sí mismo, prácticamente todos los bloques fundamentales sobre los que se apoyan estas abstracciones de más alto nivel. Un ejemplo es que la industria de la construcción tiene normas y la de software tiene muy pocos estándares. Como consecuencia, el desarrollo del software sigue siendo un negocio enormemente laborioso.

Los problemas de caracterizar el comportamiento de sistemas discretos. En una aplicación de gran tamaño puede haber cientos o hasta miles de variables, así como más de un posible flujo de control. El conjunto de todas estas variables, sus valores actuales, y la dirección de ejecución y pila actual de cada uno de los procesos del sistema constituyen el estado actual de la aplicación. Al ejecutarse el software en computadoras digitales se obtiene un sistema con estados discretos. Los sistemas discretos por su propia naturaleza tienen un número finito de estados posibles; en sistemas grandes hay una explosión combinatoria que hace este número enorme. Todos los eventos externos a un sistema de software tienen la posibilidad de llevar a ese sistema a un nuevo estado, y más aun, la transición de estado a estado no siempre es determinista. En las peores circunstancias un evento externo puede corromper el estado del sistema, porque sus diseñadores olvidaron tener en cuenta ciertas interacciones entre eventos. Se deben de probar a fondo todos los sistemas pero para cualquier sistema que no sea trivial, es imposible hacer una prueba exhaustiva. Ya que no se dispone de herramientas matemáticas ni de la capacidad intelectual necesarias para modelar el comportamiento

completo de un grande sistema discreto, hay que contentarse con un grado de confianza aceptable por lo que se refiere a su corrección.

1.2.4 CRISIS DEL SOFTWARE

La noción de *ingeniería del software* fue propuesta inicialmente en 1968 en una conferencia para discutir lo que en ese entonces se llamo la <<crisis del software>>. Esta crisis del software fue el resultado de la introducción de las nuevas computadoras hardware basadas en circuitos integrados. Su poder hizo que las aplicaciones hasta ese entonces irrealizables fueran una propuesta factible. El software resultante fue de órdenes de magnitud más grande y más complejo que los sistemas de software previos.

La experiencia previa en la construcción de sistemas sin estándares mostro que un enfoque informal para el desarrollo del software no era muy bueno. Los grandes proyectos a menudo tenían años de retraso. Costaban mucho más de lo presupuestado, eran irrealizables, difíciles de mantener y con un desempeño pobre. El desarrollo de software estaba en crisis. Los costos del hardware se tambaleaban mientras que los del software se incrementaban con rapidez. Se necesitaban nuevas técnicas y métodos para controlar la complejidad inherente a los sistemas grandes.

Estas técnicas han llegado a ser parte de la ingeniería de software y son ampliamente utilizadas. Sin embargo, cuanto más crezca nuestra capacidad para producir software, también lo hará la complejidad de los sistemas de software solicitados. Las nuevas tecnologías resultan de la convergencia de las computadoras y de los sistemas de comunicación y complejas interfaces gráficas de usuario impusieron nuevas demandas a los ingenieros de software. Debido a que muchas compañías no aplican de forma efectiva las técnicas de la ingeniería del software, demasiados proyectos todavía producen software que es irrealizable, entregado tarde y sobre presupuestado. Para tener un panorama general de esta situación ver la figura 1.4

Por lo tanto se puede resumir la crisis del software en estos cuatro puntos:

- Hace referencia a un conjunto de problemas encontrados en el desarrollo del software (años 70's).
- A partir de los 70, se realiza grandes avances en la tecnología del hardware que hace que su conste disminuya y que puedan aportarse soluciones a problemas más complejos.
- La evolución del software por el contrario ha sido más lenta y por tanto, puede decirse que el cuello de la botella de la informática está en la optimización de la producción del software.
- El coste informático de una empresa se centra en el coste producido por el software.

- Todo esto ha hecho que:
 - Estudio que lleva consigo el desarrollo del software.
 - Nuevos conceptos, control de calidad (SQA), metodologías de análisis y diseño, ingeniería del software, etc.

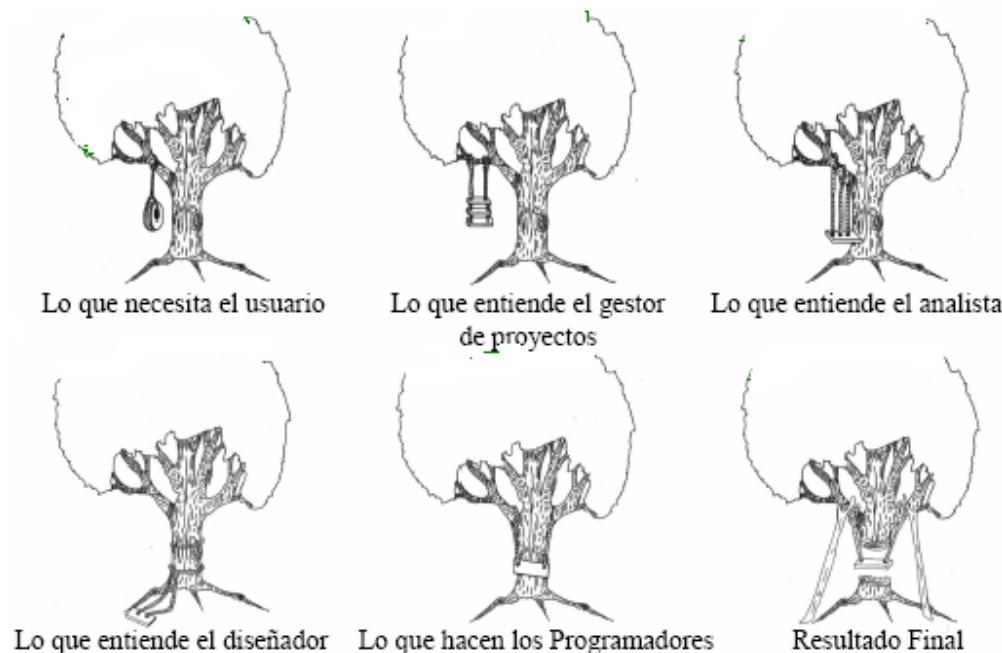


Figura 1.4 Representación visual de factores que contribuyen a que exista la ingeniería de software

1.2.5 ASPECTOS A CONSIDERAR EN EL DESARROLLO DE SOFTWARE

La práctica de cualquier disciplina de ingeniería – sea civil, mecánica, química, eléctrica o informática – involucra elementos tanto de ciencia como de arte. Como Petroski elocuentemente afirma, <<la concepción de un diseño para una nueva estructura puede involucrar un salto de la imaginación y una síntesis de experiencia y conocimiento tan grandes como el que requiere cualquier artista para plasmar su obra en una tela o en un papel. Y una vez que el ingeniero ha articulado ese diseño como artista, debe analizarlo como científico aplicando el método científico con tanto rigor como cualquier científico lo haría>>.

Cualquier proyecto de software se inicia por alguna necesidad de negocios: la necesidad de corregir un defecto en una aplicación existente; el imperativo de adaptar un sistema heredado a un ambiente de negocios cambiante; el requerimiento de extender las funciones y características de una aplicación existente; o la necesidad de crear un producto, servicio o sistema nuevos.

Con frecuencia, en el inicio de un proyecto de ingeniería del software la necesidad de negocios se expresa de manera informal durante una simple conversación.

El papel del ingeniero como artista es particularmente difícil cuando la tarea es diseñar un sistema completamente nuevo. Francamente, es la circunstancia más habitual en la ingeniería del software. Especialmente en el caso de sistemas reactivos y sistemas de dirección y control, se pide con frecuencia que se escriba software para un conjunto de requerimientos completamente exclusivo, muchas veces para ser ejecutado en una configuración de procesadores que se ha construido específicamente para este sistema. En otros casos, como en la creación de marcos estructurales, herramientas para investigación en inteligencia artificial, o incluso sistemas de gestión de la información, puede ser que exista un entorno de destino estable y bien definido, pero los requerimientos pueden llevar al límite a la tecnología del software en una o más dimensiones. Por ejemplo, puede hacerse recibido la solicitud de crear sistemas más rápidos, de mayor capacidad, o de una funcionalidad radicalmente mejorada. En todas estas situaciones, se intenta utilizar abstracciones y mecanismos ya probados (lo que Simon llamaba <>formas intermedias estables<>) como fundamento sobre el que construir nuevos sistemas complejos. En presencia de una gran biblioteca de componentes de software reusables, el ingeniero de software debe ensamblar estas partes de formas innovadoras para satisfacer los requerimientos expresados e implícitos, al igual que el pintor y el músico deben ampliar los límites impuestos por su medio. Desgraciadamente, el ingeniero de software sólo dispone de bibliotecas tan ricas en muy contadas ocasiones, lo habitual es que deba echar mano de un conjunto relativamente primitivo de utilidades.

1.2.6 CLAVES EN EL DISEÑO DEL SOFTWARE.

Así como los servicios que proveen, los productos de software tienen un cierto número de atributos asociados que reflejan la calidad de ese software. Estos atributos no están directamente asociados con lo que el software hace. Más bien, reflejan su comportamiento durante su ejecución y en la estructura y organización del programa fuente y en la documentación asociada. Ejemplos de estos atributos (algunas veces llamados atributos no funcionales) son el tiempo de respuesta del software a una pregunta del usuario y la comprensión del programa fuente.

El conjunto específico de atributos que se espera de un sistema de software depende obviamente de su aplicación. Por lo tanto, un sistema bancario debe ser seguro, un juego interactivo debe tener capacidad de respuesta, un interruptor de un sistema telefónico debe seriable, etcétera. Esto se generaliza en el conjunto de atributos que se muestra en la tabla 1.2 el cual tiene las características esenciales de un sistema de software bien diseñado.

Características del Software	
Mantenibilidad	El software debe escribirse de tal forma que pueda evolucionar para cumplir las necesidades de cambio de los clientes. Éste es un atributo crítico debido a que el cambio en el software es una consecuencia inevitable de un cambio en el entorno de negocios.
Confiabilidad	La confiabilidad del software tiene un gran número de características, incluyendo la fiabilidad, protección y seguridad. El software confiable no debe causar daños físicos o económicos en el caso de una falla en el sistema.
Eficiencia	El software no debe hacer que se malgasten los recursos del sistema, como la memoria y los ciclos de procesamiento. Por lo tanto, la eficiencia incluye tiempos de respuesta y de procesamiento, utilización de la memoria, etc.
Usabilidad	El software debe ser fácil de utilizar, sin esfuerzo adicional, por el usuario para quien está diseñado. Esto significa que debe tener una interfaz de usuario apropiada y una documentación adecuada.

Tabla 1.2 Atributos esenciales de un buen software

En la actualidad una enorme industria del software se ha convertido en un factor dominante en la economía del mundo industrializado. El programador solitario de la era inicial ha sido sustituido por equipos de especialistas en software, en los que cada uno se enfoca en una parte de la tecnología requerida para desarrollar una aplicación compleja. Hasta ahora, las preguntas formuladas al programador solitario son las mismas que se hacen cuando se construyen los sistemas basados en computadoras modernas:

- ¿Por qué tarda tanto la obtención del software terminado?
- ¿Por qué son tan altos los costos de desarrollo del software?
- ¿Por qué es imposible encontrar todos los errores en el software antes de entregarlo a los clientes?
- ¿Por qué se gastan tanto tiempo y esfuerzo en el mantenimiento de los programas existentes?
- ¿Por qué es difícil medir el progreso al desarrollar y darle mantenimiento al software?

Estas y muchas otras preguntas demuestran la preocupación de la industria por el software y por la manera en que éste se desarrolla; una preocupación que ha conducido a la adopción de la práctica de la ingeniería de software.

1.3 SISTEMAS DE INFORMACIÓN

Hoy en día, los Sistemas de Información juegan un papel cada vez más importante en las modernas organizaciones empresariales, hasta el punto de condicionar su éxito o fracaso en un entorno económico y social tan dinámico y turbulento como el que caracteriza al mundo actual.

Nuevos fenómenos como la globalización o el tránsito hacia una economía más basada en el conocimiento han inducido importantes cambios en las organizaciones empresariales. En este nuevo contexto, los Sistemas y las Tecnologías de la Información y Comunicación (TIC) se han convertido en un elemento esencial como motor del cambio y fuente de ventajas competitivas.

Las TIC han permitido acelerar el proceso de creación y de distribución del conocimiento científico y tecnológico, potenciando las capacidades de los agentes económicos. En la actualidad disponemos de una base tecnológica que no sólo puede sustituir al trabajo manual (principal característica de las tecnologías manufactureras de la economía industrial), sino que también ayuda al hombre en el proceso de generación y difusión del saber, es decir, en el trabajo mental de los nuevos “trabajadores del conocimiento” (Vilaseca y Torrent, 2003).

Dentro de una organización el Sistema de Información actúa como el “sistema nervioso”, ya que éste es el que se encarga de hacer llegar a tiempo la información que necesitan los distintos elementos de la organización empresarial (departamentos, áreas funcionales, equipos de trabajo, delegaciones, etc.) permitiendo de esta forma una actuación conjunta y coordinada, ágil y orientada hacia los resultados.

Los sistemas de información han adquirido una dimensión estratégica en las empresas del nuevo milenio y han dejado de ser considerados como una simple herramienta para automatizar procesos operativos para convertirse en una pieza clave a tener en cuenta a la hora de formular la estrategia empresarial, para llevar a cabo su implantación y para realizar el control de la gestión.

Los Sistemas de Información no sólo llegan a condicionar la estrategia de la moderna empresa, sino que, además, constituyen el elemento fundamental para poder llevar a cabo una gestión de la empresa, orientada a procesos y no a funciones, que permita poner el énfasis en la mejora continua de los resultados, con una clara orientación total hacia el cliente.

Éste es un aspecto que hoy en día se considera clave, no ya para alcanzar el éxito, sino para garantizar la supervivencia de la organización en un entorno tan competitivo y exigente como el actual.

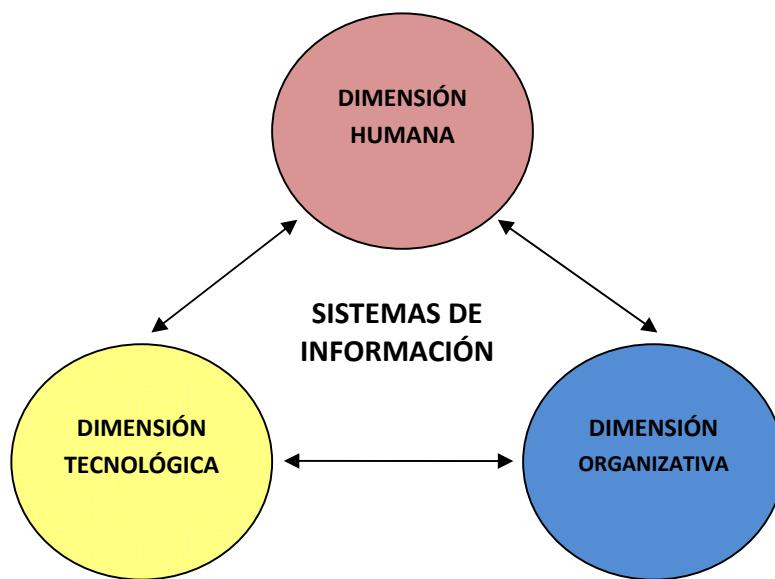


Figura 1.5 La triple dimensión Humana, Organizativa y Tecnológica de los Sistemas de Información

1.3.1 DEFINICIÓN DE LOS SISTEMAS DE INFORMACIÓN

Al definir con una sola frase el principal cometido de un Sistema de Información dentro de una organización, podríamos afirmar que éste se encarga de entregar la información oportuna y precisa, con la presentación y el formato adecuados, a la persona que la necesita dentro de la organización para tomar una decisión o realizar alguna operación y justo en el momento en que esta persona necesita disponer de dicha información.

Hoy en día, la información debería ser considerada como uno de los más valiosos recursos de una organización y el Sistema de Información es el encargado de que ésta sea gestionada siguiendo criterios de eficacia y eficiencia.

ESTRUCTURA DEL SISTEMA DE INFORMACIÓN

A la hora de identificar los principales componentes integrantes de un Sistema de Información, los distintos autores expertos en la materia coinciden en sus planteamientos.

Ralph Stair afirma que un Sistema de Información es un sistema compuesto por personas, procedimientos, equipamiento informático (distinguiendo entre hardware y software), bases de datos y elementos de telecomunicaciones.

Whitten, Bentley y Barlow proponen un modelo basado en cinco bloques elementales para definir un Sistema de Información: personas, actividades, datos, redes y tecnología.

El bloque “personas” engloba a los propietarios del sistema (entendiendo como tales a aquellas personas que patrocinan y promueven el desarrollo de los Sistemas de Información), a los usuario (directivos, ejecutivos, directivos medios, jefes de equipo, personal administrativo...), a los diseñadores y a los que implementan el sistema.

Los “datos” constituyen la “materia prima” empleada para crear información útil.

Dentro del bloque “actividades”, se incluyen las actividades (procesos) que se llevan a cabo en la empresa y las actividades de proceso de datos y generación de información que sirven de soporte a las primeras.

En el bloque “redes” se analiza la descentralización de la empresa y la distribución de los restantes bloques elementales en los lugares más útiles (centros de producción, oficinas, delegaciones...), así como la comunicación y coordinación entre dichos lugares.

Por último, el bloque “tecnología” hace referencia tanto al hardware como al software que sirven de apoyo a los restantes bloques integrantes del Sistema de Información.

La figura 1.6 presenta la interrelación existente entre los componentes de un sistema de información. Se pone de manifiesto la existencia de una interrelación entre los elementos propios de la organización y los sistemas de información.

Cambios en estos componentes dan lugar a cambios en el software, en el hardware o en las comunicaciones. Así mismo, los sistemas existentes pueden actuar como limitación para el cambio en las organizaciones.

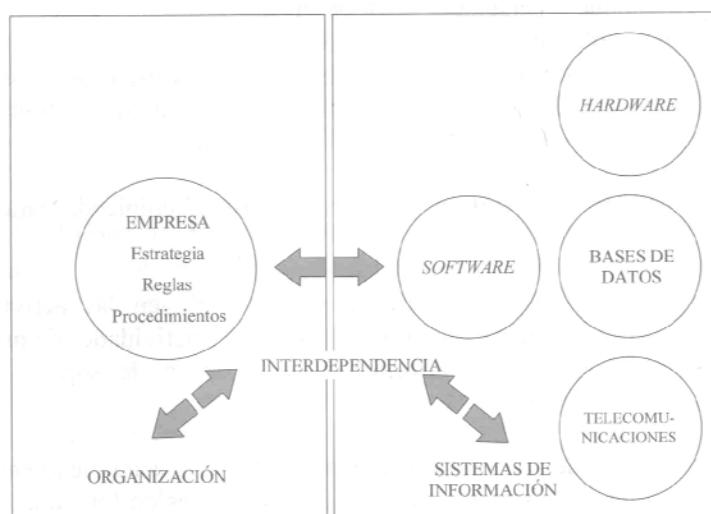


Figura 1.6.Relación entre los componentes de un sistema de información. Fuente: Laudon & Laudon, 1998

Por lo tanto se puede definir a un sistema de Información como un conjunto de elementos interrelacionados (entre los que podemos considerar los distintos medios técnicos, las personas y los procedimientos) cuyo cometido es capturar datos, almacenarlos y

transformarlos de manera adecuada y distribuir la información obtenida mediante todo este proceso.

Su propósito es apoyar y mejorar las operaciones cotidianas de la empresa, así como satisfacer las necesidades de información para la resolución de problemas y la toma de decisiones por parte de los directivos de la empresa.

Por lo tanto, se trata de un sistema que tienen unos inputs (datos) y unos outputs (información), unos procesos de transformación de los inputs en outputs y unos mecanismos de retroalimentación, como se puede apreciar en la siguiente figura:

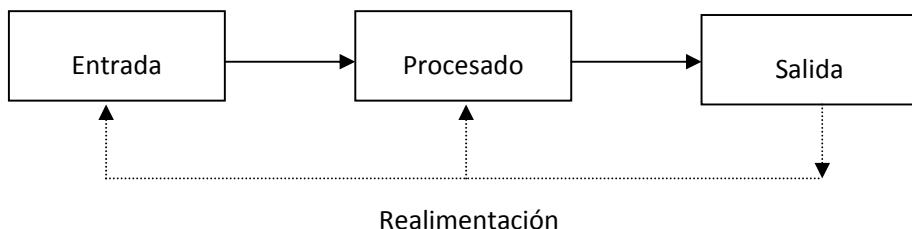


Figura 1.7 Los procesos del Sistema de Información

Es importante incidir en que el verdadero valor de un Sistema de Información no está en la complejidad o en la calidad del propio sistema en sí mismo, sino en la utilización que la organización haga de él, lo que, por regla general, depende más de factores humanos que de factores tecnológicos.

La retroalimentación (feedback) de la información obtenida en todo este proceso se puede utilizar para realizar ajustes y detectar posibles errores en la captura de los datos y/o en su transformación.

Los Sistemas de Información pueden ser manuales o estar informatizados. Hoy en día, lo más normal es que se dé el segundo caso, es decir, que se recurra a un soporte informático (constituido por elementos como el hardware, el software, las bases de datos y los sistemas de telecomunicación) para capturar los datos, procesarlos y presentar la información obtenida.

No obstante, en todo proceso de informatización debe tenerse presente la calidad de los datos y debe buscarse como objetivo principal el aportar un valor a los usuarios a los que se vaya dirigido.

Si no se tiene en cuenta el impacto que puede tener la implantación de la tecnología desde un punto de vista humano y organizativo, lo más probable es que la automatización del sistema fracase debido al rechazo o a la mala utilización de la tecnología por parte de las personas. No conviene olvidar que la tecnología debe ser un medio y no un fin en sí mismo y funciona como herramienta de soporte del sistema.

Por lo tanto, el éxito de una empresa y su diferenciación con respecto al resto de sus componentes no viene dado por la tecnología de la que dispone, sino por el uso que se hace de ella en el seno de la organización.

Debemos tener muy en cuenta que la implantación de las TIC y de los Sistemas de Información dentro de una organización no sólo tiene que resolver problemas de índole tecnológico, sino que también es necesario prestar la suficiente atención a los aspectos organizativos y a la importancia del factor humano. El ser humano depende de los vínculos y de las estructuras sociales, presenta además una resistencia natural al cambio, fruto de su capacidad de adaptación (reducida a medida que se ha ido acomodando en su situación de partida) y de su miedo ante situaciones desconocidas, que podrían provocarle una pérdida de poder o la necesidad de establecer nuevas rutinas y desarrollar nuevas habilidades (Gómez Vieites, 2006a).

Por lo tanto, las nuevas tecnologías no pueden ser implantadas sin una adecuada gestión del cambio organizativo, del rediseño de los procesos y de la formación y motivación de los empleados. No se debe olvidar que una persona hace lo que sabe hacer, lo que está motivado a hacer y lo que su entorno de trabajo le permite hacer. En este sentido, la introducción de nuevas herramientas tecnológicas produce cambios en el entorno de trabajo de una persona, afectando al equilibrio de estos tres factores (el “saber”, el “querer” y el “poder”).

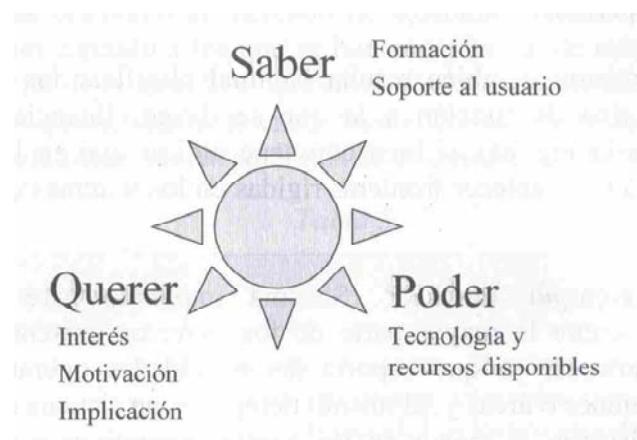


Figura 1.8 El factor humano a la hora de implantar nuevas tecnologías (Gómez Vieites, 2006a)

1.3.2 TIPOS DE SISTEMAS DE INFORMACIÓN

Por lo general, las clasificaciones más extendidas de los Sistemas de Información suelen agrupar éstos en función de su finalidad.

De una forma muy global, puede considerarse que existen dos funciones básicas para los sistemas:

- Soporte a las actividades operativas, que da lugar a sistemas de información para actividades más estructuradas (aplicaciones de contabilidad, nómina, pedidos y, en general, lo que se denomina “gestión empresarial”) o también sistemas que permiten el manejo de información menos estructurada: aplicaciones ofimáticas, programas técnicos para funciones de ingeniería, etc.
- Soporte a las decisiones y el control de gestión, que puede proporcionarse desde las propias aplicaciones de gestión empresarial (mediante salidas de información existentes) o a través de aplicaciones específicas.

Así mismo, también resulta habitual clasificar los sistemas en función del tipo de función a la que se dirige: financiera, recursos humanos, marketing, etc. Si bien conviene indicar que en la actualidad resulta complejo establecer fronteras rígidas en los sistemas que ofrece el mercado.

La literatura de sistemas de información presenta términos como TPS (Transaction Processing Systems) o MIS (Management Information Systems) para reflejar la tipología de sistemas existentes hace ahora varias décadas, que reflejaban con claridad la diferencia entre los sistemas orientados al proceso de transacciones u operaciones y los sistemas orientados a presentar información a los directivos.

Esta clasificación resulta muy difícil de asimilar a la tipología de sistemas actuales. En la siguiente tabla se resumen los sistemas que actualmente encontramos en las empresas (tabla 1.3).

Nombre del Sistema	Descripción
ERP	Enterprise Resource Planning: se trata de los sistemas de gestión integrados que permiten dar soporte a la totalidad de los procesos de una empresa: control económico financiero, logística, producción, mantenimiento, Recursos Humanos, etc.
CRM	Customer Relationship Management: sistemas para gestionar las relaciones con los clientes y el soporte a todos los contactos comerciales
BUSINESS INTELLIGENCE	Sistemas orientados a la explotación de datos y elaboración de información para el soporte a las decisiones
WEB CORPORATIVO Y APLICACIONES DE COMERCIO ELECTRONICO	Conjunto de aplicaciones desplegadas en entorno Web para facilitar la integración de herramientas y contenidos tanto a nivel interno (Intranet) como el despliegue de aplicaciones de comercio electrónico (e-business) y la aplicación de contenidos públicos en la red.

OTRAS APLICACIONES	Se presentarán distintos tipos de aplicaciones como los sistemas PLM (Product Lifecycle Management), herramientas de diseño asistido (CAD), sistemas de gestión documental, herramientas ofimáticas, herramientas de comunicación, sistemas GIS o sistemas de gestión de procesos BPM.
---------------------------	--

Tabla 1.3 Tipos de Sistemas de Información

Es importante recordar en que no existe una correspondencia directa entre los sistemas anteriores y ámbitos funcionales o niveles organizativos en una empresa. De este modo, si se considera el nivel directivo, los sistemas más habituales serían los siguientes:

- La dirección general o la dirección departamental tiene normalmente acceso a los sistemas operacionales:
 - Para consultar datos concretos
 - Como autorizadores o controladores de procesos
 - Para actualizar condiciones que afectan a la operativa (márgenes, límites de crédito, descuentos, etc.)
 - Para acceder a consultas o listados que proporcionan estos sistemas operacionales.
- Intranet, en caso de existir.
- BMP, en casos de existir, y en el rol que le corresponda.
- Sistemas orientados específicamente a la dirección para facilitar el análisis de información y la toma de decisiones (Herramientas Business Intelligence)
- Ofimática, herramientas de correo y, es especial, hojas de cálculo como Excel, que es la herramientas que realmente permite manejar información al directivo en la mayor parte de las empresas.

1.4ANÁLISIS DE REQUISITOS

La comprensión de los requisitos de un problema está entre las tareas más difíciles que enfrenta un ingeniero de software. Cuando se piensa por primera vez acerca de ello, la ingeniería de requisitos no parece tan difícil.

Después de todo, ¿el cliente no sabe lo que se requiere? ¿Los usuarios finales no deberían entender bien las características y funciones que les proporcionarán un beneficio? Es sorprendente, pero en muchas ocasiones la respuesta a estas preguntas es: "no". Y aun si los clientes y usuarios finales son explícitos en sus necesidades, estos requisitos pueden cambiar durante el proyecto. La ingeniería de requisitos es difícil.

Los requerimientos para un sistema son la descripción de los servicios proporcionados por el sistema y sus restricciones operativas. Estos requerimientos reflejan las

necesidades de los clientes de un sistema que ayude a resolver algún problema como el control de un dispositivo, hacer un pedido o encontrar información. El proceso de descubrir, analizar, documentar y verificar estos servicios y restricciones se denomina ingeniería de requerimientos

CLASIFICACIÓN DE LOS REQUERIMIENTOS

Los requerimientos del usuario son declaraciones, en lenguaje natural y en diagramas, de los servicios que se espera que el sistema proporcione y de las restricciones bajo las cuales debe de funcionar.

Se pueden clasificar en requerimientos funcionales y no funcionales.

1. **Requerimientos funcionales.** Son declaraciones de los servicios que debe proporcionar el sistema, de la manera en que éste debe reaccionar a entradas particulares y de cómo se debe comportar en situaciones particulares. En algunos casos, los requerimientos funcionales de los sistemas también pueden declarar explícitamente lo que el sistema no debe de hacer.
2. **Requerimientos no funcionales.** Son restricciones de los servicios o funciones ofrecidos por el sistema. Incluyen restricciones de tiempo, sobre el proceso de desarrollo y estándares. Los requerimientos no funcionales a menudo se aplican al sistema en su totalidad. Normalmente apenas se aplican a características o servicios individuales del sistema. Tipos de requerimientos no funcionales: del producto, organizacionales, externos.

1.4.1 IMPORTANCIA DEL ANÁLISIS DE REQUISITOS.

La meta del proceso de ingeniería de requerimientos es crear y mantener un documento de requerimientos del sistema. El proceso general corresponde cuatro subprocessos de alto nivel de la ingeniería de requerimientos. Éstos tratan de la evaluación de si el sistema es útil para el negocio (estudio de viabilidad); el descubrimiento de requerimientos (obtención y análisis); la transformación de estos requerimientos en formularios estándar (especificación), y la verificación de que los requerimientos realmente definen el sistema que quiere el cliente (validación). La figura 1.9 ilustra la relación entre estas actividades. También muestra el documento que se elabora en cada etapa del proceso de ingeniería de requerimientos.

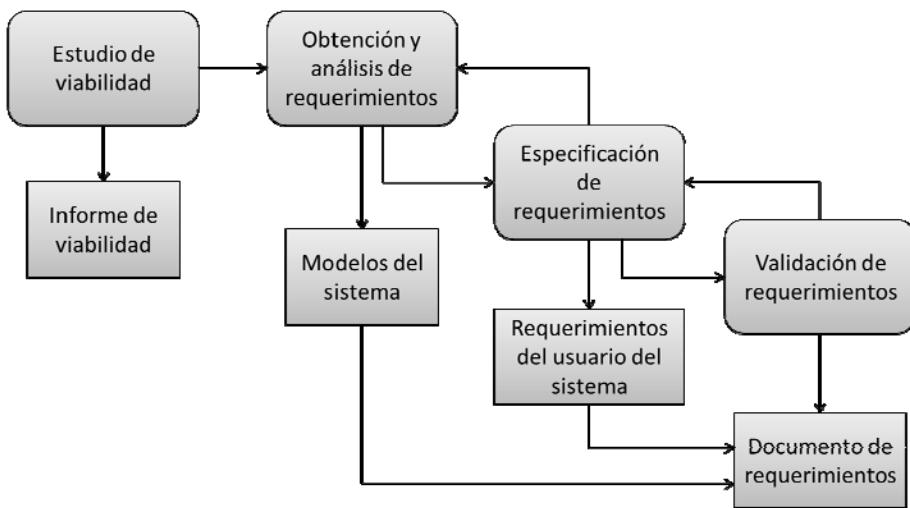


Figura 2.9 El proceso de ingeniería de requerimientos.

Las actividades que se muestran en la figura 1.9 se refieren al descubrimiento, documentación y verificación de requerimientos. Sin embargo, en casi todos los sistemas los requerimientos cambian. Las personas involucradas desarrollan una mejor comprensión de lo que quieren que haga el software, la organización que compra el sistema cambia, se hacen modificaciones a los sistemas hardware, software y al entorno organizacional. El proceso de gestionar estos cambios en los requerimientos se denomina gestión de requerimientos.

Se presenta una perspectiva alternativa sobre el proceso de ingeniería de requerimientos en la figura 1.10. Ésta muestra el proceso como una actividad de tres etapas donde las actividades se organizan como un proceso iterativo alrededor de una espiral. La cantidad de dinero y esfuerzo dedicados a cada actividad en una iteración depende de la etapa del proceso general y del tipo de sistema desarrollado. Al principio del proceso, se dedicará la mayor parte del esfuerzo a la comprensión del negocio de alto nivel y los requerimientos no funcionales y del usuario. Al final del proceso, en el anillo exterior de la espiral, se dedicará un mayor esfuerzo a la ingeniería de requerimientos del sistema y al modelado de éste.

Este modelo en espiral satisface enfoques de desarrollo en los cuales los requerimientos se desarrollan a diferentes niveles de detalle. El número de iteraciones alrededor de la espiral puede variar, por lo que se puede salir de la espiral después de que se hayan obtenido algunos o todos los requerimientos del usuario. Si la actividad de construcción de prototipos mostrada debajo de la validación de requerimientos se extiende para incluir el desarrollo iterativo, este modelo permite que los requerimientos y la implementación del sistema se desarrollen al mismo tiempo.

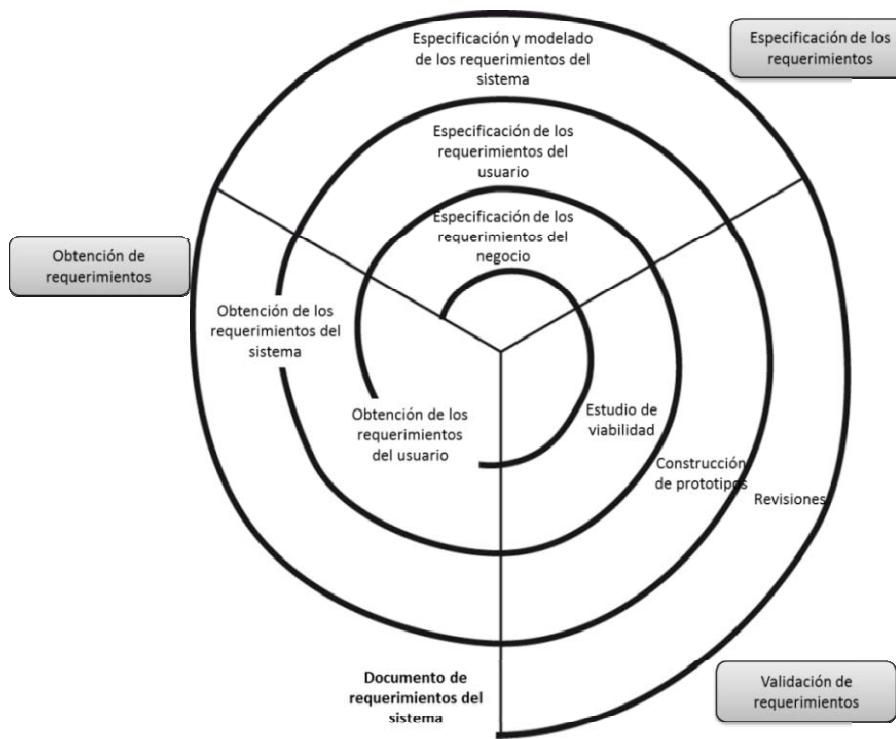


Figura 1.10 Modelo en espiral de los procesos de la ingeniería de requerimientos.

Algunas personas consideran a la ingeniería de requerimientos como el proceso de aplicar un método de análisis estructurado, como el análisis orientado a objetos (Larman, 2002). Éste comprende analizar el sistema y, desarrollar un conjunto de modelos gráficos del mismo, como los modelos de casos de uso, que sirven como una especificación del sistema. El conjunto de modelos describe el comportamiento del sistema al cual se le agregan notas con información adicional que detallan, por ejemplo, el rendimiento o fiabilidad requeridos.

1.4.2 OBTENCIÓN DE LOS REQUISITOS.

La obtención de requerimientos, en particular, es una actividad centrada en las personas, y, a éstas no les gustan las restricciones impuestas por modelos de sistemas rígidos.

Para todos los sistemas nuevos, el proceso de ingeniería de requerimientos debería empezar con un estudio de viabilidad. La entrada de éste es un conjunto de requerimientos de negocio preliminares, una descripción resumida del sistema y de cómo éste pretende contribuir a los procesos del negocio. Los resultados del estudio de viabilidad deberían ser un informe que recomiende si merece o no la pena seguir con la ingeniería de requerimientos y el proceso de desarrollo del sistema.

Un estudio de viabilidad es un estudio corto y orientado a resolver varias cuestiones:

1. ¿Contribuye el sistema a los objetivos generales de la organización?
2. ¿Se puede implementar el sistema utilizando la tecnología actual y dentro de las restricciones de coste y tiempo?
3. ¿Puede integrarse el sistema con otros sistemas existentes en la organización?

La cuestión de si el sistema contribuye a los objetivos del negocio es crítica. Sí no contribuye a estos objetivos, entonces no tiene un valor real en el negocio. Aunque esto pueda parecer obvio, muchas organizaciones desarrollan sistemas que no contribuyen a sus objetivos porque no tienen una clara declaración de estos objetivos, porque no consiguen definir los requerimientos del negocio para el sistema o porque otros factores políticos u organizacionales influyen en la creación del sistema.

Llevar a cabo un estudio de viabilidad comprende la evaluación y recopilación de la información, y la redacción de informes. La fase de evaluación de la información identifica la información requerida para contestar las tres preguntas anteriores. Una vez que dicha información se ha identificado, se debería hablar con las fuentes de información para descubrir las respuestas a estas preguntas. Algunos ejemplos de preguntas posibles son:

1. ¿Cómo se las arreglaría la organización si no se implementara este sistema?
2. ¿Cuáles son los problemas con los procesos actuales y cómo ayudaría un sistema nuevo a aliviarlos'?
3. ¿Cuál es la contribución directa que hará el sistema a los objetivos y requerimientos del negocio?
4. ¿La información se puede obtener y transferir a otros sistemas de la organización?
5. ¿Requiere el sistema tecnología que no se ha utilizado previamente en la organización?
6. ¿A qué debe ayudar el sistema y a qué no necesita ayudar?

En un estudio de viabilidad, se pueden consultar las fuentes de información, como los jefes de los departamentos donde se utilizará el sistema, los ingenieros de software que están familiarizados con el tipo de sistema propuesto, los expertos en tecnología y los usuarios finales del sistema. Normalmente, se debería intentar completar un estudio de viabilidad en dos o tres semanas.

Una vez que se tiene la información, se redacta el informe del estudio de viabilidad. Debería hacerse una recomendación sobre si debe continuar o no el desarrollo del sistema. En el informe, se pueden proponer cambios en el alcance, el presupuesto y la confección de agendas del sistema y sugerir requerimientos adicionales de alto nivel para éste.

La siguiente etapa del proceso de ingeniería de requerimientos es la obtención y análisis de requerimientos. En esta actividad, los ingenieros de software trabajan con los clientes y los usuarios finales del sistema para determinar el dominio de la aplicación, qué servicios debe proporcionar el sistema, el rendimiento requerido del sistema, las restricciones hardware, etcétera.

La obtención y análisis de requerimientos pueden afectar a varias personas de la organización. El término *stakeholder* se utiliza para referirse a cualquier persona o grupo que se verá afectado por el sistema, directa o indirectamente. Entre los *stakeholders* se encuentran los usuarios finales que interactúan con el sistema y todos aquellos en la organización que se pueden ver afectados por su instalación. Otros *stakeholders* del sistema pueden ser los ingenieros que desarrollan o dan mantenimiento a otros sistemas relacionados, los gerentes del negocio, los expertos en el dominio del sistema y los representantes de los trabajadores.

Obtener y comprender los requerimientos de los stakeholders es difícil por varias razones:

1. Los stakeholders a menudo no conocen lo que desean obtener del sistema informático excepto en términos muy generales, puede resultarles difícil expresar lo que quieren que haga el sistema o pueden hacer demandas irreales debido a que no conocen el coste de sus peticiones.
2. Los stakeholders expresan los requerimientos con sus propios términos de forma natural y con un conocimiento implícito de su propio trabajo. Los ingenieros de requerimientos, sin experiencia en el dominio del cliente, deben comprender estos requerimientos.
3. Diferentes stakeholders tienen requerimientos distintos, que pueden expresar de varias formas. Los ingenieros de requerimientos tienen que considerar todas las fuentes potenciales de requerimientos y descubrir las concordancias y los conflictos.
4. Los factores políticos pueden influir en los requerimientos del sistema. Por ejemplo, los directivos pueden solicitar requerimientos específicos del sistema que incrementarán su influencia en la organización.
5. El entorno económico y de negocios en el que se lleva a cabo el análisis es dinámico. Inevitablemente, cambia durante el proceso de análisis- Por lo tanto, la importancia de ciertos requerimientos puede cambiar. Pueden emerger nuevos requerimientos de nuevos stakeholders que no habían sido consultados previamente.

Inevitablemente, los stakeholders tienen opiniones diferentes sobre la importancia y prioridad de los requerimientos, y algunas veces estas opiniones están reñidas. Durante el proceso, se deberían organizar frecuentes negociaciones con los stakeholders para que se pueda llegar a acuerdos. Es imposible satisfacer completamente a todos los stakeholders, pero si algún stakeholder piensa que sus opiniones no se han considerado adecuadamente, deliberadamente puede intentar socavar el proceso de ingeniería de requerimientos.

Las actividades del proceso para la obtención y análisis de los requerimientos son:

1. *Descubrimiento de requerimientos.* Es el proceso de interactuar con los stakeholders del sistema para recopilar sus requerimientos. Los requerimientos del

dominio de los stakeholders y la documentación también se descubren durante esta actividad.

2. *Clasificación y organización de requerimientos.* Esta actividad toma la recopilación no estructurada de requerimientos, grupos relacionados de requerimientos y los organiza en grupos coherentes.
3. *Ordenación por prioridades y negociación de requerimientos.* Inevitablemente, cuando existen muchos stakeholders involucrados, los requerimientos entrarán en conflicto. Esta actividad se refiere a ordenar según las prioridades los requerimientos, y a encontrar y resolver los requerimientos en conflicto a través de la negociación.
4. *Documentación de requerimientos.* Se documentan los requerimientos y se entra en la siguiente vuelta de la espiral. Se pueden producir documentos de requerimientos formales o informales.

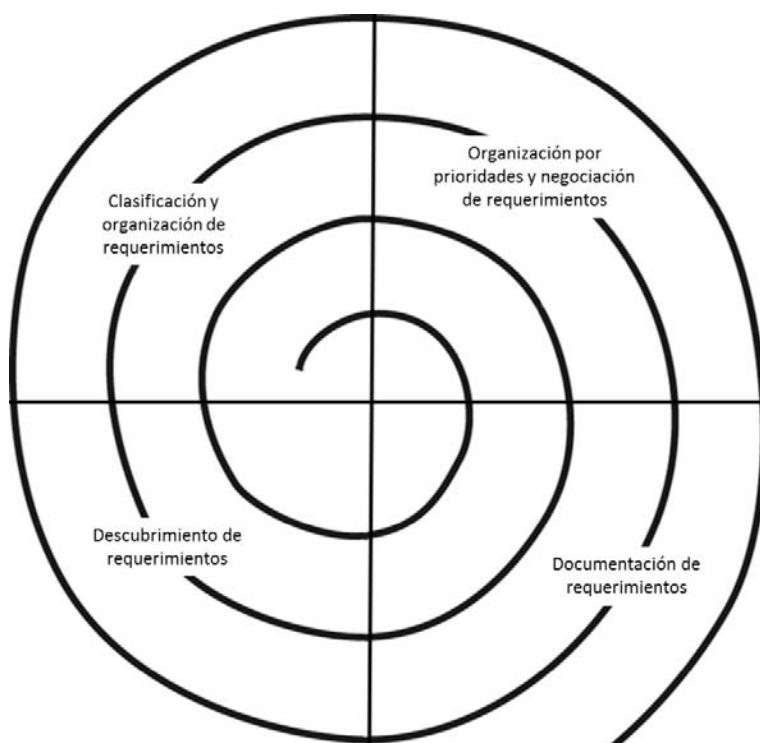


Figura 1.11 El proceso de obtención y análisis de requerimientos.

La figura 1.11 muestra que la obtención y análisis de requerimientos es un proceso iterativo con retroalimentación continua de cada actividad a las otras actividades. El ciclo del proceso comienza con el descubrimiento de requerimientos y termina con la documentación de requerimientos. La comprensión de los requerimientos por parte del analista mejora con cada vuelta del ciclo.

En la etapa de la documentación de requerimientos, los requerimientos que se han obtenido se documentan de tal forma que se pueden utilizar para ayudar al descubrimiento de nuevos requerimientos. En esta etapa, se puede producir una versión inicial del documento de requerimientos del sistema, pero faltarán secciones y habrá requerimientos incompletos. Por otra parte, los requerimientos se pueden documentar como tablas en un documento o en tarjetas, Redactar requerimientos en tarjetas (el enfoque utilizado en la programación extrema) puede ser muy efectivo, ya que son fáciles de manejar, cambiar y organizar para los stakeholders.

1.4.3 INTERPRETACIÓN Y MANEJO DE LOS REQUISITOS.

Los requerimientos para sistemas software grandes son siempre cambiantes. Una razón es que estos sistemas normalmente se desarrollan para abordar problemas «traviesos». Debido a que el problema no puede definirse completamente, es muy probable que los requerimientos del software sean incompletos. Durante el proceso del software, la comprensión del problema por parte de los stakeholders está cambiando constantemente. Estos requerimientos deben entonces evolucionar para reflejar esta perspectiva cambiante del problema.

Además, una vez que un sistema se ha instalado, inevitablemente surgen nuevos requerimientos. Es difícil para los usuarios y clientes del sistema anticipar qué efectos tendrá el Sistema nuevo en la organización. Cuando los usuarios finales tienen experiencia con un sistema, descubren nuevas necesidades y prioridades:

1. Normalmente, los sistemas grandes tienen una comunidad de usuarios diversa donde los usuarios tienen diferentes requerimientos y prioridades. Éstos pueden contradecirse o estar en conflicto. Los requerimientos finales del sistema son inevitablemente un compromiso entre ellos y, con la experiencia, a menudo se descubre que la ayuda suministrada a los diferentes usuarios tiene que cambiarse.
2. Las personas que pagan por el sistema y los usuarios de éste raramente son la misma persona. Los clientes del sistema imponen requerimientos debido a las restricciones organizacionales y de presupuesto. Éstos pueden estar en conflicto con los requerimientos de los usuarios finales y, después de la entrega, pueden tener que añadirse nuevas características de apoyo al usuario si el sistema tiene que cumplir sus objetivos.
3. El entorno de negocios y técnico del sistema cambia después de la instalación, y estos cambios se deben reflejar en el sistema. Se puede introducir nuevo hardware, puede ser necesario que el sistema interactúe con otros sistemas, las prioridades de negocio pueden cambiar con modificaciones consecuentes en la ayuda al sistema, y puede haber una nueva legislación y regulaciones que deben ser implementadas por el sistema.

La gestión de requerimientos es el proceso de comprender y controlar los cambios en los requerimientos del sistema. Es necesario mantenerse al tanto de los requerimientos particulares y mantener vínculos entre los requerimientos dependientes de forma que se pueda evaluar el impacto de los cambios en los requerimientos. Hay que establecer un proceso formal para implementar las propuestas de cambios y vincular éstos a los requerimientos del sistema. El proceso de gestión de requerimientos debería empezar en cuanto esté disponible una versión preliminar del documento de requerimientos, pero se debería empezar a planificar cómo gestionar los requerimientos que cambian durante el proceso de obtención de requerimientos.

1.5 CICLOS DE VIDA.

1.5.1 CASCADA.

El primer modelo de proceso de desarrollo de software que se publicó se derivó de procesos de ingeniería de sistemas más generales (Royce, 1970). Este modelo se muestra en la Figura 1.12. Debido a la cascada de una fase a otra, dicho modelo se conoce como modelo en cascada o como cielo de vida del software. Las principales etapas de este modelo se transforman en actividades fundamentales de desarrollo:

1. **Análisis y definición de requerimientos.** Los servicios, restricciones y metas del sistema se definen a partir de las consultas con los usuarios. Entonces, se definen en detalle y sirven como una especificación del sistema.
2. **Diseño del sistema y del software.** El proceso de diseño del sistema divide los requerimientos en sistemas hardware o software. Establece una arquitectura completa del sistema. El diseño del software identifica y describe las abstracciones fundamentales del sistema software y sus relaciones.
3. **Implementación y prueba de unidades.** Durante esta etapa, el diseño del software se lleva a cabo como un conjunto o unidades de programas. La prueba de unidades implica verificar que cada una cumpla su especificación.
4. **Integración y prueba del sistema.** Los programas o las unidades individuales de programas se integran y prueban como un sistema completo para asegurar que se cumplan los requerimientos del software. Después de las pruebas, el sistema software se entrega al cliente.
5. **Funcionamiento y mantenimiento.** Por lo general (aunque no necesariamente), ésta es la fase más larga del cielo de vida. El sistema se instala y se pone en funcionamiento práctico. El mantenimiento implica corregir errores no descubiertos en las etapas anteriores del cielo de vida, mejorar la implementación de las unidades del sistema y resaltar los servicios del sistema una vez que se descubren nuevos requerimientos.

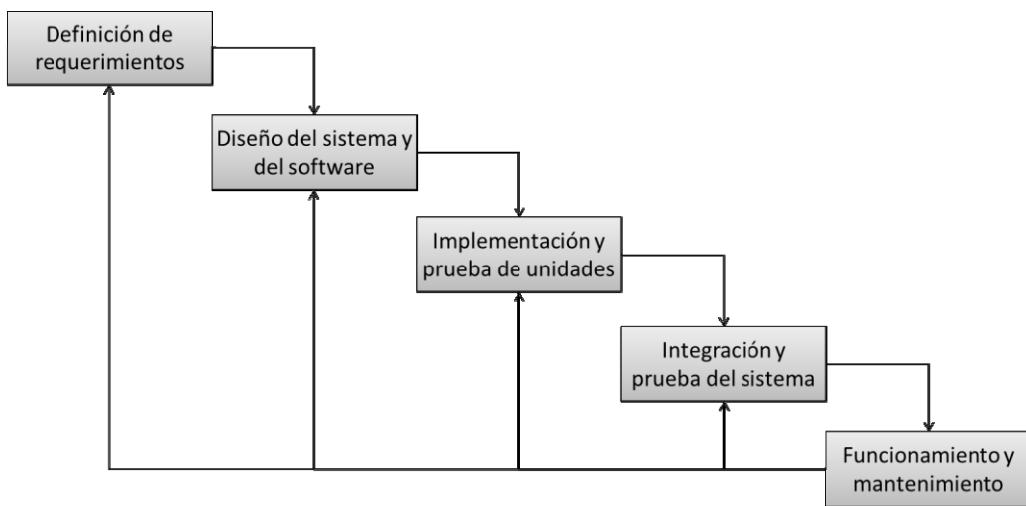


Figura 1.12 El ciclo de vida cascada

En principio, el resultado de cada fase es uno o más documentos aprobados («firmados»). La siguiente fase no debe empezar hasta que la fase previa haya finalizado. En la práctica, estas etapas se superponen y proporcionan información a las otras. Durante el diseño se identifican los problemas con los requerimientos; durante el diseño del código se encuentran problemas, y así sucesivamente. El proceso del software no es un modelo lineal simple, sino que implica una serie de iteraciones de las actividades de desarrollo.

Debido a los costos de producción y aprobación de documentos, las iteraciones son costosas e implican rehacer el trabajo. Por lo tanto, después de un número reducido de iteraciones, es normal cancelar partes del desarrollo, como la especificación, y continuar con las siguientes etapas de desarrollo. Los problemas se posponen para su resolución, se pasan por alto o se programan. Este congelamiento prematuro de requerimientos puede implicar que el sistema no haga lo que los usuarios desean. También puede conducir a sistemas mal estructurados debido a que los problemas de diseño se resuelven mediante trucos de implementación.

Durante la fase final del ciclo de vida (funcionamiento y mantenimiento), el software se pone en funcionamiento. Se descubren errores y omisiones en los requerimientos originales del software. Los errores de programación y de diseño emergen y se identifica la necesidad de una nueva funcionalidad. Por tanto, el sistema debe evolucionar para mantenerse útil. Hacer estos cambios (mantenimiento del software) puede implicar repetir etapas previas del proceso.

Las ventajas del modelo en cascada son que la documentación se produce en cada fase y que éste cuadra con otros modelos del proceso de ingeniería. Su principal problema es su inflexibilidad al dividir el proyecto en distintas etapas. Se deben hacer compromisos en las

etapas iniciales, lo que hace difícil responder a los cambios en los requerimientos del cliente.

Por lo tanto, el modelo en cascada sólo se debe utilizar cuando los requerimientos se comprendan bien y sea improbable que cambien radicalmente durante el desarrollo del sistema. Sin embargo, el modelo refleja el tipo de modelo de proceso usado en otros proyectos de la ingeniería. Por consiguiente, los procesos del software que se basan en este enfoque se siguen utilizando para el desarrollo de software, particularmente cuando éste es parte de proyectos grandes de ingeniería de sistemas.

1.5.2 ESPIRAL

El modelo en espiral del proceso del software (Figura 1.13) fue originalmente propuesto por Boehm (Boehm, 1988). Más que representar el proceso del software como una secuencia de actividades con retrospectiva de una actividad a otra, se representa como una espiral. Cada ciclo en la espiral representa una fase del proceso del software. Así, el ciclo más interno podría referirse a la viabilidad del sistema, el siguiente ciclo a la definición de requerimientos, el siguiente ciclo al diseño del sistema, y así sucesivamente.

Cada ciclo de la espiral se divide en cuatro sectores:

1. Definición de objetivos. Para esta fase del proyecto se definen los objetivos específicos. Se identifican las restricciones del proceso y el producto, y se traza un plan detallado de gestión. Se identifican los riesgos del proyecto. Dependiendo de estos riesgos, se planean estrategias alternativas.
2. Evaluación y reducción de riesgos. Se lleva a cabo un análisis detallado para cada uno de los riesgos del proyecto identificados. Se definen los pasos para reducir dichos riesgo. Por ejemplo, si existe el riesgo de tener requerimientos inapropiados, se puede desarrollar un prototipo del sistema.
3. Desarrollo y validación. Después de la evaluación de riesgos, se elige un modelo para el desarrollo del sistema. Por ejemplo, si los riesgos en la Interfaz de usuario son dominantes, un modelo de desarrollo apropiado podría ser la construcción de prototipos evolutivos. Si los riesgos de seguridad son la principal consideración, un desarrollo basado en transformaciones formales podría ser el más apropiado, y así sucesivamente. El modelo en cascada puede ser el más apropiado para el desarrollo si el mayor riesgo identificado es la integración de los subsistemas.
4. Planificación. El proyecto se revisa y se toma la decisión de si se debe continuar con un ciclo posterior de la espiral. Si se decide continuar, se desarrollan los planes para la siguiente fase del proyecto.

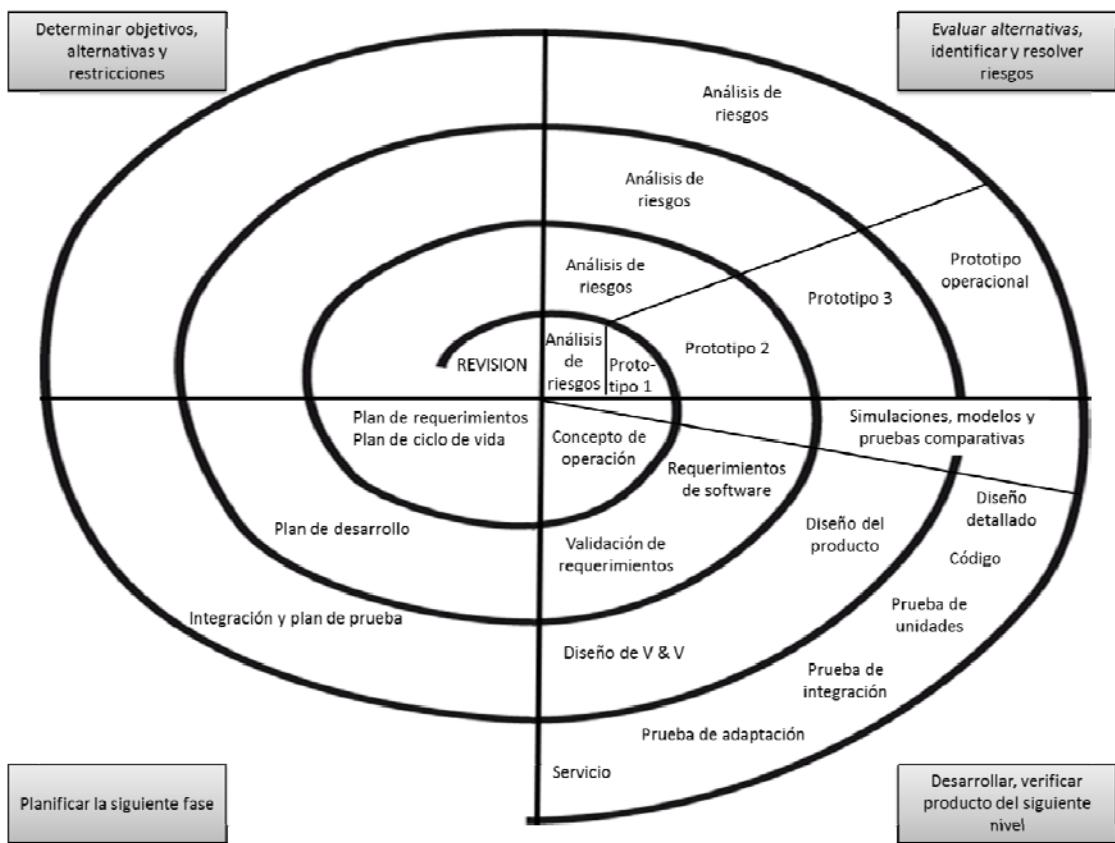


Figura 1.13 Modelo en espiral de Boehm para el proceso de software (IEEE, 1988).

La diferencia principal entre el modelo en espiral y los otros modelos del proceso del software es la consideración explícita del riesgo en el modelo en espiral. Informalmente, el riesgo significa sencillamente algo que puede ir mal. Por ejemplo, si la intención es utilizar un nuevo lenguaje de programación, un riesgo es que los compiladores disponibles sean poco fiables o que no produzcan código objeto suficientemente eficiente. Los riesgos originan problemas en el proyecto, como los de confección de agendas y excesos en los costos, por lo tanto, la disminución de riesgos es una actividad muy importante en la gestión del proyecto.

Un ciclo de la espiral empieza con la elaboración de objetivos, como el rendimiento y la funcionalidad. Entonces se enumeran formas alternativas de alcanzar estos objetivos, y las restricciones impuestas en cada una de ellas. Cada alternativa se evalúa contra cada objetivo y se identifican las fuentes de riesgo del proyecto. El siguiente paso es resolver estos riesgos mediante actividades de recopilación de información como la de detallar más el análisis, la construcción de prototipos y la simulación. Una vez que se han evaluado los riesgos, se lleva a cabo cierto desarrollo, seguido de una actividad de planificación para la siguiente fase del proceso.

1.6 LEGISLACIÓN DEL SOFTWARE.

CASOS DE NORMATIVIDAD APLICADA AL SOFTWARE

AUTORIA Y CREACIÓN DE SOFTWARE

Propiedad intelectual (propiedad industrial y derechos de autor).

La propiedad intelectual es un conjunto de prerrogativas y beneficios que las leyes reconocen y establecen a favor de los autores y de sus causahabientes, por la creación de obras artísticas, científicas, industriales y comerciales.

- Propiedad industrial. Derecho exclusivo que otorga el Estado para explotar en forma industrial y comercial las invenciones o innovaciones de aplicación industrial (patentes, modelos de utilidad y diseños industriales) o indicaciones comerciales (marcas, nombres comerciales y avisos comerciales), que realizan los individuos o las empresas para distinguir sus productos o servicios ante la clientela en el mercado.
- Derechos de autor. Conjunto de prerrogativas que las leyes reconocen y confieren a los creadores de obras intelectuales externadas mediante la escritura, la imprenta, la palabra hablada, la música, el dibujo, la pintura, la escultura, el grabado, el fotocopiado, el cinematógrafo, la radiodifusión, la televisión, el disco, el videocasete y cualquier otro medio de comunicación.

Reserva de derechos. Facultad de usar y explotar en forma exclusiva títulos, nombres, denominaciones, características físicas y psicológicas distintivas.

LEY FEDERAL DE DERECHOS DE AUTOR

En el Artículo 13 de la Ley Federal de Derechos de Autor, se reconoce a los programas de cómputo como obras protegibles dentro del marco que abarca dicha ley. Además, esta Ley cuenta con un capítulo (que abarca los artículos 101 al 114) que especifica las disposiciones oficiales sobre los derechos de autor de un programa computacional y de las bases de datos.

Además, según esta Ley, los derechos patrimoniales de autor están vigentes durante toda la vida del autor más 100 años tras el final del año de la muerte del autor más joven o de la fecha de publicación en caso de los gobiernos federal, estatal o municipal. Existen dos excepciones a esta regla:

- 1.- Las obras que ingresaron al dominio público antes del 23 de julio de 2003
- 2.- Las obras que por su naturaleza, están protegidas por una reserva de derechos.

En general, esto significa obras creadas por alguien fallecido antes del 23 de julio de 1928 (75 años antes).

La legislación mexicana reconoce y protege tres tipos de derechos: derechos patrimoniales, derechos morales y derechos conexos.

1.7METODOLOGÍAS PARA EL DESARROLLO DEL SOFTWARE.

1.7.1DEFINICIÓN, TIPOS DE METODOLOGÍAS Y ELEMENTOS.

Una metodología es definida como un conjunto integrado de métodos, técnicas y herramientas que cubren más de una etapa del ciclo de vida de los sistemas de información.

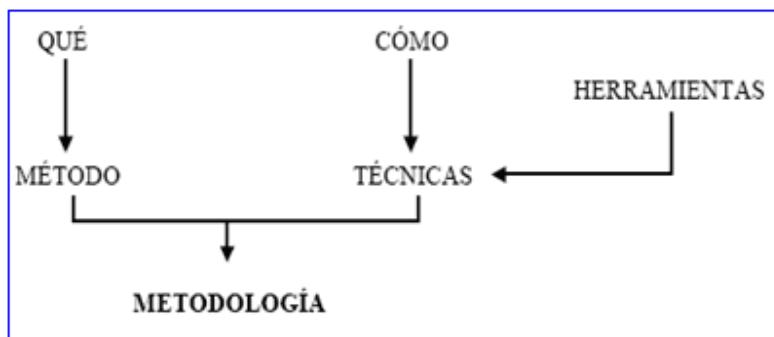


Figura 1.14 Representación del concepto metodología

¿Por qué es importante utilizar una metodología?

- Ofrece un marco y un vocabulario común para el equipo de trabajo.
- Sirve de guía en la utilización de diferentes técnicas y herramientas.
- Ayuda a comprobar la calidad del producto final y al seguimiento de los proyectos.
- Para solucionar los problemas de los sistemas de información de una empresa no es suficiente aplicar soluciones parciales, sino que se precisan enfoques globales.
- Resuelve mucho de los problemas y necesidades actuales existentes en el desarrollo de aplicaciones.

Dentro del campo de la ingeniería del software no existe un modo estándar de tratar el problema de desarrollo y mantenimiento de sistemas de información, sino que existen distintas formas de tratar el problema y por tanto distintos tipos de metodologías asociadas cada una a uno de dichos tipos. Existen diferentes enfoques desde el punto de vista metodológico, producidos principalmente por “como” surgen y evolucionan cada una de las distintas metodologías.

Por tanto, las metodologías de mantenimiento y desarrollo de un sistema de información las podemos clasificar del siguiente modo:

- Metodología clásica.
- Orientadas a objetos.
- Ágiles.

1.7.2 ANÁLISIS DE UNA METODOLOGÍA

De acuerdo a la clasificación de las metodologías planteadas en el subtema anterior a continuación se anexa la tabla 1.4, la cual contiene una comparativa de cada metodología a fin de determinar cuando es factible utilizarlas.

Clasificación de las metodologías	¿Cuándo utilizarla?
La metodología del ciclo de vida del desarrollo de sistemas (SDLC clásica)	<ul style="list-style-type: none"> • Los sistemas se hayan desarrollado y documentado mediante el uso de SDLC • Es importante documentar cada paso del proceso • Haya recursos y el tiempo adecuado para completar el SDLC completo • Sea importante la comunicación en relación con la forma en que funcionan los nuevos sistemas
Metodologías orientadas a objetos	<ul style="list-style-type: none"> • Los problemas modelados se prestan a sí mismos para convertirlos en clases • Una organización ofrece apoyo para aprender UML • Es posible agregar sistemas en forma gradual, un subsistema a la vez • La reutilización de software escrito con anterioridad es una posibilidad • Es aceptable hacer frente a los problemas difíciles primero
Metodologías ágiles	<ul style="list-style-type: none"> • Haya que desarrollar aplicaciones rápidamente en respuesta a un entorno dinámico • Haya que realizar un rescate (el sistema falló y no hay tiempo de averiguar qué salió mal) • El cliente esté satisfecho con las mejoras incrementales • Los ejecutivos y analistas están de acuerdo con los principios de las metodologías ágiles

Tabla 1.4 Análisis de las diferentes metodologías a fin de determinar cuando es conveniente utilizarlas. (Kendall & Kendall)

Las diferencias entre las tres metodologías vistas en la tabla anterior no son tan grandes. En las tres metodologías, el analista necesita comprender primero a la organización. Después el analista o el equipo del proyecto necesitan elaborar un presupuesto del tiempo y los recursos necesarios para desarrollar la propuesta del proyecto. A

continuación se debe entrevistar a los miembros de la organización y recopilar información detallada mediante el uso de cuestionarios, obtener muestras de los datos de los informes existentes y observar cómo se lleva a cabo la actividad empresarial actual. Las tres metodologías tienen todas estas actividades en común.

1.7.3 DESARROLLO A PASOS DE UNA METODOLOGÍA.

Con base a la definición dada de metodología, su importancia y clasificación a continuación se propone un caso de estudio de cómo podría aplicarse la metodología clásica a fin de enmarcar su desarrollo.

Caso de estudio: Se desea realizar un software educativo de apoyo didáctico a los programas de estudio de estudio de los niveles de educación básica y media¹. En este ejemplo lo único que se considera es que pasos se deben incluir para obtener como resultado el diseño del software.

Pasos propuestos para la metodología de desarrollo de un sistema de software:

1. Determinar la necesidad de un sistema educativo.
2. Formación del equipo de trabajo.
3. Análisis y delimitación del tema
4. Definición del usuario.
5. Estructuración del contenido.
6. Elección del tipo de software a desarrollar.
7. Diseño de interfaces.
8. Definición de las estructuras de evaluación.
9. Elección del ambiente de desarrollo.
10. Creación de una versión inicial.
11. Prueba de campo.
12. Mercadotecnia.
13. Entrega del producto final.

La siguiente figura representa brevemente las actividades que conlleva cada paso a fin de mostrar un panorama más amplio del como se lleva una metodología de desarrollo de software.

¹ Para más información consultar el siguiente link
(<http://www.revistaupiicsa.20m.com/Emilia/RevMayDic06/GustavoDESED.pdf>)

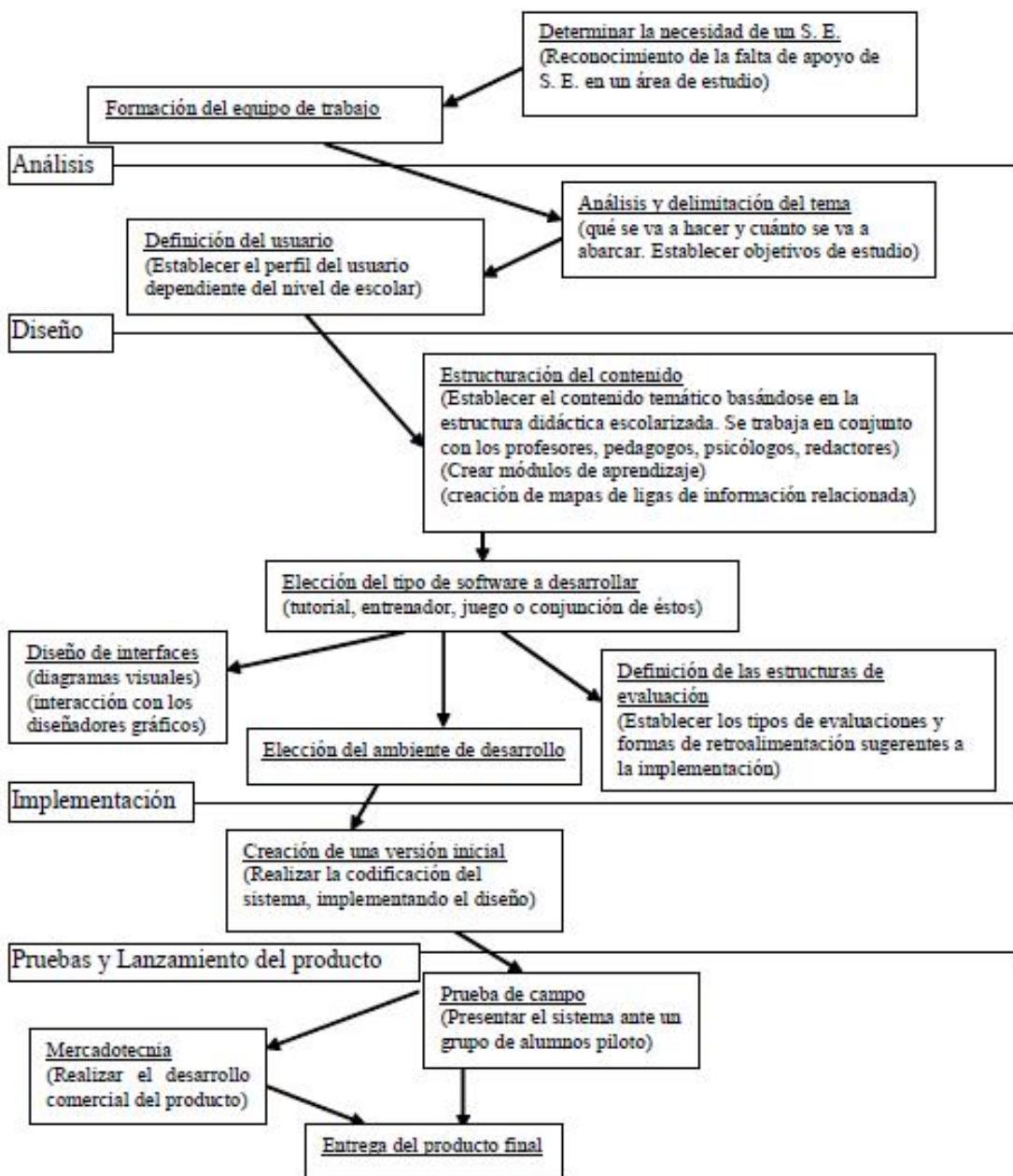


Figura 1.15 Pasos y actividades generales que se involucran en una metodología

UNIDAD 2 FUNDAMENTOS DE PROGRAMACIÓN EN C++

2.1 ESTRUCTURA DEL LENGUAJE

El C es un lenguaje de programación de propósito general. Sus principales características son:

- Programación estructurada y orientada a objetos
- Economía en las expresiones
- Abundancia en operadores y tipos de datos
- Codificación en alto y bajo nivel simultáneamente
- Reemplaza ventajosamente la programación en ensamblador
- Utilización natural de las funciones primitivas del sistema
- No está orientado a ningún área en especial
- Producción de código objeto altamente optimizado

Un programa fuente en C es una colección de cualquier numero de directrices para el compilador, declaraciones, definiciones, expresiones, sentencias y funciones.

Todo programa C debe contener una función nombrada **main**, donde el programa comienza a ejecutarse. Las llaves (<{ }) que incluyen el cuerpo de esta función principal, definen el principio y le final del programa.

ESTRUCTURA GENERAL DE UN PROGRAMA EN C

```
/* Comentarios de un párrafo completo
Comprendidos entre /* ... */, sirven para
Aclarar el programa o una parte del programa */

// Comentarios de una sola línea

// Zona de archivos de cabecera de las librerías
#include <stdio.h>
#include <conio.h>

.....
//Zona de prototipos de funciones
int Potencia (float x);
//Zona de variables globales
int valor;
float media_total;

void main()
{
//Llave de inicio del programa
//código del programa
```

```

.....
.....
.....
//fin del programa
}

//Desarrollo del código de las funciones anteriores

```

2.2 FUNCIONES DE ENTRADA Y SALIDA EN C++: CIN<< Y COUT>>

C++ proporciona una nueva biblioteca de funciones que realizan operaciones de entrada/salida (**E/S**): la biblioteca **iostream**. Esta biblioteca es una implementación orientada a objetos y está basada, al igual que **stdio**, en el concepto de flujos. Cuando se introducen caracteres desde el teclado, puede pensarse en caracteres que fluyen desde el teclado a las estructuras de datos del programa. Cuando se escribe en un archivo, se piensa en un flujo de bytes que van del programa al disco.

Para acceder a la biblioteca **iostream** se debe incluir el archivo **iostream.h**. Este archivo contiene información de diferentes funciones de **E/S**. Define también los objetos **cin** y **cout**.

Las funciones **cin** y **cout** permiten la entrada de datos desde teclado y la impresión en pantalla respectivamente.

La sintaxis de la función **cout** y **cin** queda como sigue:

```

cout<< "Texto a imprimir "; //si lo único que se va a imprimir es texto
cout<<"Texto a imprimir"<<nombre_variable; //si se imprime texto y variables
cin<<nombre_variable; //pide el valor de la variable desde teclado

```

Por ejemplo:

```

cout<< "Introduce un digito: ";
cin >> numero;
cout<< "El digito introducido es: " << numero << '\n';

```

El primer enunciado utiliza el flujo estándar de salida **cout** y el operador **<<** (el operador de inserción de flujo que se pronuncia “colocar en”). El enunciado se lee:

La cadena “Introduce un digito” es colocada en el flujo de salida cout.

2.3 DECLARACIÓN DE VARIABLES Y CONSTANTES

En un bloque en C, todas las declaraciones deben aparecer antes de cualquier enunciado ejecutable.

La sintaxis para declarar una variable o constante es:

tipo_de_dato nombre de variable;

```
int numero1;  
float digito=34;  
float x;
```

En C++, las declaraciones pueden ser colocadas en cualquier parte de un enunciado ejecutable, siempre y cuando las declaraciones anteceden el uso de lo que se está declarando. Por ejemplo:

```
cout << "Ingresa dos digitos: ";  
  
int x, y;  
  
cin >> x >> y;  
  
cout << "La suma de " << x << " y "<< y << " es " << x+y << '\n';
```

declara las variables x y y después del enunciado ejecutable cout, pero antes de que sean utilizadas en el enunciado subsecuente cin.

El alcance de una variable local C++ empieza en su declaración y se extiende hasta la llave derecha de cierre ()).

2.4 OPERADORES

Los operadores son símbolos que indican como son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, lógicos, relacionales, unitarios, lógicos para manejo de bits, de asignación, operador ternario para expresiones condicionales y otros.

2.4.1 OPERADORES DE ASIGNACIÓN

C dispone de varios operadores de asignación para la abreviatura de las expresiones de asignación. Por ejemplo, el enunciado

```
c = c + 3;
```

puede ser abreviado utilizando el operador de asignación += como

```
c += 3;
```

El operador `+=` añade el valor de la expresión, a la derecha del operador, al valor de la variable a la izquierda del operador, y almacena el resultado en la variable a la izquierda del operador. Cualquier enunciado de la forma

```
variable = variable operador expresión;
```

donde operador es uno de los operadores binarios `+`, `-`, `*`, `/` o `%` pueden ser escritos de la forma

```
variable operador=expresión;
```

Por lo tanto, la asignación `c += 3` añade 3 a c. En la tabla 2.1 aparecen los operadores de asignación aritméticos, con expresiones de muestra utilizando estos operadores y con explicaciones.

Si se tienen los siguientes valores:

`int c = 3, d = 5, e = 4, f = 6, g = 12;` la tabla nos muestra los resultados de las operaciones

Operador de asignación	Expresión de muestra	Explicación	Asignación
<code>+=</code>	<code>c += 7</code>	<code>c = c + 7</code>	10
<code>-=</code>	<code>d -= 4</code>	<code>d = d - 4</code>	1
<code>*=</code>	<code>e *= 5</code>	<code>e = e * 5</code>	20
<code>/=</code>	<code>f /= 3</code>	<code>f = f / 7</code>	2
<code>%=</code>	<code>g %= 9</code>	<code>g = g % 9</code>	3

Tabla 2.1 Operadores de asignación aritméticos.

2.4.2 OPERADORES MATEMÁTICOS: UNARIOS Y BINARIOS

La mayor parte de los programas C ejecutan cálculos aritméticos. Los operadores aritméticos de C se resumen en la tabla 2.2. Se advierte el uso de varios símbolos especiales, no utilizados en álgebra. El asterisco (*) indica multiplicación y el signo de por ciento (%) denota el operador modulo.

Operación en C	Operador aritmético	Expresión algebraica	Expresión en C
Suma	<code>+</code>	$f + 7$	<code>f + 7</code>
Substracción	<code>-</code>	$p - c$	<code>p - c</code>
Multiplicación	<code>*</code>	bm	<code>b * m</code>
División	<code>/</code>	$x/y \text{ o } x \div y$	<code>x / y</code>
Módulo	<code>%</code>	$r \bmod s$	<code>r % s</code>

Tabla 2.2 Operadores matemáticos.

2.4.3 OPERADORES LÓGICOS

C proporciona operadores lógicos, que pueden ser utilizados para formar condiciones más complejas, al combinar condiciones simples. Los operadores lógicos son `&&` (el AND lógico), `||` (OR lógico) y `!` (NOT lógico también conocido negación lógica).

Operador lógico		Descripción
AND	<code>&&</code>	Da como resultado el valor lógico 1 si ambos operandos son distintos de 0 el resultado es el valor lógico 0. Si el primer operando es igual a 9, el segundo operando no es evaluado.
OR	<code> </code>	El resultado es 0 si ambos operandos son 0. Si uno de los operandos tiene un valor distinto de 0, el resultado es 1. Si el primer operando es distinto de 0, el segundo no es evaluado.
NOT	<code>!</code>	El resultado es 0 si el operando tiene un valor distinto de 0, y 1 en caso contrario.

Tabla 2.3 Operadores lógicos

2.4.4 OPERADORES RELACIONALES

Los operadores relacionales, también denominados operadores binarios lógicos y de comparación, se utilizan para comprobar la veracidad o falsedad de determinadas propuestas de relación (comparar valores). Las expresiones que los contienen se denominan expresiones relacionales.

Operador de igualdad estándar algebraico u operador relacional	Operador de igualdad o relacional de C	Ejemplo de condición de C	Significado de la condición de C
Operadores de igualdad			
=	<code>==</code>	<code>x == y</code>	x es igual a y
<code>≠</code>	<code>!=</code>	<code>x != y</code>	x no es igual a y
Operadores relacionales			
<code>></code>	<code>></code>	<code>x > y</code>	x es mayor que y
<code><</code>	<code><</code>	<code>x < y</code>	x es menor que y
<code>≥</code>	<code>>=</code>	<code>x >= y</code>	x es mayor que o igual a y
<code>≤</code>	<code><=</code>	<code>x <= y</code>	x es menor que o igual a y

Tabla 2.4 Operadores relacionales.

2.5 SENTENCIAS DE CONTROL

La unidad aritmética y lógica es uno de los componentes más importantes de una computadora. El propósito de la unidad aritmética es el manejo de las operaciones aritméticas; la parte lógica da a la computadora la capacidad de tomar decisiones. Una decisión se especifica en una expresión lógica de la misma forma en que una operación de cálculo se especifica en una expresión numérica.

Las sentencias de control se pueden clasificar en:

- **Secuencia:** Ejecución sucesiva de una o más operaciones
- **Selección:** Se realiza una u otra operación, dependiendo de una condición
- **Iteración:** Repetición de una o varias operaciones mientras se cumpla una condición

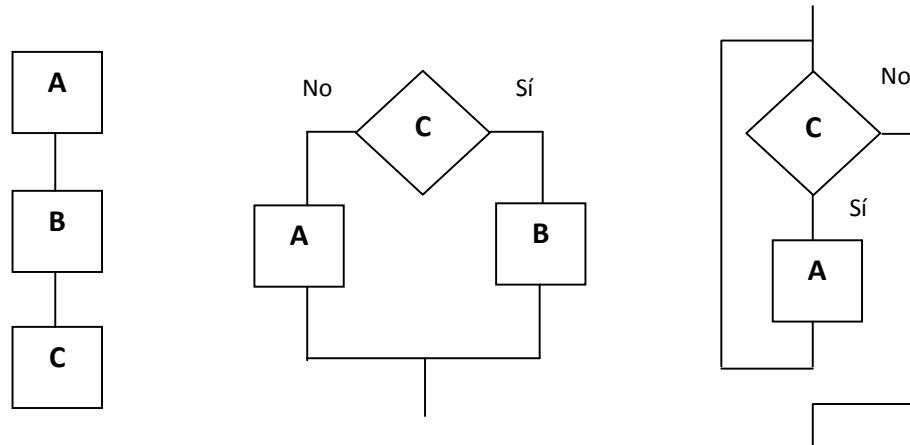


Figura 2.1 Representación de las sentencias de control.

2.5.1 SENTENCIAS CONDICIONALES: IF – ELSE, SWITCH

If

Esta sentencia condicional se utiliza para elegir entre cursos alternativos de acción. Por ejemplo, suponga que en un examen 60 es la calificación de aprobado. El enunciado en pseudocódigo

Si la calificación del estudiante es mayor a 60:
Imprimir “Aprobado”

If / else

La estructura de selección if ejecuta una acción indicada sólo cuando la condición es verdadera; de lo contrario la acción es pasada por alto. La estructura de selección if/else permite que el programador especifique que se ejecuten acciones distintas cuando la condición sea verdadera que cuando la condición sea falsa. Por ejemplo, el enunciado en seudocódigo

```
Si la calificación del estudiante es mayor a 60:  
    Imprimir "Aprobado"  
Sino  
    Imprimir "Reprobado"
```

En cualquiera de los dos casos, después de haber terminado la impresión, se ejecutara el siguiente enunciado del seudocódigo.

El seudocódigo anterior correspondiente a la estructura if/else puede ser escrito en C como

```
if (calificacion >= 60)  
    cout<<"Aprobado\n";  
else  
    cout<<"Reprobado\n";
```

Las sentencias if / else pueden estar anidados, para considerar varias condiciones

Switch

En forma ocasional, un algoritmo contendrá una serie de decisiones, en las cuales una variable o expresión se probará por separado contra cada uno de los valores enteros que se puede asumir, y se tomarán diferentes acciones. Para esta forma de toma de decisiones se proporciona una estructura de selección múltiple switch.

La estructura switch está formada de una serie de etiquetas case, y de un caso opcional default.

Por ejemplo al querer mostrar un menú en el programa para obtener la siguiente salida, es de bastante utilidad esta sentencia:

```
MENU
1. Saludar
2. Realizar una suma
3. Salir
Ingrese la opción deseada:
```

En estos casos es necesario que el usuario introduzca alguna opción (comprendida entre el número 1 y 3), para ello la sentencia switch maneja la siguiente sintaxis:

```
switch (variable)
{
    case 1: //código; break;

    case 2: //código; break;
    ...
    ...
    ...
    default: //código;
}
```

El código correspondiente al ejemplo planteado quedaría de la siguiente forma:

```
Void main( )
{
    Int x;
    Cout<<"Menu\n";
    Cout<<"1.Saludar\n";
    Cout<<"2.Realizar una suma\n";
    Cout<<"3.Salir\n";
    Cout<<"Ingrese la opción deseada\n";
    Cin>>x;

    Switch(x)
    {
        Case 1: cout<<"\n\n Hola a todos";
                  break;
        case 2: cout<<"\n\nIntroduce los numeros a sumar";
                  ....
                  ....
                  Break;
        ....
    }
}
```

```

....  

Default: cout<<"Opción no válida";  

}  

}

```

De esta forma al decir switch (x) la sentencia permitirá que se ejecute el caso correspondiente al valor de la variable x.

2.5.2 SENTENCIAS DE REPETICIÓN: FOR, WHILE, DO – WHILE

For

La estructura de repetición for maneja de manera automática todos los detalles de la repetición controlada por un contador.

La figura 2.2 analiza más de cerca la estructura for. La estructura for, especifica cada uno de los elementos necesarios para la repetición controlada por contador con una variable de control. Si en el cuerpo del for existe más de un enunciado, se requerirán de llaves para definir el cuerpo del ciclo.

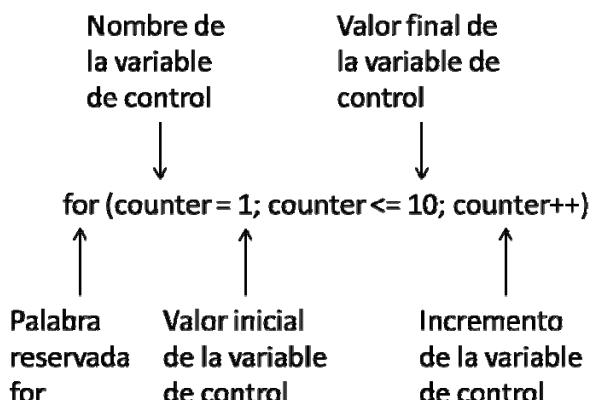


Figura 2.2 Sintaxis de la sentencia for

El formato general de la estructura for es

```

for (expresión1; expresión2; expresión3)  

    enunciado

```

donde expresión1 inicializa la variable de control de ciclo, expresión2 es la condición de continuación de ciclo, y expresión3 incrementa la variable de control.

Las tres expresiones de la estructura for son opcionales. Si se omite expresión2, C supondrá que la condición es verdadera, creando por lo tanto un ciclo infinito. También, se puede omitir la expresión1, si la variable de control se inicializa en alguna otra parte del programa. La expresión3 podría omitirse, si el incremento se calcula mediante enunciado en el cuerpo de la estructura for, o si no se requiere de ningún incremento.

While

Una estructura de repetición le permite al programador especificar que se repita una acción, en tanto cierta condición se mantenga activa.

La sintaxis del while queda de la siguiente forma:

```
While (condición)
{
    //código a ejecutar
}
```

Esto puede leerse como un mientras la condición sea cierta se realiza el siguiente código, una vez que la condición es falsa es cuando el control del programa sale del ciclo y continúan ejecutándose las siguientes líneas de código.

El siguiente fragmento representa el uso del while:

```
producto = 2;
while (producto <= 1000)
{
    i = i++;
    producto = 2 * producto;
}
```

Existe una variante en el ciclo while y esta dada por la incorporación de la palabra do. En el ciclo do – while se ejecuta el código y posteriormente se verifica la condición con lo cual se deduce que en este al menos se entra al ciclo una sola vez en comparación con el ciclo while (el cual primero verifica la condición).

La sintaxis de esta variación esta dada por el siguiente recuadro:

```
do
{
    //código a ejecutar
} While (condición);
```

Tanto el ciclo while como su variante el do – while pueden emular el comportamiento del ciclo for.

2.6 ARREGLOS UNIDIMENSIONALES Y BIDIMENSIONALES

Un arreglo es un grupo de posiciones en memoria relacionadas entre sí, por el hecho de que todas tienen el mismo nombre y son del mismo tipo. Para referirse a una posición en particular o elemento dentro del arreglo, se especifica el nombre del arreglo y el número de posición del elemento particular dentro del mismo.

En la figura 2.3 se muestra un arreglo de enteros llamado c. Este arreglo contiene doce elementos. Cualquiera de estos elementos puede ser referenciado dándole el nombre del arreglo seguido por el número de posición de dicho elemento en particular en paréntesis cuadrados o corchetes ([]). El primer elemento de cualquier arreglo es el elemento cero. Entonces, el primer elemento de un arreglo c se conoce como c[0], el segundo como c[1], el séptimo como c[6] y en general, el elemento de orden i del arreglo c se conoce como c[i-1]. Los nombres de los arreglos siguen las mismas reglas convencionales que los demás nombres de variables.

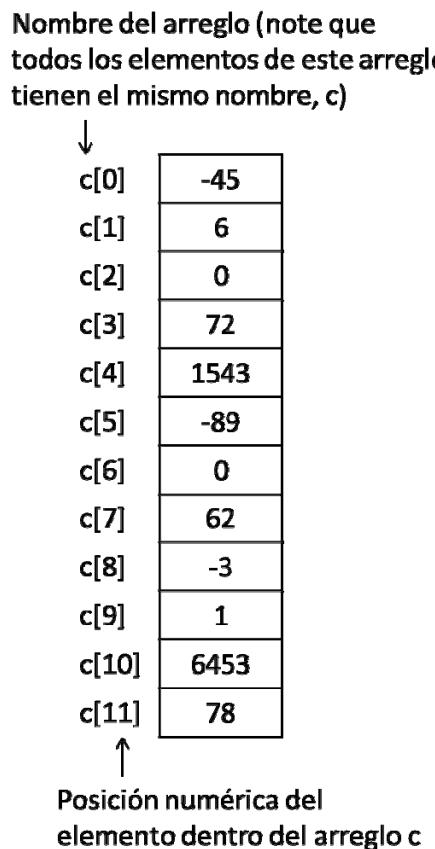


Figura 2.3 Un arreglo de 12 elementos.

El número de posición que aparece dentro de los corchetes se conoce más formalmente como subíndice. Un subíndice debe ser un entero o una expresión entera. Si un programa

utiliza una expresión como subíndice, entonces la expresión se evalúa para determinar el subíndice. Por ejemplo, si $a = 5$ y $b = 6$, entonces el enunciado

```
c[a + b] += 2;
```

añade 2 al elemento del arreglo $c[11]$.

Como declarar arreglos

Los arreglos ocupan espacio en memoria. El programador especifica el tipo de cada elemento y el número de elementos requerido por cada arreglo, de tal forma que la computadora pueda reservar la cantidad apropiada de memoria. Para indicarle a la computadora que reserve 12 elementos para el arreglo entero c , la declaración:

```
int c[12];
```

es utilizada. La memoria puede ser reservada para varios arreglos dentro de una sola declaración. Para reservar 100 elementos para el arreglo entero b y 27 elementos para el arreglo entero x , se utiliza la siguiente declaración:

```
int b[100], x[27];
```

Los arreglos pueden ser declarados para que contengan otros tipos de datos. Por ejemplo, un arreglo del tipo char puede ser utilizado para almacenar una cadena de caracteres.

Cadena de caracteres

Los arreglos de caracteres tienen varias características únicas y van a servir para almacenar frase u oraciones completas. Un arreglo de caracteres puede ser inicializado utilizando una literal de cadena. Por ejemplo, la declaración

```
char frase[] = "first";
```

inicializa los elementos de la cadena frase a los caracteres individuales de la literal de cadena "first". El tamaño del arreglo cadena en la declaración anterior queda determinada por el compilador, basado en la longitud de la cadena. Es importante hacer notar que la cadena "first" contiene cinco caracteres, más un carácter especial de terminación de cadena, conocido como carácter nulo. Entonces, el arreglo frase, de hecho contiene seis elementos. La representación de la constante de caracteres del carácter nulo es '\0'. En C todas las cadenas terminan con este carácter. Un arreglo de caracteres, representando una cadena, debería declararse siempre lo suficiente grande para contener el número de caracteres de la cadena, incluyendo el carácter nulo de terminación.

Los arreglos de caracteres también pueden ser inicializados con constantes individuales de caracteres en una lista de inicialización. La declaración anterior es equivalente a

```
char frase[] = {'f', 'i', 'r', 's', 't', '\0'};
```

Dado que la cadena de hecho es un arreglo de caracteres, podemos tener acceso directo a los caracteres individuales de una cadena, utilizando la notación de subíndices de arreglos. Por ejemplo, frase[0], es el carácter 'f' y frase[3] es el carácter 's'.

Para introducir una cadena de caracteres basta únicamente utilizar las funciones gets() y puts(). A continuación se proporciona un ejemplo de su uso:

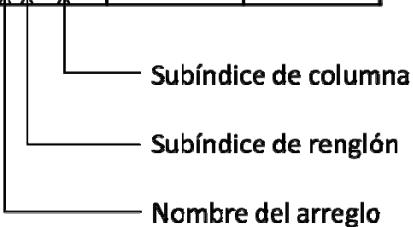
```
char nombre[20];  
.....  
cout << "Introduce tu nombre: ";  
gets(nombre);  
.....  
cout >> Tu te llamas.";  
puts(nombre);
```

Arreglos con múltiples subíndices

Sin embargo en C los arreglos pueden tener múltiples subíndices. Una utilización común de los arreglos con múltiples subíndices es la representación de tablas de valores, consistiendo de información arreglada en renglones y columnas. Para identificar un elemento particular de la tabla, deberemos especificar subíndices, el primero (por regla convencional) identifica el renglón del elemento, y el segundo (también por regla convencional) identifica la columna del elemento. Tablas o arreglos que requieren dos subíndices para identificar un elemento en particular se conocen como arreglos de doble subíndice. Los arreglos de múltiples subíndices pueden tener más de dos subíndices. El estándar ANSI indica que un sistema ANSI C debe soportar por lo menos 12 subíndices de arreglo.

En la figura 2.4 se ilustra un arreglo de doble subíndice, a. El arreglo contiene tres renglones y cuatro columnas, por lo que se dice que se trata de una arreglo de 3 por 4. En general, un arreglo con m renglones y n columnas se llama un arreglo de m por n.

	Columna 0	Columna 1	Columna 2	Columna 3
Renglón 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Renglón 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Renglón 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]



Subíndice de columna
Subíndice de renglón
Nombre del arreglo

Figura 2.4 Un arreglo de doble subíndice con tres renglones y cuatro columnas

Un arreglo de múltiple subíndice puede ser inicializado en su declaración en forma similar a un arreglo de un subíndice. Por ejemplo, un arreglo de doble subíndice b[2][2] podría ser declarado e inicializado con

```
int b[2][2] = {{1, 2}, {3, 4}};
```

Los valores se agrupan por renglones entre llaves. Por lo tanto, 1 y 2 inicializan b[0][0] y b[0][1], 3 y 4 inicializan b[1][0] y b[1][1]. Si para un renglón dado no se proporcionan suficientes inicializadores, los elementos restantes de dicho renglón se inicializarán a 0. Por lo tanto, la declaración

```
int b[2][2] = {{1}, {3, 4}};
```

inicializaría b[0][0] a 1, b[0][1] a 0, b[1][0] a 3 y b[1][1] a 4.

2.6.1 LECTURA Y ESCRITURA EN ARREGLOS

2.6.2 OPERACIONES BÁSICAS CON ARREGLOS

Con los elementos almacenados dentro de un arreglo se pueden realizar cualquier operación, siempre y cuando se respeten los índices del arreglo en donde los elementos están almacenados.

A continuación se muestra un ejemplo de cómo lograr obtener la suma de todos los elementos que integran el arreglo:

```
float arreglo[15], sumatoria;  
int i;  
  
void main()  
{  
....  
for ( i=0; i<15; i++ )  
sumatoria+=arreglo[i];  
.....  
}
```

En el código presentado la variable sumatoria tendrá almacenado al terminar el ciclo for la suma de todos los elementos que integran el arreglo, resaltando la utilización del índice del arreglo.

2.6.3 MÉTODOS DE ORDENAMIENTO

La ordenación de datos (es decir, colocar datos en un orden particular, como orden ascendente o descendente) es una de las aplicaciones más importantes de la computación. Un banco clasifica todos los cheques por número de cuenta, de tal forma que al final de cada mes pueda preparar estados bancarios individuales. Para facilitar la búsqueda de números telefónicos, las empresas telefónicas clasifican sus listas de cuentas por apellido y dentro de ello, por nombre. Virtualmente todas las organizaciones deben clasificar algún dato y en muchos casos, cantidades masivas de información. La ordenación de datos es un problema intrigante que ha atraído alguno de los esfuerzos más intensos de investigación en el campo de la ciencia de la computación.

Una de las técnicas conocidas es la llamada ordenación tipo burbuja u ordenación por hundimiento, debido a que los valores más pequeños de forma gradual flotan hacia la parte superior del arreglo, como suben las burbujas de aire en el agua, en tanto que los valores más grandes se hunden hacia el fondo del arreglo. La técnica consiste en llevar a cabo varias pasadas a través del arreglo. En cada pasada, se comparan pares sucesivos de elementos. Si un par está en orden creciente (o son idénticos sus valores), dejamos los valores tal y como están. Si un par aparece en orden decreciente, sus valores en el arreglo se intercambian de lugar.

Observemos en las siguientes figuras la forma en que funciona el método burbuja:

14	56	8	72	12
----	----	---	----	----

Figura 2.5 Elementos de un arreglo desordenado

A continuación con la figura anterior se aprecia como los elementos aparecen bastante desordenados, razón por la cual se empieza la comparación del primer elemento con el que sigue y así sucesivamente para verificar que se ordenen:

14	56	8	72	12
↑				

14>56?

Una vez que se obtiene la respuesta se procede a realizar el cambio respectivo en caso de existir y sino se continua verificando el ultimo elemento comparado con el siguiente

14	56	8	72	12
↑				

56>8?

aquí se aprecia claramente que 56 si es mayor que el numero 8 razón por la cual procede un intercambio de lugares en el arreglo quedando como sigue:

14	8	56	72	12
----	---	----	----	----

Continuando con las comparaciones tenemos que al realizar la primera pasada el arreglo quedaría ordenado de la siguiente forma:

14	8	56	12	72
----	---	----	----	----

Sin embargo con esa pasada el arreglo continua estando desordenado razón por la cual es necesario volver a ejecutar la comparación de nueva cuenta, el método burbuja semanal que la cantidad de comparaciones completas en el arreglo será en total $n - 1$; donde n es el número de elementos que contiene el arreglo. Continuando con la ejecución del algoritmo tenemos que:

14	56	8	72	12
----	----	---	----	----

14	8	56	12	72
8	14	12	56	72
8	12	14	56	72

Figura 2.6 Los elementos del arreglo ordenados, utilizando el método burbuja

Se puede apreciar que no basta con solo una comparación de elemento con elemento del arreglo, sino que debe de volver a realizarse la comparación de nueva cuenta con los elementos del arreglo. El código que ejemplifica el método burbuja se expone a continuación:

```
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        if (arre[j]<arre[j+1])
        {aux=arre[j];
        arre[j]=arre[j+1];
        arre[j+1]=aux; }
    }
}
```

En donde n equivale al total de elementos contenidos en el arreglo

Cabe resaltar que el método burbuja no es del todo óptimo, puesto que cuando el arreglo este casi ordenado por las condiciones de la búsqueda se debe de seguir ejecutando el algoritmo y esto hace que se desperdicie tiempo de ejecución en la computadora.

2.7 FUNCIONES (MÉTODOS)

Una función es una colección independiente de declaraciones y sentencias, generalmente enfocadas a realizar una tarea específica. Todo programa en C consta al menos de una función, main. Además de ésta, puede haber otras funciones cuya finalidad es fundamentalmente, descomponer el problema general en subproblemas más fáciles de resolver y de mantener. La ejecución de un programa comienza por la función main.

Cuando una función se llama, el control se pasa a la misma para su ejecución y cuando ésta finaliza, el control es devuelto de nuevo al módulo que llamó, para continuar con la ejecución del mismo, a partir de la sentencia que efectuó la llamada. (ver figura 2.7)

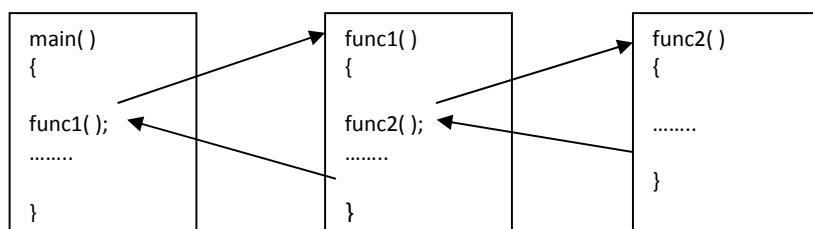


Figura 2.7 Muestra el comportamiento de las funciones

La declaración de una función esta dada por la siguiente sintaxis:

Tipo_valor_devueltoporlafuncion Nombre_funcion (argumentos o parámetros)
--

Ejemplo:

void calculo(); float registro(int a, float b);
--

La definición de una función consta de una cabecera de función y del cuerpo de la función encerrado entre llaves

Tipo_valor_devueltoporlafuncion Nombre_funcion (argumentos o parámetros) { declaraciones; sentencias; [return (expression)]; }

2.7.1 TIPOS DE FUNCIONES

Existen funciones que devuelven valor y otras que no lo hacen. Cuando una función NO devuelve valor se dice que es de tipo *void*. El tipo del resultado especifica que tipos de datos retorna la función. Este puede ser cualquier tipo fundamental, pero no puede ser un arreglo o una función. Por defecto el valor devuelto es *int*. Este valor es devuelto a la sentencia de llamada, por medio de la sentencia *return* (expresión).

Los parámetro formales de una función son las variables que reciben los valores de los argumentos en la llamada a la función.

Todos los argumentos excepto los arreglos son pasados por valor (se copia el argumento y no su dirección). Si se desea alterar los contenidos de las variables pasadas, entonces hay que pasar los argumentos por referencia. Esto es, a la función se pasa la dirección del argumento.

A continuación se muestran dos funciones, una que no devuelve valor y otra que si lo hace:

Void calculo () { cout<<"Funcion que no devuelve valor, es de tipo void"; }
--

```

float registro ( int a, float b)
{
    float resultado;
    resultado = (a*10)/b;
    return resultado;
}

```

La función llamada calculo() no devuelve valor y se aprecia eso al ver que es de tipo void. En este tipo de funciones los cálculos que se realicen deben de ser mostrados ahí mismo. A diferencia de una función que devuelve valor como lo es la función registro(int a, float b) la cual al devolver un valor de tipo flotante es necesario que utilice la palabra return.

La función que devuelve valor debe de ser asignada a una variable dentro del main(), esto para que el valor devuelto no se pierda en la ejecución del programa.

2.7.2 LLAMADO A FUNCIONES

A continuación se presenta un programa que muestra el llamado a funciones:

```

#include<iostream.h>
#include<stdio.h>

void farenheit( float g);
float kelvin (float h);

int main()
{ float grados, conversion;
    cout<<"\nPrograma que calcula grados centigrados a Kelvin y Farenheit";
    cout<<"\nIntroduce los grados a convertir:";
    cin>>grados;
farenheit(grados);
conversion=kelvin(grados);
    cout<<"\n\n Y a grados kelvin equivalen a "<<conversion;
    system("pause");
    return 0;
}

void farenheit (float g)
{
    float faren;

```

```

faren=(32+(g*1.8));
cout<<"\n\n Estos "<<g<< " centigrados equivalen a "<< faren<<" farenheit ";
}

float kelvin (float g)
{
    return (g+273.5);
}

```

Figura 2.8 Programa codificado en Dev C++

En el código de la figura 2.8 muestra como se llaman a las funciones dependiendo el tipo de valor que devuelven (observe las letras en negrita), así mismo se resalta el hecho de como se declaran las funciones y como son declaradas.

En la figura 2.9 se presenta la ejecución del programa.

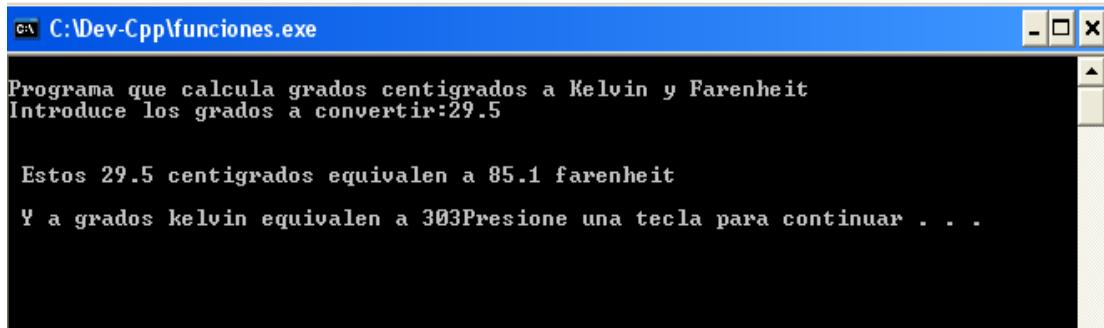


Figura 2.9 Programa en ejecución utilizando funciones.

UNIDAD 3 FUNDAMENTOS DEL PARADIGMA ORIENTADA A OBJETOS

3.1 DEFINICIÓN DEL PARADIGMA ORIENTADO A OBJETOS

La Programación Orientada a Objetos (POO) es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas, miembros de una jerarquía de clases unidad mediante relaciones de herencia.

En la definición resaltan tres aspectos importantes: la programación orientada a objetos (1) utiliza objetos, no algoritmos, como sus bloques lógicos de construcción fundamentales; (2) cada objeto es una instancia de alguna clase; y (3) las clases están relacionadas con otras clases por medio de relaciones de herencia. Un programa puede parecer orientado a objetos, pero si falta cualquiera de estos elementos, no es un programa orientado a objetos. Específicamente, la programación sin herencia es explícitamente no orientada a objetos, se denomina programación con tipos abstractos de datos.

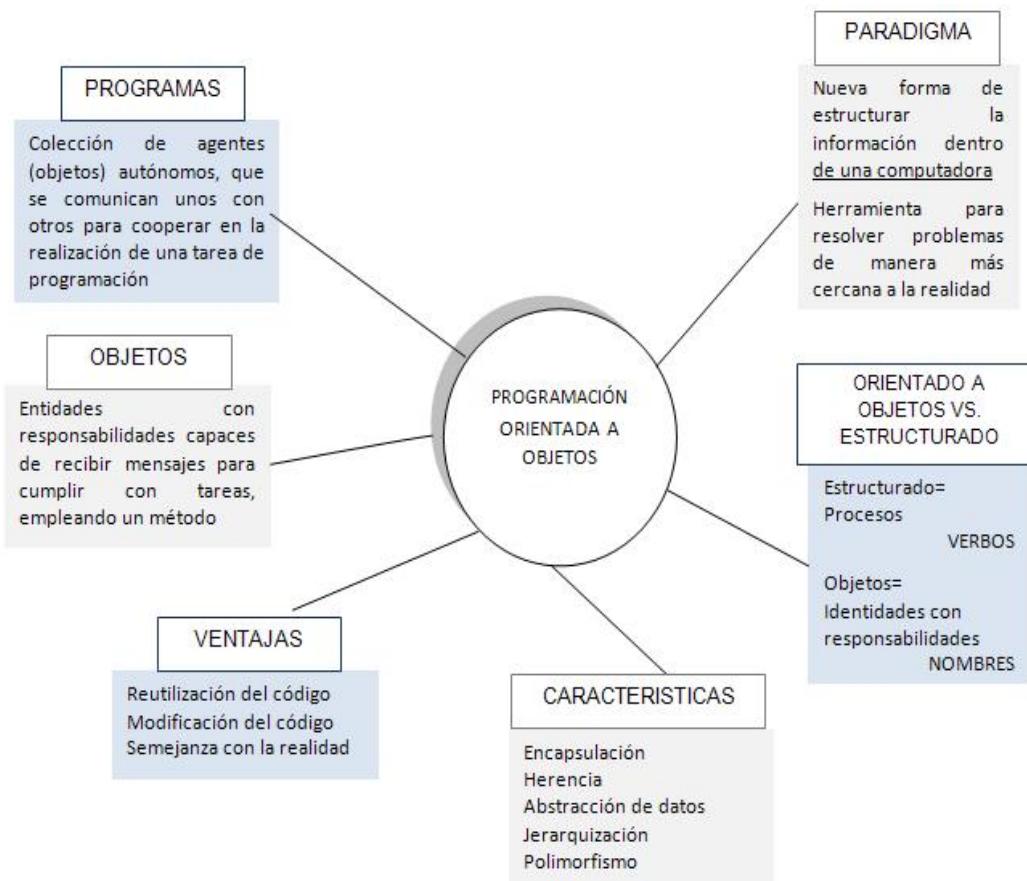


Figura 3.1 Representación del paradigma orientado a objetos

3.1.1 FILOSOFÍA ORIENTADA A OBJETOS.

El concepto o filosofía orientada a objetos consiste en representar un problema por medio de un conjunto de identidades con comportamientos, responsabilidades y características específicas, que interactúan entre sí por medio de mensajes.

Este concepto se puede asociar a sistemas, programas y lenguajes de programación; de tal manera que es posible realizar el análisis o describir la solución de un problema orientado a objetos, para con base en ello, escribir un programa orientado a objetos empleando un lenguaje de programación para el caso.

3.2 DEFINICIÓN DE CLASES Y OBJETOS EN C++

Dentro del paradigma orientado a objetos surgen dos conceptos fundamentales al momento de entenderlo las clases y los objetos, para entender estos conceptos se presenta la definición de ambos:

- *Clase* es la definición abstracta de los elementos que constituyen a un objeto, empleando algún lenguaje de programación orientada a objetos.
- *Objeto* es la variable del tipo de la clase. El objeto existe en tiempo real a partir del momento en que se declara hasta el momento en el que termina el bloque de instrucciones en donde se declaró.

Entonces podemos decir que cada objeto es una instancia de la clase, o que la clase define un nuevo tipo de datos y la variable de ese tipo de datos es el objeto.

3.2.1 ALCANCE DE CLASES EN C++

Los nombres de variables y los nombres de funciones declarados en una definición de clase, y los nombres de datos y funciones miembro de una clase, pertenecen al alcance de dicha clase. Las funciones no miembro se definen en alcance de archivo.

Dentro del alcance de la clase, los miembros de clase son accesibles de inmediato por todas las funciones miembro de dicha clase y pueden ser referenciados sólo por su nombre. Fuera del alcance de una clase, los miembros de clase se referencian, ya sea a través del nombre del objeto, una referencia a un objeto, o un apuntador a un objeto.

Las funciones miembros de una clase pueden ser homónimas, pero sólo por funciones dentro del alcance de dicha clase.

Las funciones miembro tienen alcance de la función dentro de una clase. Si la función miembro define una variable con el mismo nombre que una variable con alcance

de clase, la variable con alcance de clase queda oculta por la variable con alcance de función, dentro del alcance de función. Se puede tener acceso a esta variable oculta mediante el operador de resolución de alcance precediendo el operador con el nombre de la clase. Se puede tener acceso a las variables globales ocultas mediante el operador de resolución de alcance unario.

Los operadores utilizados para tener acceso a los miembros de clase son idénticos a los operadores para tener acceso a los miembros de estructura. El operador de selección de miembro de punto (.) se combina con el nombre de un objeto o con una referencia de objeto para tener acceso a los miembros del objeto.

3.2.2 Atributos de clases en C++

En un sentido amplio, un programa tiene dos tipos de entidades: datos (atributos) e instrucciones (métodos), y que a su vez, los datos pueden ser constantes y variables. Una característica importante de los datos (en especial los variables) es que pueden tener varios atributos, estos atributos o propiedades pueden ser relativos al tiempo de compilación y/o al tiempo de ejecución (runtime).

En comparación con la programación tradicional, un atributo es lo mismo que una variable cualquiera, salvo que como los atributos se declaran para pertenecer a una clase específica, se dice que todos los atributos de dicha clase son miembros de la misma. Por lo demás, la declaración de los atributos es exactamente igual que declarar cualquier otra variable.

Consideremos que el objeto-dato puede tener los siguientes atributos:

- Identificador
- Tipo de dato
- Clase de almacenamiento
- Enlazado
- Dirección
- Valor

Nota: estos atributos no son exclusiva de los objetos; las funciones (algoritmos) también tienen los atributos identificador, tipo, enlazado, dirección y valor (el que devuelven en su caso).

Identificador

El identificador es el nombre con el que el programador reconoce un objeto-dato específico, constante o variable. En la práctica, el identificador puede ser un simple nombre o una expresión que represente únicamente al objeto (un objeto se puede conocer por su dirección, sin que se sepa o preocupe su nombre). Ejemplo:

```
int x = 5;
```

x es el identificador de un miembro de la clase de los enteros de valor 5.

```
char* ptr = "String";
```

ptr es el identificador de un miembro de la clase de los punteros a carácter. Apunta a una constante de cadena de la que no conocemos su nombre, solo su dirección.

Tipo

El tipo (también conocido como tipo de dato) define cuanta memoria es necesaria para almacenar el objeto, como se interpreta el datagrama de bits que hay en el lugar de almacenamiento del objeto; que rango de valores son posibles, y que operaciones les son permitidas a los objetos de dicho tipo.

El concepto tipo de dato tiene una importancia capital en el ámbito de la programación y es la piedra angular sobre la que está construido el propio lenguaje C++. Es pues de suma importancia asimilar íntimamente el significado de este concepto para comprender y utilizar C++ con un mínimo de soltura.

Cada tipo de dato tiene un sentido particular para el compilador (y para el programador). El tipo puede considerarse como un conjunto de propiedades (dependiente a veces de la implementación) que, junto con un conjunto de operaciones que les son permitidas y el rango de valores posibles, es asumido por los miembros de dicho tipo. El compilador C++ deduce este atributo a partir del código; bien de forma implícita, bien mediante declaraciones explícitas.

En C++ existen dos clases de tipos: los definidos intrínsecamente en el lenguaje (tipos preconstruidos en el lenguaje, fundamentales o básicos), y los definidos por el usuario (también llamados abstractos). Precisamente esta capacidad de permitir al usuario "inventar" nuevos tipos a la medida de sus necesidades, es una de las características del lenguaje.

Nota: conviene recordar que C++ dispone de dos operadores específicos que permiten obtener información sobre los tipos en tiempo de ejecución: `sizeof`, que permite averiguar el tamaño en bytes de cualquier tipo estándar o definido por el usuario, y `typeid`, con el que pueden obtenerse algunos datos adicionales.

Clase

La clase de almacenamiento ("storage class"), determina donde se guarda la información; su duración (por cuanto tiempo), que puede ser toda la vida del programa o

solo durante la ejecución de ciertos bloques de código, y algunos aspectos de la visibilidad y el ámbito.

- Situación; es el sitio donde se aloja el dato (constante o variable). Este atributo es significativo porque el programa dispone de varios sitios distintos donde guardar los objetos; sitio que se elige en función de ciertas características del objeto.
- Duración ("Lifetime") es el periodo durante el que la variable tiene existencia real (datos físicamente alojados en memoria). Es un atributo de tiempo de ejecución.
- Ámbito, también llamado campo de acción o ámbito léxico ("Lexical scope") es la parte del programa en que un objeto es conocido por el compilador. Es una propiedad de tiempo de compilación.
- Visibilidad es la región de código fuente desde la que se puede acceder a la información asociada a un objeto.

La clase de almacenamiento puede ser definida de forma explícita o implícita.

Enlazado

Enlazado es el proceso de creación de un programa que sigue a la compilación. Permite que a cada instancia de un identificador sea asociada correctamente con una función u objeto particular. Cada objeto tiene un tipo de enlazado que puede ser de dos clases: estático y dinámico. A su vez, cada objeto tiene además un atributo de enlazado; atributo que está estrechamente relacionado con el ámbito, y que puede ser de tres clases: externo, interno y sin enlazado.

Dirección

La dirección, localización, localizador o *Lvalue* del objeto-dato es una dirección de memoria donde comienza el almacenamiento. Puede ser expresada directamente por un valor (que represente una dirección de memoria en la arquitectura de la computadora utilizada) o una constante, variable, o expresión que pueda traducirse en una dirección.

El *Lvalue* de un objeto puede ser fijo (constante) o variable. Un *Lvalue* modificable significa que la dirección puede ser accedida y su contenido legalmente modificado. Por ejemplo: $x = 4$; es una expresión válida si x es una variable de tipo entero; en su dirección puede ponerse un patrón de bits que significará un valor 4 para el compilador (precisamente porque asume que ahí se aloja un *int*). En otro caso el mismo patrón de bits puede significar algo muy distinto.

Un *Lvalue* constante significa lo contrario, por ejemplo, la expresión $4 = x$ no es correcta porque el *Lvalue* de 4 no es modificable. La expresión $a+b = 4$ tampoco es correcta, porque $a+b$ no tiene *Lvalue* (no es un "objeto", no puede interpretarse como una dirección de memoria).

El *Lvalue* de las variables estáticas se asigna en tiempo de compilación, tienen una dirección fija y conocida desde el principio en una zona de memoria especial reservada para las variables estáticas y globales.

Valor

Valor o *Rvalue* es la interpretación que hace el programa del patrón de bits alojado en la dirección asignada al objeto-dato. Un *Rvalue* es una constante, variable, o expresión que pueda traducirse en un valor. Por ejemplo $4 + 3$ es un *Rvalue*.

Cuando queramos referirnos específicamente al *Rvalue* de un objeto de nombre *x* lo expresaremos: *Rvalue(x)*.

Las variables que no son constantes pueden modificar su *Rvalue* a lo largo del programa. Una expresión del tipo *x = a*; coloquialmente: “*hacer el valor de la variable x igual al valor de la variable (o constante) a*”, puede enunciarse más formalmente: “*copiar el Rvalue de la variable cuyo nemónico es a en la dirección de memoria (Lvalue) de la variable cuyo nemónico es x*”.

3.2.3 Sobrecarga de clases en C++

La sobrecarga es uno de los mecanismos más utilizados del C++ pues reporta una gran cantidad de beneficios a la hora de diseñar las prestaciones de nuestras funciones de miembro. Existen dos tipos fundamentales de sobrecarga: la sobrecarga de funciones y la sobrecarga de operadores.

Sobrecarga de funciones

La sobrecarga de funciones consiste, básicamente, en crear funciones con el mismo nombre dentro de una clase pero con distinto tipo de argumentos de tal forma que, al llamar a la función el compilador se encargará de escoger la adecuada mediante la comparación de la lista de argumentos pasados en la invocación. Un sencillo ejemplo sería pensar en dos funciones llamadas *ImprimirMensaje*. Podemos definirlas así:

```
void ImprimirMensaje(void);
void ImprimirMensaje (char *texto);
```

Como pueden observar, se llaman igual pero tienen argumentos distintos. Mientras que la primera no está preparada para recibir ningún tipo de datos, la segunda sí que lo está y esperará recibir una cadena de caracteres. Si ahora implementamos las funciones de esta forma:

```

void NombreClase::ImprimirMensaje(void)
{
    cout << "\n Esta es la función sin argumentos";
}
void NombreClase::ImprimeMensaje(char *texto)
{
    cout << "\n Esta es la función con argumentos
              y ha recibido el mensaje " << texto;
}

```

Si creáramos una instancia a la clase que contiene a esas dos funciones y llamáramos a la función

```
ImprimeMensaje();
```

Obtendríamos por pantalla:

```
Esta es la función sin argumentos
```

Si, con esa misma instancia, llamáramos a la función

```
ImprimeMensaje("Macedonia Magazine");
```

Lo que obtendríamos sería:

```
Esta es la función con argumentos y ha recibido el mensaje Macedonia Magazine
```

Como pueden observar, es muy sencillo sobrecargar funciones (más que sobrecargar operadores) y pueden ofrecer, sin un coste computacional demasiado elevado, prestaciones excelentes para trabajar dada la gran flexibilidad que proporcionan al programador. He aquí el ejemplo codificado formalmente:

```

#include <iostream.h>
class SobreCargaFunciones
{
public:
void ImprimeMensaje(void);
void Imprime Mensaje(char* texto);
};

void SobreCargaFunciones::ImprimirMensaje(void)
{
    cout << "\n Esta es la función sin argumentos";
}

```

```

void SobrecargaFunciones::ImprimeMensaje(char *texto)
{
    cout << "\n Esta es la función con argumentos
              y ha recibido el mensaje " << texto;
}

int main(void)
{
    SobrecargaFunciones sobrecarga;
    sobrecarga.ImprimeMensaje();
    sobrecarga.ImprimeMensaje("Macedonia Magazine");
}

```

Sobrecarga y constructores

La sobrecarga de constructores se presenta como una de las aplicaciones más directas y comunes de aplicación de este mecanismo del C++, es más, la sobrecarga de constructores es algo que está "a la orden del día" para cualquier programador de C++ pues permite inicializar las instancias de muy distintas formas y dotar al objeto de unos valores iniciales acordes a la finalidad a la que vamos a llevar dicho objeto. Es muy común, pues, ver clases con muchos constructores. Por ejemplo:

```

class Personaje
{
public:
    Personaje(char *nombre);
    Personaje();
}

```

Aquí, podríamos decidir inicializar a la instancia con un nombre que el usuario ha introducido por teclado o bien, crear nosotros la instancia *Personaje* con un nombre por defecto llamando al constructor sin ningún tipo de argumentos.

Sobrecarga de operadores

La sobrecarga de operadores no es un mecanismo que los programadores de C++ recién iniciados se aventuren a utilizar ya que tardan un tiempo en adaptarse, pero es masivamente utilizado en cualquier biblioteca de C++ como la *MFC* de *Microsoft*.

Pero, ¿en qué consiste una sobrecarga de operadores? Sobrecargar operadores sirve para que las operaciones tales como la suma (+), resta (-), multiplicación (*), asignación (=), incremento (++) etc., se comporten de forma diferente al trabajar con nuestros objetos, más exactamente, con los objetos preparados para soportar la sobrecarga de operadores. Imaginar que tenemos dos objetos de tipo Cadena. La clase Cadena está preparada para manejar una cadena de caracteres y darla funcionalidad a través de diversos métodos. Si nosotros quisiéramos sumar dos objetos de tipo Cadena, bastaría con tener sobrecargado el operador suma (+) para los objetos de dicha clase con lo que la siguiente sentencia sería válida:

```
Cadena cadena1, cadena2, cadenaResult;  
cadenaResult = cadena1 + cadena2;
```

Con esto podríamos conseguir, por ejemplo, que el objeto *cadenaResult* almacenara la concatenación de las cadenas de caracteres que soportan los objetos *cadena1* y *cadena2* respectivamente.

Cómo implementar la sobrecarga de operadores

Para sobrecargar un operador, debemos de definir una función de sobrecarga de operador en la clase que nos interese (aunque también la podemos hacer de carácter global). Un operador que se sobrecarga se define siempre con el nombre de la clase seguido de la palabra clave *operator* y del operador. Después del operador pondremos, entre paréntesis, los tipos de datos con los que queremos que funcione nuestro operador. Por ejemplo, si quisiéramos sobrecargar el operador suma de la clase *Cadena*, deberíamos de incluir, en la definición de la clase, la siguiente definición:

```
Cadena operator+(Cadena);
```

Con esto estaríamos declarando que estamos sobrecargando el operador suma (+) para que sea capaz de trabajar con argumentos (sumandos) que sean instancias a la clase *Cadena*. Obviamente, para que esto funcione en la parte izquierda del operador asignación debe de haber otro objeto de tipo de *Cadena*, es decir, no podemos hacer esto:

```
int valor;  
valor = cadena1 + cadena2;
```

Recordemos que estamos sobrecargando el operador suma para que trabaje con objetos que sean instancias de la clase *Cadena*.

3.3. DEFINICIÓN Y CREACIÓN DE MÉTODOS

Un método es un conjunto de instrucciones a las que se les da un determinado nombre de tal manera que sea posible ejecutarlas en cualquier momento sin tenerlas que escribir sino usando sólo su nombre. A estas instrucciones se les denomina cuerpo del método, y a su ejecución a través de su nombre se le denomina llamada al método.

La ejecución de las instrucciones de un método puede producir como resultado un objeto de cualquier tipo. A este objeto se le llama valor de retorno del método y es completamente opcional, pudiéndose escribir métodos que no devuelvan ninguno.

La ejecución de las instrucciones de un método puede depender del valor de unas variables especiales denominadas parámetros del método, de manera que en función del valor que se dé a estas variables en cada llamada la ejecución del método se pueda realizar de una u otra forma y podrá producir uno u otro valor de retorno.

Al conjunto formado por el nombre de un método y el número y tipo de sus parámetros se le conoce como firma del método. La firma de un método es lo que verdaderamente lo identifica, de modo que es posible definir en un mismo tipo varios métodos con idéntico nombre siempre y cuando tengan distintos parámetros. Cuando esto ocurre se dice que el método que tiene ese nombre está sobrecargado.

Para definir un método hay que indicar tanto cuáles son las instrucciones que forman su cuerpo como cuál es el nombre que se le dará, cuál es el tipo de objeto que puede devolver y cuáles son los parámetros que puede tomar. Esto se indica definiéndolo así:

```
<tipoRetorno> <nombreMétodo>(<parámetros>
{
    <cuerpo>
}
```

En *<tipoRetorno>* se indica cuál es el tipo de dato del objeto que el método devuelve, y si no devuelve ninguno se ha de escribir *void* en su lugar.

3.3.1 CONSTRUCTORES EN C++

Los constructores son funciones miembro especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara. Los constructores son especiales por varios motivos:

- Tienen el mismo nombre que la clase a la que pertenecen.
- No tienen tipo de retorno, y por lo tanto no retornan ningún valor.

- No pueden ser heredados.

Por último, deben ser públicos, no tendría ningún sentido declarar un constructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.

Sintaxis:

```
class <identificador de clase> {
    public:
        <identificador de clase>(<lista de parámetros>) [:<lista de constructores>]
        {
            < código del constructor >
        }
        ...
}
```

Si no definimos un constructor el compilador creará uno por defecto, sin parámetros, que no hará absolutamente nada. Los datos miembros del los objetos declarados en el programa contendrán basura.

Si una clase posee constructor, será llamado siempre que se declare un objeto de esa clase. Si ese constructor requiere argumentos, como en este caso, es obligatorio suministrárselos.

Cuando no especifiquemos un constructor para una clase, el compilador crea uno por defecto sin argumentos. Cuando se crean objetos locales, los datos miembros no se inicializarían, contendrían la "basura" que hubiese en la memoria asignada al objeto. Si se trata de objetos globales, los datos miembros se inicializan a cero.

3.3.2 SOBRECARGA DE METODOS EN C++

La sobrecarga (*overload*) de métodos (*funciones*) consiste en declarar y definir varias funciones distintas que tienen un mismo nombre. Dichas funciones se definen de forma diferente. En el momento de la ejecución se llama a una u otra función dependiendo del número y/o tipo de los argumentos actuales de la llamada a la función. Por ejemplo, se pueden definir varias funciones para calcular el valor absoluto de una variable, todas con el mismo nombre *abs()*, pero cada una aceptando un tipo de argumento diferente y con un valor de retorno diferente.

La sobrecarga de funciones no admite funciones que difieran sólo en el tipo del valor de retorno, pero con el mismo número y tipo de argumentos. De hecho, el valor de retorno no influye en la determinación de la función que es llamada; sólo influyen el número y tipo de los argumentos. Tampoco se admite que la diferencia sea el que en una

función un argumento se pasa por valor y en otra función ese argumento se pasa por referencia.

A continuación se presenta un ejemplo con dos funciones sobrecargadas, llamadas ambas *string_copy()*, para copiar cadenas de caracteres. Una de ellas tiene dos argumentos y la otra tres. Cada una de ellas llama a una de las funciones estándar del C: *strcpy()* que requiere dos argumentos, y *strncpy()* que requiere tres. El número de argumentos en la llamada determinará la función concreta que vaya a ser ejecutada:

```
// Ejemplo de función sobrecargada
#include <iostream.h>
#include <string.h>

inline void string_copy(char *copia, const char *original)
{
    strcpy(copia, original);
}

inline void string_copy(char *copia, const *original, const int longitud)
{
    strncpy(copia, original, longitud);
}

static char string_a[20], string_b[20];

void main(void)
{
    string_copy(string_a, "Aquello");
    string_copy(string_b, "Esto es una cadena", 4);
    cout << string_b << " y " << string_a;
    // La última sentencia es equivalente a un printf() de C
    // y se explica en un próximo apartado de este manual
}
```

3.3.3 SOBRECARGA DE CONSTRUCTORES EN C++

Ya se ha descrito que C++ no permite crear objetos sin dar un valor inicial apropiado a todas sus variables miembro. Esto se hace por medio de unas funciones llamadas constructores, que se llaman automáticamente siempre que se crea un objeto de una clase.

El nombre del constructor es siempre el nombre de la clase. Los constructores se caracterizan porque se declaran y definen sin valor de retorno, ni siquiera void. C++ utiliza las capacidades de sobrecarga de funciones de para que una clase tenga varios constructores. Ejemplo:

```
class Cuenta {  
    // Variables miembro  
    private:  
        double Saldo; // Saldo Actual de la cuenta  
        double Interes; // Interés aplicado  
    public:  
        // Constructor  
        Cuenta(double unSaldo, double unInteres);  
        // Acciones básicas  
        double GetSaldo()  
        { return Saldo; }  
        double GetInteres()  
        { return Interes; }  
        void SetSaldo(double unSaldo)  
        { Saldo = unSaldo; }  
        void SetInteres(double unInteres)  
        { Interes = unInteres; }  
        void Ingreso(double unaCantidad)  
        { SetSaldo( GetSaldo() + unaCantidad ); }  
};
```

La definición del constructor de la clase *Cuenta* pudiera ser:

```
Cuenta::Cuenta(double unSaldo, doublé unInteres)  
{ //Hace llamadas los otros métodos  
    SetSaldo(unSaldo);  
    SetInteres(unInteres);  
}
```

Como el constructor es una función miembro, tiene acceso directo a las variables miembro privadas. Luego el constructor también podría definirse del siguiente modo:

```
Cuenta:: Cuenta(double unSaldo, doublé unInteres)  
{ //Asigna a los datos el valor de los  
    parametros  
    Saldo = unSaldo;  
    Interes = unInteres;  
}
```

La llamada al constructor se puede hacer explícitamente en la forma:

```
Cuenta c1 = Cuenta(500,10);
```

o bien, de una forma implícita, más abreviada, permitida por C++:

```
Cuenta c1(500, 10);
```

Se llama constructor por defecto a un constructor que no necesita que se le pasen parámetros o argumentos para inicializar las variables miembro de la clase.

El constructor por defecto es necesario si se quiere hacer una declaración en la forma:

```
Cuenta c1;
```

y también cuando se quiere crear un vector de objetos, por ejemplo:

```
Cuenta cuentas[100];
```

Se llama constructor de oficio al constructor por defecto que define automáticamente el compilador si el usuario no define ningún constructor. Si bien todo constructor de oficio es constructor por defecto (ya que no tiene argumentos), lo contrario no es cierto: el programador puede definir constructores por defecto que no son de oficio.

Importante: el compilador sólo crea un constructor de oficio en el caso de que el programador no haya definido ningún constructor.

En caso de que sólo se haya definido un constructor con argumentos y se necesite un constructor por defecto para crear, por ejemplo un vector de objetos, el compilador no crea este constructor por defecto sino que da un mensaje de error.

Un caso particular se produce cuando se crea un objeto inicializándolo a partir de otro objeto de la misma clase. Por ejemplo, C++ permite crear tres objetos c1, c2 y c3 de la siguiente forma:

```
Cuenta c1(1000,8.5);
Cuenta c2 = c1;
Cuenta c3(c1) (ó Cuenta c3=Cuenta(c1));
```

En la primera sentencia se crea un objeto c1 con un saldo de 1000 y un interés del 8.5%. En la segunda se crea un objeto c2 a cuyos atributos se les asignan los mismos valores que tienen en c1. La tercera sentencia es equivalente a la segunda:c3 se inicializa con los valores de c1.

En las sentencias anteriores se han creado tres objetos y por definición se ha tenido que llamar tres veces a un constructor: en la primera sentencia se ha llamado al

constructor con argumentos definido en la clase, pero en la segunda y en la tercera se ha llamado a un constructor especial llamado constructor de copia (*copy constructor*).

Por definición, el *constructor de copia* tiene un único argumento que es una referencia constante a un objeto de la clase. Su declaración sería pues como sigue:

```
Cuenta::Cuenta(const Cuenta& c2){  
    Saldo = c2.Saldo;  
    Interes = c2.Interes;}
```

Las sentencias anteriores de declaración de los objetos c2 y c3 funcionarían correctamente aunque no se haya declarado y definido en la clase Cuenta ningún constructor de copia. Esto es así porque el compilador de C++ proporciona también un constructor de copia de oficio, cuando el programador no lo define.

El constructor de copia de oficio se limita a realizar una copia bit a bit de las variables miembro del objeto original al objeto copia. En este caso, eso es perfectamente correcto y es todo lo que se necesita. Otras veces, esta copia bit a bit no da los resultados esperados.

Entonces el programador debe preparar su propio constructor de copia e incluirlo en la clase como un constructor sobrecargado más. Dos casos muy importantes en los que se utiliza el constructor de copia:

1. Cuando a una función se le pasan objetos como argumentos por valor, y
2. Cuando una función tiene un objeto como valor de retorno.

3.4 ENCAPSULADO

3.4.1 ENCAPSULADO DE CLASES

Ya sabemos que los miembros privados de una clase no son accesibles para funciones y clases exteriores a dicha clase.

Este es uno de los conceptos de POO, el encapsulamiento, que tiene como objetivo hacer que lo que pase en el interior de cada objeto sea inaccesible desde el exterior, y que el comportamiento de otros objetos no pueda influir en él. Cada objeto sólo responde a ciertos mensajes y proporciona determinadas salidas.

Pero, en ciertas ocasiones, necesitaremos tener acceso a determinados miembros de un objeto de una clase desde otros objetos de clases diferentes, pero sin perder ese encapsulamiento para el resto del programa, es decir, manteniendo esos miembros como privados.

C++ proporciona un mecanismo para sortear el sistema de protección.

Declaraciones friend

El modificador *friend* puede aplicarse a clases o funciones para inhibir el sistema de protección.

Las relaciones de "amistad" entre clases son parecidas a las amistades entre personas:

La amistad no puede transferirse, si A es amigo de B, y B es amigo de C, esto no implica que A sea amigo de C. (La famosa frase: "los amigos de mis amigos son mis amigos" es falsa en C++).

La amistad no puede heredarse. Si A es amigo de B, y C es una clase derivada de B, A no es amigo de C.

La amistad no es simétrica. Si A es amigo de B, B no tiene por qué ser amigo de A.

Funciones amigas externas

El caso más sencillo es el de una relación de amistad con una función externa.

```
#include <iostream>
using namespace std;

class A {
public:
    A(int i=0) : a(i) {}
    void Ver()
private:
    int a;
    friend void Ver(A); // "Ver" es amiga de la clase A
};

void Ver(A Xa) {
    // La función Ver puede acceder a miembros privados
    // de la clase A, ya que ha sido declarada "amiga" de A
    cout << Xa.a << endl;
}
```

```

int main() {
    A Na(10);
    Ver(Na); // Ver el valor de Na.a
    Na.Ver(); // Equivalente a la anterior
    return 0;
}

```

La función "Ver", que no pertenece a la clase A puede acceder al miembro privado de A y visualizarlo. Incluso podría modificarlo.

Funciones amigas en otras clases

El siguiente caso es más común, se trata de cuando la función amiga forma parte de otra clase. El proceso es más complejo. Veamos otro ejemplo:

```

#include <iostream>
using namespace std;

class A; // Declaración previa (forward)

class B {
public:
    B(int i=0) : b(i) {}
    void Ver()
    bool EsMayor(A Xa); // Compara b con a
private:
    int b;
};

class A {
public:
    A(int i=0) : a(i) {}
    void Ver()
private:
    // Función amiga tiene acceso
    // a miembros privados de la clase A
    friend bool B::EsMayor(A Xa);
    int a;
};

```

```

bool B::EsMayor(A Xa) {
    return b > Xa.a;
}

int main() {
    A Na(10);
    B Nb(12);
    Na.Ver();
    Nb.Ver();
    if(Nb.EsMayor(Na)) cout << "Nb es mayor que Na" << endl;
    else cout << "Nb no es mayor que Na" << endl;

    return 0;
}

```

Podemos comprobar lo que pasa si eliminas la línea donde se declara "*EsMayor*" como amiga de A.

Es necesario hacer una declaración previa de la clase A (forward) para que pueda referenciarse desde la clase B.

Veremos que estas "amistades" son útiles cuando sobrecarguemos algunos operadores.

El caso más común de amistad se aplica a clases completas. Lo que sigue es un ejemplo de implementación de una lista dinámica mediante el uso de dos clases "amigas".

```

#include <iostream>
using namespace std;

/* Clase para elemento de lista enlazada */
class Elemento {
public:
    Elemento(int t);           /* Constructor */
    int Tipo() tipo;           /* Obtener tipo */
private:
    int tipo;                  /* Datos: */
    Elemento *sig;             /* Tipo */
    friend class Lista;         /* Siguiente elemento */
};                           /* Amistad con lista */

```

```

/* Clase para lista enlazada de números*/
class Lista {
public:
    Lista() : Cabeza(NULL) {}           /* Constructor */
                           /* Lista vacía */
    ~Lista()           /* Destructor */
    void Nuevo(int tipo);             /* Insertar figura */
    Elemento *Primero();            /* Obtener primer elemento */
    Cabeza;
    /* Obtener el siguiente elemento a p */
    Elemento *Siguiente(Elemento *p) {
        if(p) return p->sig; else return p;};
    /* Si p no es NULL */
    /* Averiguar si la lista está vacía */
    bool EstaVacio() Cabeza == NULL;}

private:
    Elemento *Cabeza;      /* Puntero al primer elemento */
    void LiberarLista(); /* Función privada para borrar lista */
};

/* Constructor */
Elemento::Elemento(int t) : tipo(t), sig(NULL) {}
/* Asignar datos desde lista de parámetros */

/* Añadir nuevo elemento al principio de la lista */
void Lista::Nuevo(int tipo) {
    Elemento *p;

    p = new Elemento(tipo); /* Nuevo elemento */
    p->sig = Cabeza;
    Cabeza = p;
}

/* Borra todos los elementos de la lista */
void Lista::LiberarLista() {
    Elemento *p;

    while(Cabeza) {
        p = Cabeza;
        Cabeza = p->sig;
}

```

```

        delete p;
    }
}

int main() {
    Lista miLista;
    Elemento *e;

    // Insertamos varios valores en la lista
    miLista.Nuevo(4);
    miLista.Nuevo(2);
    miLista.Nuevo(1);

    // Y los mostramos en pantalla:
    e = miLista.Primero();
    while(e) {
        cout << e->Tipo() << ",";
        e = miLista.Siguiente(e);
    }
    cout << endl;

    return 0;
}

```

La clase `Lista` puede acceder a todos los miembros de `Elemento`, sean o no públicos, pero desde la función "main" sólo podemos acceder a los miembros públicos de nuestro elemento.

3.5. HERENCIA SIMPLE Y MÚLTIPLE

La herencia, entendida como una característica de la programación orientada a objetos y más concretamente del C++, permite definir una clase modificando una o más clases ya existentes. Estas modificaciones consisten habitualmente en añadir nuevos miembros (variables o funciones), a la clase que se está definiendo, aunque también se puede redefinir variables o funciones miembro ya existentes.

La clase de la que se parte en este proceso recibe el nombre de clase base, y la nueva clase que se obtiene se denomina clase derivada. Ésta a su vez puede ser clase base en un nuevo proceso de derivación, iniciando de esta manera una jerarquía de clases. De ordinario las clases base suelen ser más generales que las clases derivadas.

Esto es así porque a las clases derivadas se les suelen ir añadiendo características, en definitiva variables y funciones que diferencian concretan y particularizan.

Este mecanismo de herencia presenta múltiples ventajas evidentes a primera vista, como la posibilidad de reutilizar código sin tener que escribirlo de nuevo. Esto es posible porque todas las clases derivadas pueden utilizar el código de la clase base sin tener que volver a definirlo en cada una de ellas.

Uno de los problemas que aparece con la herencia es el del control del acceso a los datos. ¿Puede una función de una clase derivada acceder a los datos privados de su clase base? En principio una clase no puede acceder a los datos privados de otra, pero podría ser muy conveniente que una clase derivada accediera a todos los datos de su clase base. Para hacer posible esto, existe el tipo de dato `protected`. Este tipo de datos es privado para todas aquellas clases que no son derivadas, pero público para una clase derivada de la clase en la que se ha definido la variable como `protected`.

Por otra parte, el proceso de herencia puede efectuarse de dos formas distintas: siendo la clase base `public` o `private` para la clase derivada. En el caso de que la clase base sea `public` para la clase derivada, ésta hereda los miembros `public` y `protected` de la clase base como miembros `public` y `protected`, respectivamente. Por el contrario, si la clase base es `private` para la clase derivada, ésta hereda todos los datos de la clase base como `private`.

Una clase puede heredar variables y funciones miembro de una o más clases base. En el caso de que herede los miembros de una única clase se habla de herencia simple y en el caso de que herede miembros de varias clases base se trata de un caso de herencia múltiple. Esto se ilustra en las siguientes figuras:

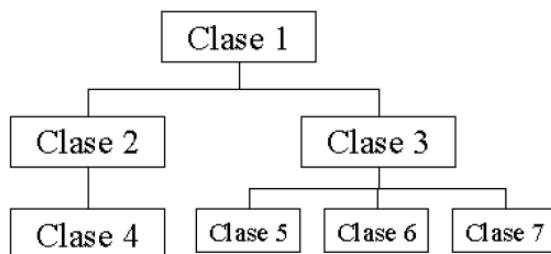


Figura 3.2 Herencia Simple: Todas las clases derivadas tienen una única clase base.

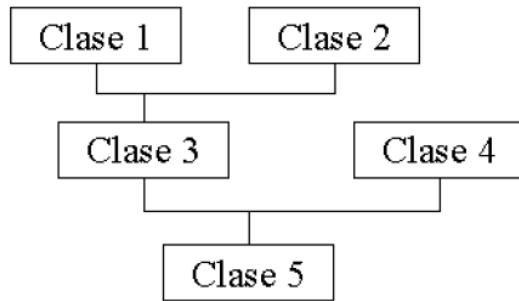


Figura 3.3 Herencia Múltiple: Las clases derivadas tienen varias clases base

3.5.1. DEFINICIÓN DE CLASES SIMPLES

La herencia en C++ es un mecanismo de abstracción creado para poder facilitar y mejorar el diseño de las clases de un programa. Con ella se pueden crear nuevas clases a partir de clases ya hechas, siempre y cuando tengan un tipo de relación especial.

En la herencia, las clases derivadas "heredan" los datos y las funciones miembro de las clases base, pudiendo las clases derivadas redefinir estos comportamientos (polimorfismo) y añadir comportamientos nuevos propios de las clases derivadas. Para no romper el principio de encapsulamiento (ocultar datos cuyo conocimiento no es necesario para el uso de las clases), se proporciona un nuevo modo de visibilidad de los datos/funciones: "protected". Cualquier cosa que tenga visibilidad protected se comportará como pública en la clase Base y en las que componen la jerarquía de herencia, y como privada en las clases que NO sean de la jerarquía de la herencia. Antes de utilizar la herencia, nos tenemos que hacer una pregunta, y si tiene sentido, podemos intentar usar esta jerarquía: Si la frase <claseB> ES-UN <claseA> tiene sentido, entonces estamos ante un posible caso de herencia donde clase A será la clase base y clase B la derivada. Ejemplo: clases Barco, Acorazado, Carguero, etc. un Acorazado ES-UN Barco, un Carguero ES-UN Barco, un Trasatlántico ES-UN Barco, etc. En este ejemplo tendríamos las cosas generales de un Barco (en C++)

```

class Barco {
protected:
    char* nombre;
    float peso;
public:
    //Constructores y demás funciones básicas de barco
};

```

y ahora las características de las clases derivadas, podrían (a la vez que heredan las de barco) añadir cosas propias del subtipo de barco que vamos a crear, por ejemplo:

```

class Carguero: public Barco { // Esta es la manera de especificar que hereda de
Barco
    private:
        float carga;
        //El resto de cosas
    };
class Acorazado: public Barco {
    private:
        int numeroArmas;
        int Soldados;
        // El resto de cosas
};

```

Por último, hay que mencionar que existen 3 clases de herencia que se diferencian en el modo de manejar la visibilidad de los componentes de la clase resultante:

Herencia pública (class Derivada: public Base): Con este tipo de herencia se respetan los comportamientos originales de las visibilidades de la clase Base en la clase Derivada.

Herencia privada (clase Derivada: private Base): Con este tipo de herencia todo componente de la clase Base, será privado en la clase Derivada (las propiedades heredadas serán privadas aunque estas sean públicas en la clase Base)

Herencia protegida (clase Derivada: protected Base) : Con este tipo de herencia, todo componente público y protegido de la clase Base, será protegido en la clase Derivada, y los componentes privados, siguen siendo privados.

La herencia múltiple es el mecanismo que permite al programador hacer clases derivadas a partir, no de una sola clase base, sino de varias. Para entender esto mejor, pongamos un ejemplo: Cuando ves a quien te atiende en una tienda, como persona que es, podrás suponer que puede hablar, comer, andar, pero, por otro lado, como empleado que es, también podrás suponer que tiene un jefe, que puede cobrarte dinero por la compra, que puede devolverte el cambio, etc. Si esto lo trasladamos a la programación sería herencia múltiple (clase empleado_tienda):

```

class Persona {
...
Hablar();
Caminar();
...
};

class Empleado {

```

```

Persona jefe;
int sueldo;
Cobrar();
...
};

class empleado_tienda: public Persona, Empleado {
...
AlmacenarStock();
ComprobarExistencias();
...
};

```

Por tanto, es posible utilizar más de una clase para que otra herede sus características.

3.5.2 CLASES ABSTRACTAS

Habitualmente las funciones virtuales de la clase base de la jerarquía no se utilizan porque en la mayoría de los casos no se declaran objetos de esa clase, y/o porque todas las clases derivadas tienen su propia definición de la función virtual. Sin embargo, incluso en el caso de que la función virtual de la clase base no vaya a ser utilizada, debe declararse.

De todos modos, si la función no va a ser utilizada no es necesario definirla, y es suficiente con declararla como función virtual pura. Una función virtual pura se declara así:

```
virtual funcion_1( ) const=0; //Función virtual pura
```

La única utilidad de esta declaración es la de posibilitar la definición de funciones virtuales en las clases derivadas. De alguna manera se puede decir que la definición de una función como virtual pura hace necesaria la definición de esa función en las clases derivadas, a la vez que imposibilita su utilización con objetos de la clase base.

Al definir una función como virtual pura hay que tener en cuenta que:

- No hace falta definir el código de esa función en la clase base.
- No se pueden definir objetos de la clase base, ya que no se puede llamar a las funciones virtuales puras.
- Sin embargo, es posible definir punteros a la clase base, pues es a través de ellos como será posible manejar objetos de las clases derivadas.

Se denomina *clase abstracta* a aquella que contiene una o más funciones virtuales puras. El nombre proviene que no puede existir ningún objeto de esa clase. Si una clase derivada no redefine una función virtual pura, la clase derivada la hereda como función

virtual pura y se convierte también en clase abstracta. Por el contrario, aquellas clases derivadas que redefinen todas las funciones virtuales puras de sus clases base reciben el nombre de clases derivadas concretas, nomenclatura únicamente utilizada para diferenciarlas de las antes mencionadas.

Aparentemente puede parecer que carece de sentido definir una clase de la que no va a existir ningún objeto, pero se puede afirmar, sin miedo a equivocarse, que la abstracción es una herramienta imprescindible para un correcto diseño de la Programación Orientada a Objetos.

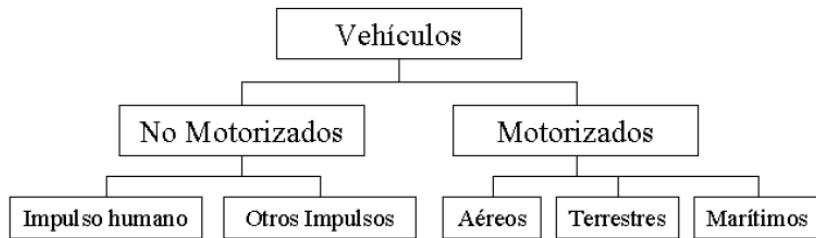


Figura 3.4 Clases base virtuales

3.6 POLIMORFISMO

Polimorfismo, por definición, es la capacidad de adoptar formas distintas. En el ámbito de la Programación Orientada a Objetos se entiende por polimorfismo la capacidad de llamar a funciones distintas con un mismo nombre. Estas funciones pueden actuar sobre objetos distintos dentro de una jerarquía de clases, sin tener que especificar el tipo exacto de los objetos. Esto se puede entender mejor con el ejemplo de la figura 3.4.

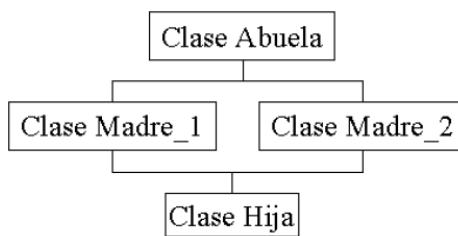


Figura 3.5 Funciones virtuales

En el ejemplo que se ve en la figura 3.4 se observa una jerarquía de clases. En todos los niveles de esta jerarquía está contenida una función llamada `Funcion_1()`. Esta función no tiene por qué ser igual en todas las clases, aunque es habitual que sea una función que efectúe una operación muy parecida sobre distintos tipos de objetos.

Es importante comprender que el compilador no decide en tiempo de compilación cuál será la función que se debe utilizar en un momento dado del programa. Esa decisión se toma en tiempo de ejecución. A este proceso de decisión en tiempo de ejecución se le denomina vinculación dinámica o tardía, en oposición a la habitual vinculación estática o temprana, consistente en decidir en tiempo de compilación qué función se aplica en cada caso. A este tipo de funciones, incluidas en varios niveles de una jerarquía de clases con el mismo nombre pero con distinta definición, se les denomina funciones virtuales. Hay que insistir en que la definición de la función en cada nivel es distinta.

El polimorfismo hace posible que un usuario pueda añadir nuevas clases a una jerarquía sin modificar o recompilar el código original. Esto quiere decir que si desea añadir una nueva clase derivada es suficiente con establecer la clase de la que deriva, definir sus nuevas variables y funciones miembro, y compilar esta parte del código, ensamblándolo después con lo que ya estaba compilado previamente.

Es necesario comentar que las funciones virtuales son algo menos eficientes que las funciones normales. A continuación se explica, sin entrar en gran detalle, el funcionamiento de las funciones virtuales. Cada clase que utiliza funciones virtuales tiene un vector de punteros, uno por cada función virtual, llamado *v-table*. Cada uno de los punteros contenidos en ese vector apunta a la función virtual apropiada para esa clase, que será, habitualmente, la función virtual definida en la propia clase. En el caso de que en esa clase no esté definida la función virtual en cuestión, el puntero de *v-table* apuntará a la función virtual de su clase base más próxima en la jerarquía, que tenga una definición propia de la función virtual. Esto quiere decir que buscará primero en la propia clase, luego en la clase anterior en el orden jerárquico y se irá subiendo en ese orden hasta dar con una clase que tenga definida la función buscada.

Cada objeto creado de una clase que tenga una función virtual contiene un puntero oculto a la *v-table* de su clase. Mediante ese puntero accede a su *v-table* correspondiente y a través de esta tabla accede a la definición adecuada de la función virtual. Es este trabajo extra el que hace que las funciones virtuales sean menos eficientes que las funciones normales.

Como ejemplo se puede suponer que la cuenta_joven y la cuenta_empresarial antes descritas tienen una forma distinta de abonar mensualmente el interés al saldo.

- En la cuenta_joven, no se abonará el interés pactado si el saldo es inferior a un límite.
- En la cuenta_empresarial se tienen tres cantidades límite, a las cuales se aplican factores de corrección en el cálculo del interés. El cálculo de la cantidad abonada debe realizarse de la siguiente forma:
 1. Si el saldo es menor que 50000, se aplica el interés establecido previamente.
 2. Si el saldo está entre 50000 y 500.000, se aplica 1.1 veces el interés establecido previamente.

3. Si el saldo es mayor a 500.000, se aplica 1.5 veces el interés establecido previamente.

El código correspondiente quedaría de la siguiente forma:

```
class C_Cuenta {  
    // Variables miembro  
    private:  
        double Saldo; // Saldo Actual de la cuenta  
        double Interes; // Interés calculado hasta el momento, anual,  
        // en tanto por ciento %  
    public:  
        //Constructor  
        C_Cuenta(double unSaldo=0.0, double unInteres=4.0)  
        {  
            SetSaldo(unSaldo);  
            SetInteres(unInteres);  
        }  
        // Acciones Básicas  
        inline double GetSaldo()  
        { return Saldo; }  
        inline double GetInteres()  
        { return Interes; }  
        inline void SetSaldo(double unSaldo)  
        { Saldo = unSaldo; }  
        inline void SetInteres(double unInteres)  
        { Interes = unInteres; }  
        void Ingreso(double unaCantidad)  
        { SetSaldo( GetSaldo() + unaCantidad ); }  
        virtual void AbonaInteresMensual()  
        {  
            SetSaldo( GetSaldo() * ( 1.0 + GetInteres() / 12.0 / 100.0 ) );  
        }  
        // etc...  
};  
class C_CuentaJoven : public C_Cuenta {  
public:  
    C_CuentaJoven(double unSaldo=0.0, double unInteres=2.0,  
    double unLimite = 50.0E3) :  
        C_Cuenta(unSaldo, unInteres)  
    {  
        Limite = unLimite;
```

```

}

virtual void AbonaInteresMensual()
{
if (GetSaldo() > Limite)
SetSaldo( GetSaldo() * (1.0 + GetInteres() / 12.0 / 100.0) );
else
SetSaldo( GetSaldo() );
}

private:
double Limite;
};

class C_CuentaEmpresarial : public C_Cuenta {
public:
C_CuentaEmpresarial(double unSaldo=0.0, double unInteres=4.0)
:C_Cuenta(unSaldo, unInteres)
{
CantMin[0] = 50.0e3;
CantMin[1] = 500.0e3;
}
virtual void AbonaInteresMensual()
{
SetSaldo( GetSaldo() * (1.0 + GetInteres() * CalculaFactor() / 12.0 / 100.0 ) );
}
double CalculaFactor()
{
if (GetSaldo() < CantMin[0])
return 1.0;
else if (GetSaldo() < CantMin[1])
return 1.1;
else return 1.5;
}
private:
double CantMin[2];
};

```

La idea central del polimorfismo es la de poder llamar a funciones distintas aunque tengan el mismo nombre, según la clase a la que pertenece el objeto al que se aplican. Esto es imposible utilizando nombres de objetos: siempre se aplica la función miembro de la clase correspondiente al nombre del objeto, y esto se decide en tiempo de compilación.

Sin embargo, utilizando punteros puede conseguirse el objetivo buscado. Recuérdese que un puntero a la clase base puede contener direcciones de objetos de cualquiera de las clases derivadas. En principio, el tipo de puntero determina también la función que es llamada, pero si se utilizan funciones virtuales es el tipo de objeto el que apunta el puntero lo que determina la función que se llama. Esta es la esencia del polimorfismo.

4.1 INTRODUCCIÓN A UML

El Lenguaje Unificado de Modelado (Unified Modeling Language, UML) es un lenguaje estándar para escribir planos de software. UML puede utilizarse para visualizar, especificar, construir y documentar los artefactos de un sistema que involucra una gran cantidad de software.

UML es apropiado para modelar desde sistemas de información en empresas hasta aplicaciones distribuidas basadas en la Web, e incluso para sistemas empotrados de tiempo real muy exigentes. Es un lenguaje muy expresivo, que cubre todas las vistas necesarias para desarrollar y luego desplegar tales sistemas. Aunque sea expresivo, UML no es difícil de aprender ni de utilizar. Aprender a aplicar UML de modo eficaz comienza por crear un modelo conceptual del lenguaje, lo cual requiere aprender tres elementos principales: los bloques básicos de construcción de UML, las reglas que dictan cómo pueden combinarse esos bloques y algunos mecanismos comunes que se aplican a lo largo del lenguaje.

UML es sólo un lenguaje y por tanto es tan sólo una parte de un método de desarrollo de software. UML es independiente del proceso, aunque para utilizarlo óptimamente se debería usar en un proceso que fuese dirigido por los casos de uso, centrado en 1 arquitectura, iterativo e incremental.

UML es un lenguaje para:

- **Visualizar.**
- **Especificar.**
- **Construir.**
- **Documentar.**

Artefactos de un sistema con gran cantidad de software.

Un lenguaje proporciona un vocabulario y las reglas para combinar palabras de ese vocabulario con el objetivo de posibilitar la comunicación. Un lenguaje de modelado es un lenguaje cuyo vocabulario y reglas se centran en la representación conceptual y física de un sistema. Un lenguaje de modelado como UML es por tanto un lenguaje estándar para los planos del software.

El modelado proporciona una comprensión de un sistema. Nunca es suficiente un único modelo. Más bien, para comprender cualquier cosa, a menudo se necesitan múltiples modelos conectados entre sí, excepto en los sistemas más triviales. Para sistemas con gran cantidad de software, se requiere un lenguaje que cubra las diferentes vistas de la arquitectura de un sistema mientras evoluciona a través del ciclo de vida del desarrollo de software.

El vocabulario y las reglas de un lenguaje como UML indican cómo crear y leer modelos bien formados, pero no dicen qué modelos se deben crear ni cuándo se deberían crear. Esta es la tarea del proceso de desarrollo de software. Un proceso bien definido guiará a sus usuarios al decidir qué artefactos producir, qué actividades y qué personal se emplea para crearlos y gestionarlos, y cómo usar esos artefactos para medir y controlar el proyecto de forma global.

¿Dónde puede utilizarse UML?

UML está pensado principalmente para sistemas con gran cantidad de software. Ha sido utilizado de forma efectiva en dominios tales como:

- Sistemas de información de empresa.
- Bancos y servicios financieros.
- Telecomunicaciones.
- Transporte.
- Defensa/industria aeroespacial.
- Comercio.
- Electrónica médica.
- Ámbito científico.
- Servicios distribuidos basados en la Web.

UML no está limitado al modelado de software. De hecho, es lo suficientemente expresivo para modelar sistemas que no son software, como flujos de trabajo (workflows) en el sistema jurídico, estructura y comportamiento de un sistema de vigilancia médica de un enfermo, y el diseño de hardware.

4.2 HERRAMIENTAS DEL MODELADO

Las herramientas de modelado de sistemas informáticos, son herramientas que se emplean para la creación de modelos de sistemas que ya existen o que se desarrollarán.

Las herramientas de modelado, permiten crear un "simulacro" del sistema, a bajo costo y riesgo mínimo. A bajo costo porque, al fin y al cabo, es un conjunto de gráficos y textos que representan el sistema, pero no son el sistema físico real (el cual es más costoso). Además minimizan los riesgos, porque los cambios que se deban realizar (por errores o cambios en los requerimientos), se pueden realizar más fácil y rápidamente sobre el modelo que sobre el sistema ya implementado.

Las herramientas de modelado, permiten concentrarse en ciertas características importantes del sistema, prestando menos atención a otras. Los modelos resultados, son

una buena forma de determinar si están representados todos los requerimientos del sistema, como también saber si el analista comprendió qué hará el sistema.

Un sistema informático puede requerir diferentes herramientas de modelado, que resultarán en diferentes tipos de modelos. Las herramientas de modelado utilizadas dependen del analista, del tipo de sistema, de los requerimientos, etc.

Algunas herramientas de modelado:

1. Diagrama de flujo de datos.
2. Diagrama de entidad relación.
3. Diagrama de transición de estados.
4. Diccionario de datos.
5. Especificación de procesos.

4.2.1 DIAGRAMAS DE FLUJO DE DATOS (DFD)

Los diagramas de flujo de datos son un tipo de herramienta de modelado, permiten modelar todo tipo de sistemas, concentrándose en las funciones que realiza, y los datos de entrada y salida de esas funciones.

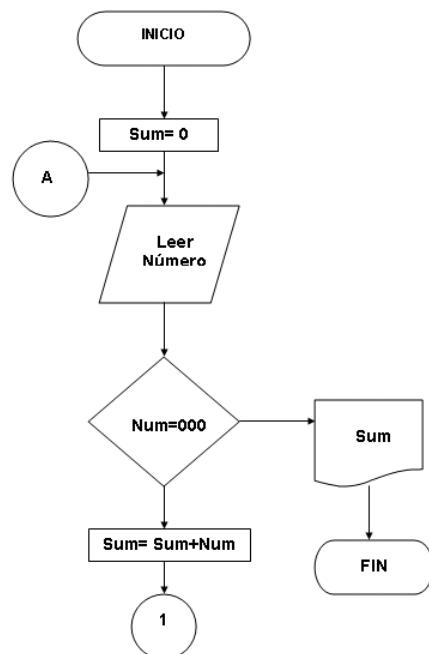


Figura 4.1 Diagrama de flujo de datos.

4.2.1.1 COMPONENTES DE LOS DIAGRAMAS DE FLUJO DE DATOS

Procesos (burbujas): representan la parte del sistema que transforma ciertas entradas en ciertas salidas.

Flujos: representan los datos en movimiento. Pueden ser flujos de entrada o flujos de salida. Los flujos conectan procesos entre sí y también almacenes con procesos.

Almacenes: representan datos almacenados. Pueden ser una base de datos, un archivo físico, etc.

Terminadores: representan entidades externas que se comunican con el sistema. Esas entidades pueden ser personas, organizaciones u otros sistemas, pero no pertenecen al sistema que se está modelando.

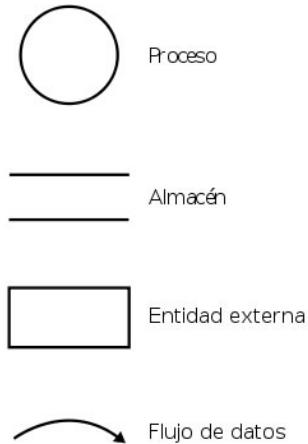


Figura 4.2 Componentes DFD

Existen procesos y flujos especiales llamados procesos de control y flujos de control. Se emplean para modelar sistemas en tiempo real. Los flujos de control son señales o interrupciones, en tanto los procesos de control son burbujas que coordinan y sincronizan otros procesos. Los procesos de control sólo se conectan con flujos de control. Los flujos de control de salida "despiertan" otras burbujas, en tanto los flujos de control de entrada, especifican que una tarea terminó o se presentó un evento extraordinario.

4.2.1.2 REPRESENTACIÓN DE UN SISTEMA EN DIAGRAMA DE FLUJO DE DATOS

Un sistema puede representarse empleando varios diagramas de flujos de datos, cada flujo de datos puede representar una parte "más pequeña" del sistema. Los *DFD* permiten una partición por niveles del sistema. El nivel más general se representa con un *DFD* global llamado diagrama de contexto. El diagrama de contexto *DFD* representa a todo el sistema con una simple burbuja o proceso, las entradas y salidas de todo el sistema, y las interacciones con los terminadores.

4.2.1.3 COMPLEMENTOS DEL DIAGRAMA DE FLUJO DE DATOS

Los *Diagramas de Flujos de Datos* suelen servir para comprender fácilmente el funcionamiento de un sistema. De todas maneras, no es la única herramienta para diagramar sistemas, es más, se debe complementar con otras herramientas para agregar comprensión y exactitud al *DFD*.

4.2.2 DIAGRAMAS ENTIDAD RELACIÓN

Un **Diagrama Entidad Relación** es una herramienta de modelado de sistemas, que se concentra en los datos almacenados en el sistema y las relaciones entre éstos. Un *Diagrama de Entidad Relación* o *DER* es un modelo de red que describe la distribución de los datos almacenados en un sistema de forma abstracta. Algunas bibliografías diferencian entre el diagrama **Entidad Relación** y el **Modelo Entidad Relación**, donde el *Modelo Entidad Relación* vendría a ser el lenguaje utilizado para crear *Diagramas de Entidad Relación*.

Componentes de un DER:

1. Tipos de Objetos o Entidades.
2. Relaciones (Conectan los objetos o entidades)

4.2.3 DIAGRAMAS DE TRANSICIÓN DE ESTADOS

Los **Diagramas de Transición de Estados** son herramientas de modelado de sistemas en tiempo real. Los componentes de un *DTE* son:

Estados: comportamiento del sistema que es observable en el tiempo. Los sistemas tienen un estado inicial, pero pueden tener múltiples estados finales (mutuamente

excluyentes).

Cambios de estados: condiciones y acciones.

Un diagrama de transición de estados puede utilizarse como una especificación de proceso de un proceso de control de un **Diagrama de Flujo de Datos**.

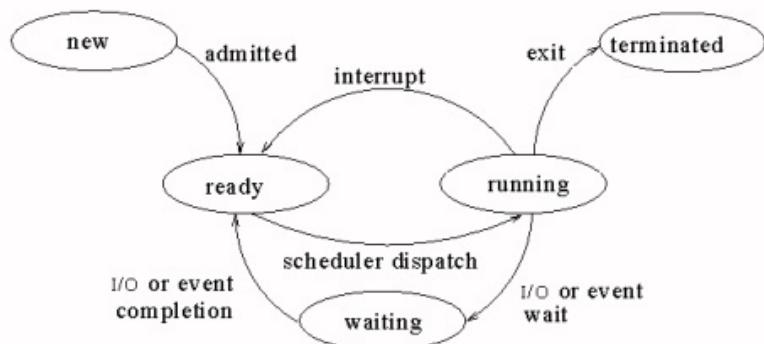


Figura 4.3 Diagrama de transición de estados.

4.2.4 DICCIONARIO DE DATOS

El **Diccionario de Datos** es un listado organizado de todos los datos que pertenecen a un sistema. El objetivo de un diccionario de datos es dar precisión sobre los datos que se manejan en un sistema, evitando así malas interpretaciones o ambigüedades.

Define con precisión los datos de entrada, salida, componentes de almacenes, flujos, detalles de las relaciones entre almacenes, etc. Los diccionarios de datos son buenos complementos a los **Diagramas de Flujo de Datos**, los **Diagramas de Entidad Relación**, etc.

4.2.5 ESPECIFICACIÓN DE PROCESOS

La **Especificación de Procesos**, es una herramienta de modelado de sistemas, que permite definir qué sucede en los procesos o funciones de un sistema. El objetivo es definir qué debe hacerse para transformar ciertas entradas en ciertas salidas.

No hay una única forma de realizar la especificación de procesos; existen múltiples herramientas que facilitan esta tarea, aunque debería emplearse aquellas que permitan fácil comprensión.

Desarrollo de una especificación de procesos. Algunas herramientas utilizadas para generar especificaciones de procesos son:

- ✓ *Lenguaje estructurado*: se emplea un lenguaje natural limitado en palabras y construcciones, dándole más precisión y claridad, evitando ambigüedades (el lenguaje natural humano carece de precisión y es muy ambiguo). Definen un algoritmo.
- ✓ Uso de *Pre-Condiciones* y *Post-Condiciones*: describen la función del proceso, sin detallar un algoritmo específico.
- ✓ Otras: tablas de decisiones, lenguaje narrativo, diagramas de flujos, diagrama Nassi-Shneiderman, gráficas, etc.

4.3 DIAGRAMAS BÁSICOS DE UML

El vocabulario de UML incluye tres clases de bloques de construcción:

1. Elementos.
2. Relaciones.
3. Diagramas.

Los elementos son abstracciones que son ciudadanos de primera clase en un modelo; las relaciones ligan estos elementos entre sí; los diagramas agrupan colecciones interesantes de elementos.

ELEMENTOS EN UML

Hay cuatro tipos de elementos en UML:

1. **Elementos estructurales.**
2. **Elementos de comportamiento.**
3. **Elementos de agrupación.**
4. **Elementos de anotación.**

Estos elementos son los bloques básicos de construcción orientados a objetos de UML. Se utilizan para escribir modelos bien formados.

Elementos estructurales. Los elementos estructurales son los nombres de los modelos UML. En su mayoría son las partes estáticas de un modelo, y representan cosas que son conceptuales o materiales. En total, hay siete tipos de elementos estructurales.

Primero, una **clase** es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Una clase implementa una o más interfaces. Gráficamente, una clase se representa como un rectángulo, que normalmente incluye su nombre, atributos y operaciones, como se muestra en la Figura 4.4.

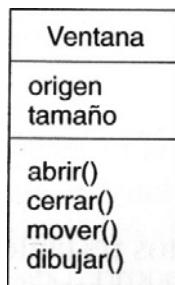


Figura 4.4 Clases

Segundo, una **interfaz** es una colección de operaciones que especifican un servicio de una clase o componente. Por lo tanto, una interfaz describe el comportamiento visible externamente de ese elemento. Una interfaz puede representar el comportamiento completo de una clase o componente o sólo una parte de ese comportamiento. Una interfaz define un conjunto de especificaciones de operaciones (o sea, sus signaturas), pero nunca un conjunto de implementaciones de operaciones. Gráficamente, una interfaz se representa como un Círculo junto con su nombre. Una interfaz raramente se encuentra aislada. Más bien, suele estar conectada a la clase o componente que la realiza, como se muestra en la Figura 4.5.



Figura 4.5 Interfaces

Tercero, una **colaboración** define una interacción y es una sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos. Por lo tanto, las colaboraciones tienen dimensión tanto estructural como de comportamiento. Una clase dada puede participar en varias colaboraciones. Estas colaboraciones representan, pues, la implementación de patrones que forman un sistema. Gráficamente, una colaboración se

representa como una elipse de borde discontinuo, incluyendo normalmente sólo su nombre, como se muestra en la Figura 4.6.

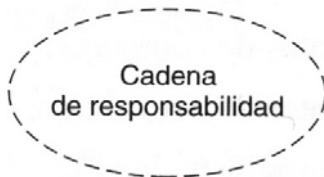


Figura 4.6 Colaboraciones

Cuarto, un **caso de uso** es una descripción de un conjunto de secuencias de acciones que un sistema ejecuta y que produce un resultado observable de interés para un actor particular. Un caso de uso se utiliza para estructurar los aspectos de comportamiento en un modelo. Un caso de uso es realizado por una colaboración. Gráficamente, un caso de uso se representa como una elipse de borde continuo, incluyendo normalmente sólo su nombre, como se muestra en la Figura 4.7.

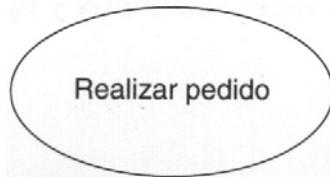


Figura 4.7 Casos de uso.

Los tres elementos restantes (**clases activas, componentes y nodos**) son todos similares a las clases, en cuanto que también describen un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Sin embargo, estos tres son suficientemente diferentes y son necesarios para modelar ciertos aspectos de un sistema orientado a objetos, así que está justificado un tratamiento especial.

Quinto, una **clase activa** es una clase cuyos objetos tienen uno o más procesos o hilos de ejecución y por lo tanto pueden dar origen a actividades de control. Una clase activa es igual que una clase, excepto en que sus objetos representan elementos cuyo comportamiento es concurrente con otros elementos. Gráficamente, una clase activa se representa como una clase, pero con líneas más gruesas, incluyendo normalmente su nombre, atributos y operaciones, como se muestra en la Figura 4.8.

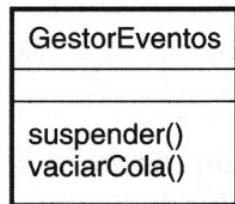


Figura 4.8 Clases activas.

Los dos elementos restantes (**componentes** y **nodos**) también son diferentes. Representan elementos físicos, mientras los cinco elementos anteriores representan cosas conceptuales o lógicas.

Sexto, un **componente** es una parte física y reemplazable de un sistema que conforma con un conjunto de interfaces y proporciona la implementación de dicho conjunto. En un sistema, se podrán encontrar diferentes tipos de componentes de despliegue, tales como componentes COM+ o JavaBeans, así como componentes que sean artefactos del proceso de desarrollo, tales como archivos de código fuente. Un componente representa típicamente el empaquetamiento físico de diferentes elementos lógicos, como clases, interfaces y colaboraciones. Gráficamente, un componente se representa como un rectángulo con pestañas, incluyendo normalmente sólo su nombre, como se muestra en la Figura 4.9.

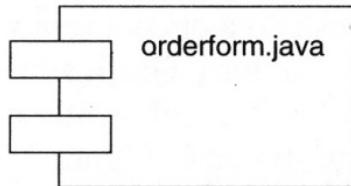


Figura 4.9 Componentes.

Séptimo, un **nodo** es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional, que por lo general dispone de algo de memoria y, con frecuencia, capacidad de procesamiento. Un conjunto de componentes puede residir en un nodo y puede también migrar de un nodo a otro. Gráficamente, un nodo se representa como un cubo, incluyendo normalmente sólo su nombre, como se muestra en la Figura 4.10.

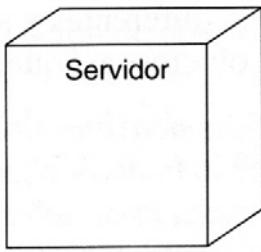


Figura 4.10 Nodos.

Estos siete elementos (**clases**, **interfaces**, **colaboraciones**, **casos de uso**, **clases activas**, **componentes** y **nodos**) son los elementos estructurales básicos que se pueden incluir en un modelo UML. También existen variaciones de estos siete elementos, tales como actores, señales, utilidades (tipos de clases), procesos e hilos (tipos de clases activas), y aplicaciones, documentos, archivos, bibliotecas, páginas y tablas (tipos de componentes).

Elementos de comportamiento. Los elementos de comportamiento son las partes dinámicas de los modelos UML. Estos son los verbos de un modelo, y representan comportamiento en el tiempo y el espacio. En total hay dos tipos principales de elementos de comportamiento:

Primeramente, una **interacción** es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular, para alcanzar un propósito específico. El comportamiento de una sociedad de objetos o una operación individual puede especificarse con una interacción. Una interacción involucra muchos otros elementos, incluyendo mensajes, secuencias de acción (el comportamiento invocado por un mensaje) y enlaces (conexiones entre objetos). Gráficamente, un mensaje se muestra como una línea dirigida, incluyendo casi siempre el nombre de su operación, como se muestra en la Figura 4.11.

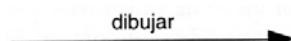


Figura 4.11 Mensajes.

Segundo, una **máquina de estados** es un comportamiento que especifica las secuencias de estados por los que pasa un objeto o una interacción durante su vida en respuesta a eventos, junto con sus reacciones a estos eventos. El comportamiento de una clase individual o una colaboración de clases puede especificarse con una máquina de estados. Una máquina de estados involucra a otros elementos, incluyendo estados, transiciones (el flujo de un estado a otro), eventos (que disparan una transición) y actividades (la respuesta a una transición). Gráficamente, un estado se representa como

un rectángulo de esquinas redondeadas, incluyendo normalmente su nombre y sus sub estados, si los tiene, como se muestra en la Figura 4.12.

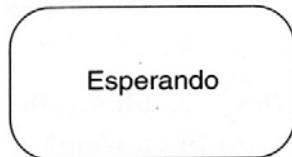


Figura 4.12 Estados.

Estos dos elementos (*interacciones* y *máquinas de estados*) son los elementos básicos de comportamiento que se pueden incluir en un modelo UML. Semánticamente, estos elementos están conectados normalmente a diversos elementos estructurales, principalmente clases, colaboraciones y objetos.

Elementos de agrupación. Los *elementos de agrupación* son las partes organizativas de los modelos UML. Estos son las cajas en las que puede descomponerse un modelo. En total, hay un elemento de agrupación principal, los paquetes.

Un **paquete** es un mecanismo de propósito general para organizar elementos en grupos. Los elementos estructurales, los elementos de comportamiento, e incluso otros elementos de agrupación pueden incluirse en un paquete. Al contrario que los componentes (que existen en tiempo de ejecución), un paquete es puramente conceptual (sólo existe en tiempo de desarrollo). Gráficamente, un paquete se visualiza como una carpeta, incluyendo normalmente sólo su nombre y, a veces, su contenido, como se muestra en la Figura 4.13.

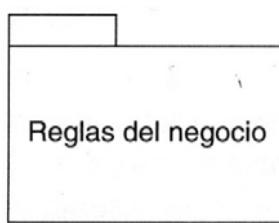


Figura 4.13 Paquete

Los paquetes son los elementos de agrupación básicos con los cuales se puede organizar un modelo UML. También hay variaciones, tales como los frameworks, los modelos y los subsistemas (tipos de paquetes).

Elementos de anotación. Los *elementos de anotación* son las partes explicativas de los modelos UML. Son comentarios que se pueden aplicar para describir, clarificar y hacer observaciones sobre cualquier elemento de un modelo. Hay un tipo principal de

elemento de anotación llamado nota. Una nota es simplemente un símbolo para mostrar restricciones y comentarios junto a un elemento o una colección de elementos. Gráficamente, una nota se representa como un rectángulo con una esquina doblada, junto con un comentario textual o gráfico, como se muestra en la Figura 4.14.

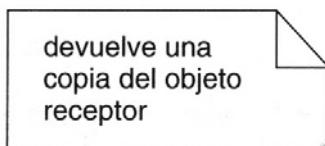


Figura 4.14 Notas

Este elemento es el objeto básico de anotación que se puede incluir en un modelo UML. Típicamente, las notas se utilizarán para adornar los diagramas con restricciones o comentarios que se expresen mejor en texto informal o formal. También hay variaciones sobre este elemento, tales como los requisitos (que especifican algún comportamiento deseado desde la perspectiva externa del modelo).

RELACIONES EN UML

Hay cuatro tipos de relaciones en UML:

1. Dependencia.
2. Asociación.
3. Generalización.
4. Realización.

Estas relaciones son los bloques básicos de construcción para relaciones de UML. Se utilizan para escribir modelos bien formados:

Primero, una **dependencia** es una relación semántica entre dos elementos, en la cual un cambio a un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (el elemento dependiente). Gráficamente, una dependencia se representa como una línea discontinua, posiblemente dirigida, que incluye a veces una etiqueta, como se muestra en la Figura 4.15.



Figura 4.15 Dependencias

Segundo, una **asociación** es una relación estructural que describe un conjunto de enlaces, los cuales son conexiones entre objetos. La agregación es un tipo especial de asociación, que representa una relación estructural entre un todo y sus partes. Gráficamente, una asociación se representa como una línea continua, posiblemente dirigida, que a veces incluye una etiqueta, y a menudo incluye otros adornos, como la multiplicidad y los nombres de rol, como se muestra en la Figura 4.16.



Figura 4.16 Asociaciones

Tercero, una **generalización** es una relación de especialización/generalización en la cual los objetos del elemento especializado (el hijo) pueden sustituir a los objetos del elemento general (el padre). De esta forma, el hijo comparte la estructura y el comportamiento del padre.

Gráficamente, una relación de generalización se representa como una línea continua con una punta de flecha vacía apuntando al padre, como se muestra en la Figura 4.17.



Figura 4.17 Generalizaciones

Cuarto, una **realización** es una relación semántica entre clasificadores, en donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Se pueden encontrar relaciones de realización en dos sitios: entre interfaces y las clases y componentes que las realizan, y entre los casos de uso y las colaboraciones que los realizan. Gráficamente, una relación de realización se representa como una mezcla entre una generalización y una relación de dependencia, como se muestra en la Figura 4.18.



Figura 4.18 Realización

Estos cuatro elementos son los elementos básicos relacionales que se pueden incluir en un modelo UML. También existen variaciones de estos cuatro, tales como el **refinamiento**, la **traza**, la **inclusión** y la **extensión** (para las dependencias).

DIAGRAMAS EN UML

Un **diagrama** es la representación gráfica de un conjunto de elementos, visualizado la mayoría de las veces como un grafo conexo de nodos (elementos) y arcos (relaciones). Los diagramas se dibujan para visualizar un sistema desde diferentes perspectivas, de forma que un diagrama es una proyección de un sistema.

Para todos los sistemas, excepto los más triviales, un diagrama representa una vista resumida de los elementos que constituyen un sistema. El mismo elemento puede aparecer en todos los diagramas, sólo en unos pocos diagramas (el caso más común), o en ningún diagrama (un caso muy raro). En teoría, un diagrama puede contener cualquier combinación de elementos y relaciones. En la práctica, sin embargo, sólo surge un pequeño número de combinaciones, las cuales son consistentes con las cinco vistas más útiles que comprenden la arquitectura de un sistema con gran cantidad de software. Por esta razón, UML incluye nueve de estos diagramas:

1. Diagrama de clases.
2. Diagrama de objetos.
3. Diagrama de casos de uso.
4. Diagrama de secuencia.
5. Diagrama de colaboración.
6. Diagrama de estados (statechart).
7. Diagrama de actividades.
8. Diagrama de componentes.
9. Diagrama de despliegue.

Un **diagrama de clases** muestra un conjunto de clases, interfaces y colaboraciones, así como sus relaciones. Estos diagramas son los diagramas más comunes en el modelado de sistemas orientados a objetos. Los diagramas de clases cubren la vista de diseño estática de un sistema. Los diagramas de clases que incluyen clases activas cubren la vista de procesos estática de un sistema.

Un **diagrama de objetos** muestra un conjunto de objetos y sus relaciones. Los diagramas de objetos representan instantáneas de instancias de los elementos encontrados en los diagramas de clases. Estos diagramas cubren la vista de diseño estática o la vista de procesos estática de un sistema como lo hacen los diagramas de clases, pero desde la perspectiva de casos reales o prototípicos.

Un **diagrama de casos** de uso muestra un conjunto de casos de uso y actores (un tipo especial de clases) y sus relaciones. Los diagramas de casos de uso cubren la vista de casos de uso estática de un sistema. Estos diagramas son especialmente importantes en el modelado y organización del comportamiento de un sistema.

Tanto los diagramas de secuencia como los diagramas de colaboración son un tipo de diagramas de interacción. Un **diagrama de interacción** muestra una interacción, que consta de un conjunto de objetos y sus relaciones, incluyendo los mensajes que

pueden ser enviados entre ellos. Los diagramas de interacción cubren la vista dinámica de un sistema. Un diagrama de secuencia es un diagrama de interacción que resalta la ordenación temporal de los mensajes; un diagrama de colaboración es un diagrama de interacción que resalta la organización estructural de los objetos que envían y reciben mensajes. Los diagramas de secuencia y los diagramas de colaboración son isomorfos, es decir, que se puede tomar uno y transformarlo en el otro.

Un **diagrama de estados** muestra una máquina de estados, que consta de estados, transiciones, eventos y actividades. Los diagramas de estados cubren la vista dinámica de un sistema. Son especialmente importantes en el modelado del comportamiento de una interfaz, una clase o una colaboración y resaltan el comportamiento dirigido por eventos de un objeto, lo cual es especialmente útil en el modelado de sistemas reactivos.

Un **diagrama de actividades** es un tipo especial de diagrama de estados que muestra el flujo de actividades dentro de un sistema. Los diagramas de actividades cubren la vista dinámica de un sistema. Son especialmente importantes al modelar el funcionamiento de un sistema y resaltan el flujo de control entre objetos.

Un **diagrama de componentes** muestra la organización y las dependencias entre un conjunto de componentes. Los diagramas de componentes cubren la vista de implementación estática de un sistema. Se relacionan con los diagramas de clases en que un componente se corresponde, por lo común, con una o más clases, interfaces o colaboraciones.

Un **diagrama de despliegue** muestra la configuración de nodos de procesamiento en tiempo de ejecución y los componentes que residen en ellos. Los diagramas de despliegue cubren la vista de despliegue estática de una arquitectura. Se relacionan con los diagramas de componentes en que un nodo incluye, por lo común, uno o más componentes.

Esta no es una lista cerrada de diagramas. Las **herramientas** pueden utilizar UML para proporcionar otros tipos de diagramas, aunque estos nueve son, con mucho, los que con mayor frecuencia aparecerán en la práctica.

4.3.1 DIAGRAMAS DE CASOS DE USO

Ningún sistema se encuentra aislado. Cualquier sistema interesante interactúa con actores humanos o mecánicos que lo utilizan con algún objetivo y que esperan que el sistema funcione de forma predecible. Un caso de uso especifica el comportamiento de un sistema o de una parte del mismo, y es una descripción de un conjunto de secuencias de acciones, incluyendo variantes, que ejecuta un sistema para producir un resultado observable de valor para un actor.

Los casos de uso se emplean para capturar el comportamiento deseado del sistema en desarrollo, sin tener que especificar cómo se implementa ese comportamiento. Los casos de uso proporcionan un medio para que los desarrolladores, los usuarios finales del sistema y los expertos del dominio lleguen a una comprensión común del sistema. Además, los casos de uso ayudan a validar la arquitectura y a verificar el sistema mientras evoluciona a lo largo del desarrollo. Conforme se desarrolla el sistema, los casos de uso son realizados por colaboraciones, cuyos elementos cooperan para llevar a cabo cada caso de uso.

Los casos de uso bien estructurados denotan sólo comportamientos esenciales del sistema o de un subsistema, y nunca deben ser excesivamente genéricos ni demasiado específicos.

UML proporciona una representación gráfica de un caso de uso y un actor, como se muestra en la Figura 4.3.16. Esta notación permite visualizar un caso de uso independientemente de su realización y en un contexto con otros casos de uso.

Nombres. Cada caso de uso debe tener un nombre que lo distinga de otros casos de uso. Un nombre es una cadena de texto. Ese nombre solo se llama nombre simple; un nombre de camino consta del nombre del caso de uso precedido del nombre del paquete en el que se encuentra. Normalmente, un caso de uso se dibuja mostrando sólo su nombre, como se ve en la Figura 4.19.



Figura 4.19 Nombres simples y de Camino.

El nombre de un caso de uso puede constar de texto con cualquier número de letras, números y la mayoría de los signos de puntuación (excepto signos como los dos puntos utilizados para separar el nombre de un caso de uso del nombre del paquete que lo contiene y puede extenderse a lo largo de varias líneas. En la práctica, los nombres de los casos de uso son expresiones verbales que describen algún comportamiento del vocabulario del sistema que se está modelando.

4.3.1.1 ACTORES

Un **actor** representa un conjunto coherente de roles que los usuarios de los casos de uso juegan al interactuar con éstos. Normalmente, un actor representa un rol que es jugado por una persona, un dispositivo hardware o incluso otro sistema al interactuar nuestro sistema. Por ejemplo, si una persona trabaja para un banco, podría ser un ResponsablePrestarnos. Si tiene sus cuentas personales en ese banco, está jugando también el rol de Cliente. Una instancia de un actor, por lo tanto, representa una interacción individual con el sistema de una forma específica. Aunque se utilizan actores en los modelos, éstos no forman parte del sistema. Son externos a él.

Como se muestra en la Figura 4.20 los actores se representan como monigote. Se pueden definir categorías generales de actores (como Cliente) y especializarlos (como ClienteComercial) a través de relaciones de generalización.

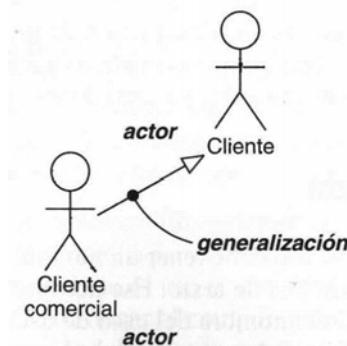


Figura 4.20 Actores

4.3.1.2 FLUJO DE LA INFORMACIÓN

Un caso de uso describe qué hace un sistema (o un subsistema, una clase o una interfaz), pero no especifica cómo lo hace. Cuando se modela, es importante tener clara la separación de objetivos entre las vistas externa e interna.

El comportamiento de un caso de uso se puede especificar describiendo un **flujo de eventos** de forma textual, lo suficientemente claro para que alguien ajeno al sistema lo entienda fácilmente. Cuando se escribe este flujo de eventos se debe incluir cómo y cuándo empieza y acaba el caso de uso, cuándo interactúa con los actores y qué objetos se intercambian, el flujo básico y los flujos alternativos del comportamiento.

4.3.1.2.1 FLUJO NORMAL Y ALTERNATIVO

Flujo normal. Provee una descripción detallada de las acciones del usuario y las respuestas del sistema durante la ejecución normal del caso de uso, son las condiciones esperadas cuando el caso de uso se ejecuta sin contratiempos. Esta secuencia de diálogo en última instancia conduce a lograr el objetivo expuesto en el nombre y la descripción de casos de uso. Esta descripción puede ser escrita como una respuesta a la pregunta hipotética: ¿Cómo se pueden hacer cumplir las tareas que sugiere el nombre del caso de uso? Para lograr una buena conformación del flujo normal se recomienda numerar las acciones de los usuarios y las respuestas del sistema a manera de pasos para llegar a la meta del caso de uso.

Por ejemplo, en el contexto de un cajero automático, se podría describir el caso de uso ValidarUsuario de la siguiente forma:

- *Flujo de eventos principal:* El caso de uso comienza cuando el sistema pide al Cliente un número de identificación personal (PIN, Personal Identification Number). El Cliente puede introducir un PIN a través del teclado. El Cliente acepta la entrada pulsando el botón Enter. El sistema comprueba entonces este PIN para ver si es válido. Si el PIN es válido, el sistema acepta la entrada, y así acaba el caso de uso.

Flujo alternativo. Documenta los escenarios que pueden ocurrir cuando el caso de uso no puede ser ejecutado completamente. Se enumeran igual que el flujo normal de eventos con las acciones del usuario y las respuestas del sistema.

Del ejemplo anterior:

- *Flujo de eventos excepcional:* El Cliente puede cancelar una transacción en cualquier momento pulsando el botón Cancelar, reiniciando de esta forma el caso de uso. No se efectúa ningún cambio a la cuenta del Cliente.
- *Flujo de eventos excepcional:* El Cliente puede borrar un PIN en cualquier momento antes de introducirlo, y volver a teclear un nuevo PIN.
- *Flujo de eventos excepcional:* Si el Cliente introduce un PIN inválido, el caso de uso vuelve a empezar. Si esto ocurre tres veces en una sesión, el sistema cancela la transacción completa, impidiendo que el Cliente utilice el cajero durante 60 segundos.

Nombre:	Crear mensaje foro
Autor:	Joaquin Gracia
Fecha:	24/08/2003
Descripción:	Permite crear un mensaje en el foro de discusión.
Actores:	Usuario de Internet logeado.
Precondiciones:	El usuario debe haberse logeado en el sistema.
Flujo Normal:	<ol style="list-style-type: none"> 1. El actor pulsa sobre el botón para crear un nuevo mensaje. 2. El sistema muestra una caja de texto para introducir el título del mensaje y una zona de mayor tamaño para introducir el cuerpo del mensaje. 3. El actor introduce el título del mensaje y el cuerpo del mismo. 4. El sistema comprueba la validez de los datos y los almacena.
Flujo Alternativo:	<ol style="list-style-type: none"> 4. El sistema comprueba la validez de los datos, si los datos no son correctos, se avisa al actor de ello permitiéndole que los corrija
Poscondiciones:	El mensaje ha sido almacenado en el sistema.

Figura 4.21 Ejemplo de un caso de uso.

4.3.1.3 ROL

En el Lenguaje Unificado de Modelado (UML), un actor "especifica un rol jugado por un usuario o cualquier otro sistema que interactúa con el sujeto". Un actor modela un tipo de rol jugado por una entidad que interactúa con el sujeto (esto es, intercambiando signos y datos), pero que es externo a dicho sujeto.

Los actores pueden representar roles jugados por usuarios humanos, hardware externo, u otros sujetos. Un actor no necesariamente representa una entidad física específica, sino simplemente una faceta particular (es decir, un "rol") de alguna actividad que es relevante a la especificación de sus casos de uso asociados. Así, una única instancia física puede jugar el rol de muchos actores diferentes y, asimismo, un actor dado puede ser interpretado por múltiples instancias diferentes.

4.4 DIAGRAMAS DE ACTIVIDADES

Los **diagramas de actividades** son uno de los cinco tipos de diagramas de UML que se utilizan para el modelado de los aspectos dinámicos de los sistemas. Un diagrama de actividades es fundamentalmente un diagrama de flujo que muestra el flujo de control entre actividades.

Los diagramas de actividades se utilizan para modelar los aspectos dinámicos de un sistema. La mayoría de las veces, esto implica modelar los pasos secuenciales (y posiblemente concurrentes) de un proceso computacional. Con un diagrama de actividades también se puede modelar el flujo de un objeto conforme pasa de estado a estado en diferentes puntos del flujo de control.

Los diagramas de actividades pueden utilizarse para visualizar, especificar, construir y documentar la dinámica de una sociedad de objetos, o pueden emplearse para modelar el flujo de control de una operación. Mientras que los diagramas de interacción destacan el flujo de control entre objetos, los diagramas de actividades destacan el flujo de control entre actividades. Una actividad es una ejecución no atómica en curso, dentro de una máquina de estados. Las actividades producen alguna acción, compuesta de computaciones atómicas ejecutables que producen un cambio en el estado del sistema o el retorno de un valor.

Los diagramas de actividades no son sólo importantes para modelar los aspectos dinámicos de un sistema, sino también para construir sistemas ejecutables a través de ingeniería directa e inversa.

Un diagrama de actividades muestra el flujo de actividades. Una actividad es una ejecución no atómica en curso, dentro de una máquina de estados. Las actividades producen finalmente alguna acción, que está compuesta de computaciones atómicas ejecutables que producen un cambio en el estado del sistema o la devolución de un valor. Las acciones incluyen llamadas a otras operaciones, envío de señales, creación o destrucción de objetos o simples cálculos, como la evaluación de una expresión. Gráficamente diagrama de actividades es una colección de nodos y arcos.

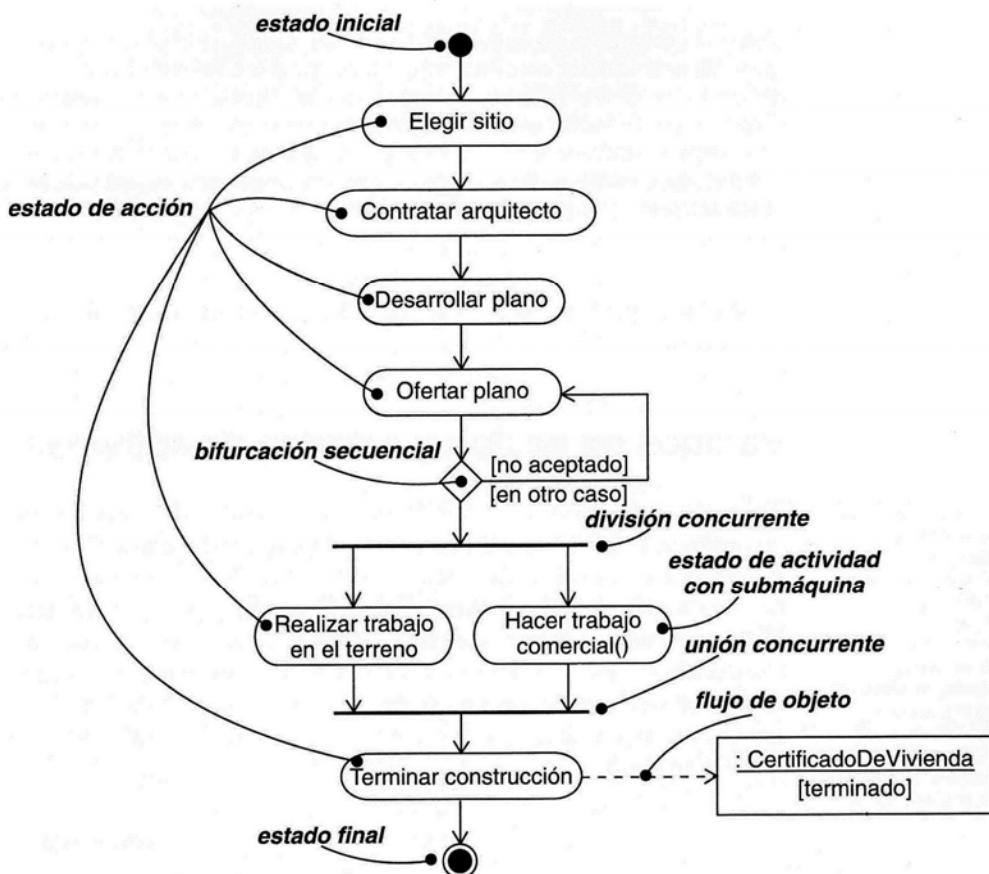


Figura 4.22 Diagramas de Actividades

Un diagrama de actividades es un tipo especial de diagrama y comparte las propiedades comunes al resto de los diagramas (un nombre y un contenido gráfico que es una proyección de un modelo). Lo que distingue a un diagrama de actividades de los otros tipos de diagramas es su contenido.

Un diagrama de actividades es básicamente una proyección de los elementos de grafo de actividades, un caso especial de máquina de estados en la que todos o la mayoría de los estados son estados de actividad y en la cual todas o casi todas las transiciones se disparan por la terminación de actividades en el estado origen. Un diagrama de actividades es un tipo de máquina de estados, por lo que se le aplican todas las características de las máquinas de estados. Esto significa que los diagramas de actividades pueden contener estados simples y compuestos, bifurcaciones, divisiones y uniones.

4.4.1 OPERACIONES

En el flujo de control modelado por un diagrama de actividades suceden cosas. Por ejemplo, se podría evaluar una expresión que estableciera el valor de un atributo o devolviera algún valor. También se podría invocar una operación sobre un objeto, enviar una señal a un objeto o incluso crear o destruir un objeto. Estas computaciones ejecutables y atómicas se llaman estados de acción, porque son estados del sistema, y cada representa la ejecución de una acción. Como se muestra en la Figura 4.23, un estado acción se representa con una figura en forma de píldora (un símbolo con líneas horizontales arriba y abajo y lados convexos). Dentro de esa figura se puede escribir cualquier expresión.

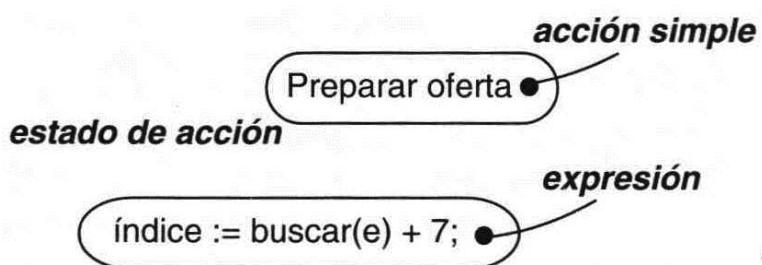


Figura 4.23 Estados de acción.

Los **Estados de Acción** no se pueden descomponer. Además, los estados de acción atómicos, lo que significa que pueden ocurrir eventos, pero no se interrumpe la ejecución del estado de acción. Por último, se considera generalmente que la ejecución de un estado de acción conlleva un tiempo insignificante.

En contraposición, los **Estados de Actividad** pueden descomponerse aún más, representando su actividad con otros diagramas de actividades. Además, los estados de actividad no son atómicos, es decir, pueden ser interrumpidos y, en general, se considera que invierten algún tiempo en completarse. Un estado de acción se puede ver como un caso especial de un estado de actividad. Un estado de acción es un estado de actividad que no se puede descomponer más. Análogamente, un estado de actividad puede ser visto como un elemento compuesto, cuyo flujo de control se compone de otros estados de actividad y estados de acción. Si se entra en los detalles de un estado de actividad se encontrará otro diagrama de actividades. Como se muestra en la Figura 4.24, no hay distinción en cuanto a la notación de los estados de actividad y los estados de acción, excepto que un estado de actividad puede tener partes adicionales, como acciones de entrada y salida (entry/exit) (acciones relacionadas con la entrada y la salida del estado, respectivamente) y especificaciones de submáquinas.

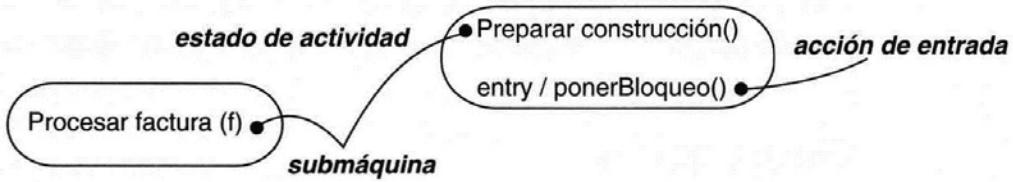


Figura 4.24 Estados de Actividad.

Los estados de acción y los estados de actividad son tipos especiales de estados de una máquina de estados. Al entrar en un estado de acción o un estado de actividad, simplemente se ejecuta la acción o la actividad; al terminar, el control pasa a la siguiente acción o actividad. Por lo tanto, los estados de actividad son una especie de forma simplificada. Un estado de actividad es semánticamente equivalente a la expansión de su grafo de actividades (y así transitivamente) hasta ver sólo acciones. Sin embargo, los estados de actividad son importantes porque ayudan a dividir los cálculos complejos en partes, de la misma forma en que utilizamos las operaciones para agrupar y reutilizar expresiones.

4.4.2 TRANSICIONES

Cuando se completa la acción o la actividad de un estado, el flujo de control pasa inmediatamente al siguiente estado de acción o estado de actividad. Este flujo se especifica con transiciones que muestran el camino de un estado de actividad o estado de acción al siguiente. En UML, una transición se representa como una línea dirigida, como se muestra en la Figura 4.25

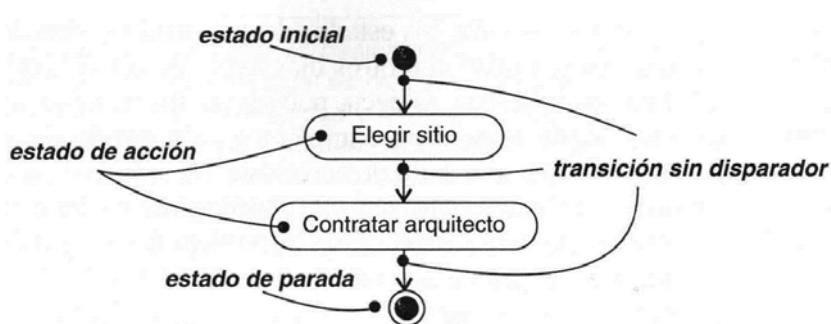


Figura 4.25 Transiciones sin disparadores.

Semánticamente, estas transiciones se llaman transiciones sin disparadores o de terminación, porque el control pasa inmediatamente una vez que se ha finalizado la tarea del estado origen. Cuando se completa la acción de un estado origen dado, se ejecuta la acción de salida del estado (si la hay). A continuación, sin ningún retraso, el control sigue por la transición y pasa al siguiente estado de actividad o estado de acción. Se ejecuta la acción de entrada de ese estado (si la hay), a continuación se ejecuta la acción o actividad del estado destino, y de nuevo se sigue por la siguiente transición una vez finalizada la tarea del estado. Este flujo de control continúa indefinidamente (en el caso de una actividad infinita) o hasta que se encuentra un estado de parada.

En realidad, un flujo de control tiene que empezar y parar en algún sitio (a menos por supuesto, que sea un flujo infinito, en cuyo caso tendrá un principio pero no un final). Por lo tanto, como se aprecia en la figura, se puede especificar un estado inicial círculo lleno) y un estado final (un círculo lleno dentro de una circunferencia).

4.5 DIAGRAMAS DE CLASES

Las **clases** son los bloques de construcción más importantes de cualquier sistema orientado a objetos. Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Una clase implementa una o más interfaces.

Las clases se pueden utilizar para capturar el vocabulario del sistema que se está desarrollando. Estas clases pueden incluir abstracciones que formen parte del dominio del problema, así como clases que constituyan una implementación. Se pueden utilizar las clases para representar cosas que sean software, hardware o puramente conceptuales.

Las clases bien estructuradas están bien delimitadas y forman parte de una distribución equilibrada de responsabilidades en el sistema.

UML también proporciona una representación gráfica de las clases, como muestra la Figura 4.26. Esta notación permite visualizar una abstracción independientemente de cualquier lenguaje de programación específico y de forma que permite resaltar las partes más importantes de una abstracción: su nombre, sus atributos y sus operaciones.

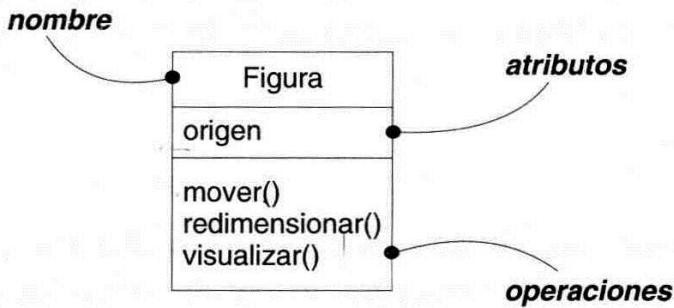


Figura 4.26 Representación gráfica de Clase en UML

Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Gráficamente, una clase se representa como un rectángulo.

4.5.1 RELACIONES

Al realizar abstracciones, uno se da cuenta de que muy pocas clases se encuentran aisladas. En vez de ello, la mayoría colaboran con otras de varias maneras. Por lo tanto, al modelar un sistema, no sólo hay que identificar los elementos que conforman el vocabulario del sistema, también hay que modelar cómo se relacionan estos elementos entre sí.

En el modelado orientado a objetos hay tres tipos de relaciones especialmente importantes: *dependencias*, que representan relaciones de uso entre clases (incluyendo *refinamiento*, *traza* y *ligadura*); *generalizaciones*, que conectan clases generales con sus *especializaciones*, y *asociaciones*, que representan relaciones estructurales entre objetos. Cada una de estas relaciones proporciona una forma diferente de combinar las abstracciones.

La construcción de redes de relaciones no es muy diferente de establecer una distribución equilibrada de responsabilidades entre las clases. Si se modela en exceso, se acabará con un lío de relaciones que harán el modelo incomprensible; si se modela insuficientemente, se habrá perdido un montón de la riqueza del sistema, incluida en la forma en que las cosas colaboran entre ellas.

UML proporciona una representación gráfica para cada uno de estos tipos de relaciones, como se muestra en la Figura 4.27. Esta notación permite visualizar relaciones independientemente de cualquier lenguaje de programación específico, y de forma que permite destacar las partes más importantes de una relación: su nombre, los elementos que conecta y sus propiedades.

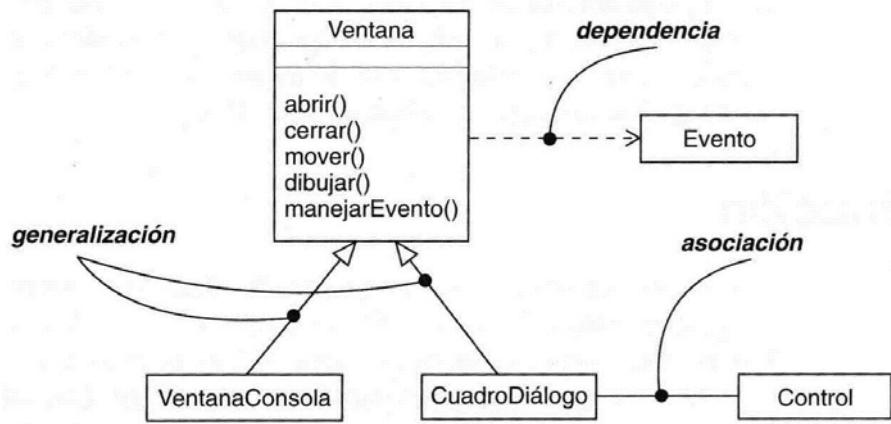


Figura 4.27 Relaciones entre clases

Una **relación** es una conexión entre elementos. En el modelado orientado a objetos, las tres relaciones más importantes son las dependencias, las generalizaciones y las asociaciones. Gráficamente, una relación se representa como una línea, usándose diferentes tipos de línea para diferenciar los tipos de relaciones.

Dependencia. Una dependencia es una relación de uso que declara que un cambio en la especificación de un elemento (por ejemplo, la clase **Evento**) puede afectar a otro elemento que la utiliza (por ejemplo, la clase **Ventana**), pero no necesariamente a la inversa. Gráficamente, una dependencia se representa como una línea discontinua dirigida hacia el elemento del cual se depende. Las dependencias se usarán cuando se quiera indicar que un elemento utiliza a otro.

La mayoría de las veces, las dependencias se utilizarán en el contexto de las clases, para indicar que una clase utiliza a otra como argumento en la firma de una operación; véase la Figura 4.28. Esto es claramente una relación de uso (si la clase utilizada cambia, la operación de la otra clase puede verse también afectada, porque la clase utilizada puede presentar ahora una interfaz o comportamiento diferentes). En UML también se pueden crear dependencias entre otros muchos elementos, especialmente notas y paquetes.

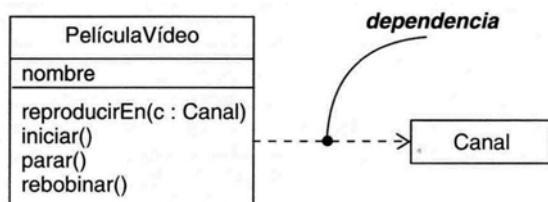


Figura 4.28 Dependencias.

Generalización. Una generalización es una relación entre un elemento general (llamado superclase o padre) y un caso más específico de ese elemento (llamado subclase o hijo). La generalización se llama a veces relación "es-un-tipo-de": un elemento es-un-tipo-de un elemento más general. La generalización significa que los objetos hijos se pueden emplear en cualquier lugar que pueda aparecer el padre, pero no a la inversa. En otras palabras, la generalización significa que el hijo puede sustituir al padre. Una clase hija hereda las propiedades de sus clases padres, especialmente sus atributos y operaciones.

A menudo (no siempre) el hijo añade atributos y operaciones a los que hereda de sus padres. Una operación de un hijo con la misma firma que una operación del padre redefine la operación del padre; esto se conoce como polimorfismo. Gráficamente, la generalización se representa como una línea dirigida continua, con una gran punta de flecha vacía, apuntando al padre, como se muestra en la Figura 4.29. Las generalizaciones se utilizarán cuando se quieran mostrar relaciones padre/hijo.

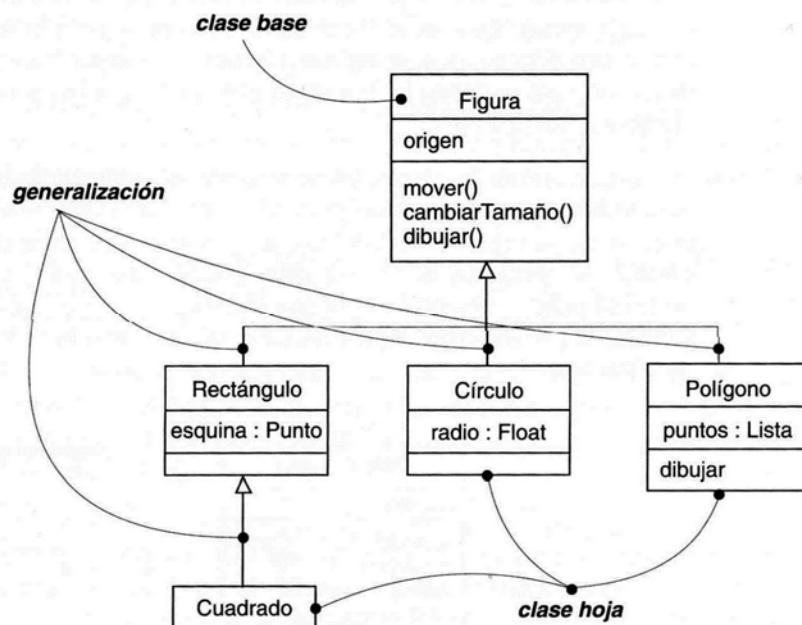


Figura 4.29 Generalización.

Una clase puede tener ninguno, uno o más padres. Una clase sin padres y uno o más hijos se denomina clase raíz o clase base. Una clase sin hijos se llama clase hoja. Una clase con un único parente se dice que utiliza herencia simple; una clase con más de un parente se dice que utiliza herencia múltiple.

En el modelado son muy frecuentes las generalizaciones entre clases e interfaces para reflejar relaciones de herencia. En UML también se pueden establecer generalizaciones entre otros elementos (especialmente entre paquetes).

Asociación. Una asociación es una relación estructural que especifica que los objetos de un elemento están conectados con los objetos de otro. Dada una asociación entre dos clases, se puede navegar desde un objeto de una clase hasta un objeto de la otra clase, y viceversa. Es legal que ambos extremos de una asociación estén conectados a la misma clase. Esto significa que, dado un objeto de la clase, se puede conectar con otros objetos de la misma clase. Una asociación que conecta exactamente dos clases se dice binaria. Aunque no es frecuente, se pueden tener asociaciones que conecten más de dos clases; éstas son llamadas asociaciones n-arias. Gráficamente, una asociación se representa como una línea continua que conecta la misma o diferentes clases. Las asociaciones se utilizarán cuando se quieran representar relaciones estructurales.

Una asociación puede tener un *nombre*, que se utiliza para describir la naturaleza de la relación. Para que no haya ambigüedad en su significado, se puede dar una dirección al nombre por medio de una flecha que apunte en la dirección en la que se pretende que se lea el nombre, como se muestra en la Figura 4.30.

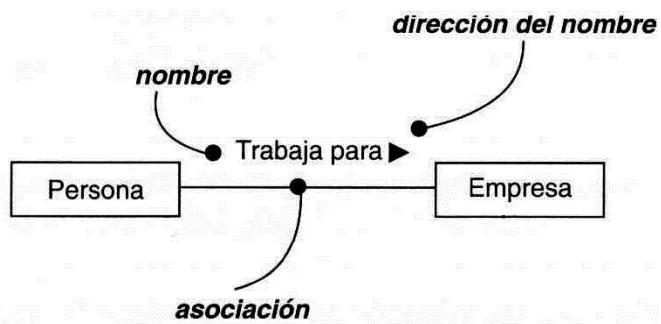


Figura 4.30 Nombre de Asociaciones.

Aunque una asociación puede tener un nombre, normalmente no se necesita incluirlo si se proporcionan explícitamente nombres de rol para la asociación, excepto si se tiene un modelo con muchas asociaciones y es necesario referirse a una o distinguir unas de otras. Esto es especialmente cierto cuando se tiene más de una asociación entre las mismas clases.

Cuando una clase participa en una asociación, tiene un *rol* específico que juega en la asociación; un rol es simplemente la cara que la clase de un extremo de la asociación presenta a la clase del otro extremo. Se puede nombrar explícitamente el rol que juega una clase en una asociación. En la Figura 4.31, una Persona que juega el rol de empleado está asociada con una Empresa que juega el rol de patrón.

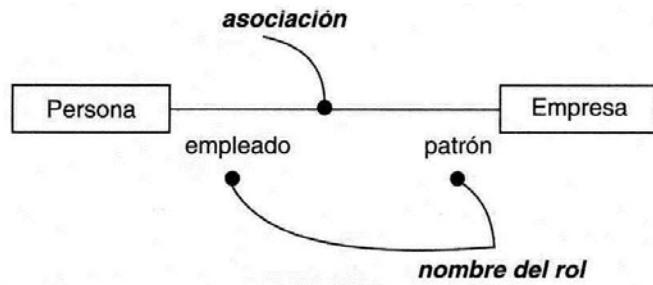


Figura 4.31 Roles.

Una asociación representa una relación estructural entre objetos. En muchas situaciones de modelado, es importante señalar cuántos objetos pueden conectarse a través de una instancia de una asociación. Este "cuántos" se denomina *multiplicidad* del rol de la asociación, y se escribe como una expresión que se evalúa a un rango de valores o a un valor explícito, como se muestra en la Figura 4.32. Cuando se indica una multiplicidad en un extremo de una asociación, se está especificando que, para cada objeto de la clase en el extremo opuesto debe haber tantos objetos en este extremo. Se puede indicar una multiplicidad de exactamente uno (1), cero o uno (0 .. 1), muchos (0 .. *), o uno o más (1 .. *). Incluso se puede indicar un número exacto (por ejemplo, 3).

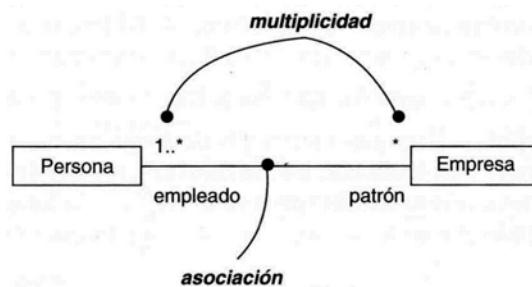


Figura 4.32 Multiplicidad.

Una asociación normal entre dos clases representa una relación estructural entre iguales, es decir, ambas clases están conceptualmente en el mismo nivel, sin ser ninguna más importante que la otra. A veces, se desea modelar una relación "todo/parte", en la cual una clase representa una cosa grande (el "todo"), que consta de elementos más pequeños (las "partes"). Este tipo de relación se denomina *agregación*, la cual representa una relación del tipo "tiene-un", o sea, un objeto del todo tiene objetos de la parte. En realidad, la agregación es sólo un tipo especial de asociación y se especifica añadiendo a una asociación normal un rombo vacío en la parte del todo, como se muestra en la Figura 4.33.

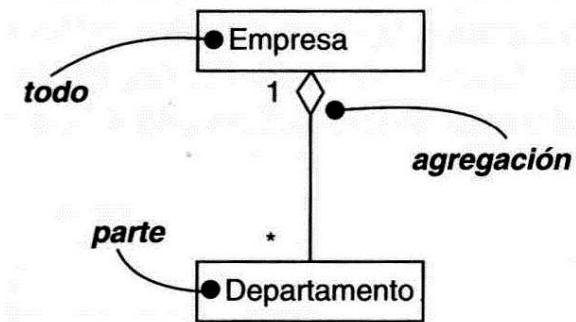


Figura 4.33 Agregación

El significado de esta forma simple de agregación es completamente conceptual. El rombo vacío distingue el "todo" de la "parte", ni más ni menos. Esto significa que la agregación simple no cambia el significado de la navegación a través de la asociación entre el todo y sus partes, ni liga la existencia del todo y sus partes.

4.5.2 ATRIBUTOS

Un **atributo** es una propiedad de una clase identificada con un nombre, que describe un rango de valores que pueden tomar las instancias de la propiedad. Una clase puede tener cualquier número de atributos o no tener ninguno. Un atributo representa alguna propiedad del elemento que se está modelando que es compartida por todos los objetos de esa clase. Por ejemplo, toda pared tiene una altura, una anchura y un grosor; se podrían modelar los clientes de forma que cada uno tenga un nombre, una dirección, un teléfono y una fecha de nacimiento. Un atributo es, por tanto, una abstracción de un tipo de dato o estado que puede incluir un objeto de la clase. En un momento dado, un objeto de una clase tendrá valores específicos para cada uno de los atributos de su clase. Gráficamente, los atributos se listan en un compartimento justo debajo del nombre de la clase. Los atributos se pueden representar mostrando sólo sus nombres, como se ve en la Figura 4.34.

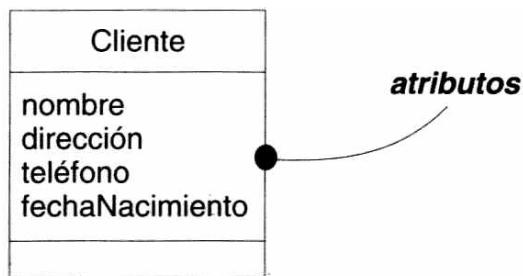


Figura 4.34 Atributos

El nombre de un atributo puede ser texto, igual que el nombre de una clase. En la práctica, el nombre de un atributo es un nombre corto o una expresión nominal que representa alguna propiedad de la clase que lo engloba. Normalmente, se pone en mayúsculas la primera letra de cada palabra de un atributo, excepto la primera letra, como en nombre o esMaestra. En la figura 4.35 se muestra como se especifican los atributos de una clase con el tipo de dato.

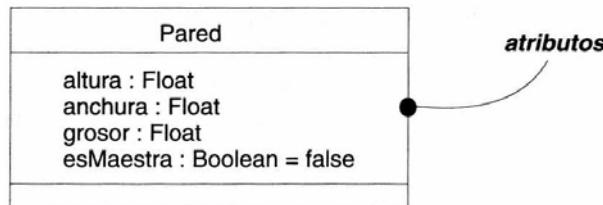


Figura 4.35 Atributos y su clase.

4.5.3 OPERACIONES

Una **operación** es la implementación de un servicio que puede ser requerido a cualquier objeto de la clase para que muestre un comportamiento. En otras palabras, una operación es una abstracción de algo que se puede hacer a un objeto y que es compartido por todos los objetos de la clase. Una clase puede tener cualquier número de operaciones o ninguna. Por ejemplo, en una biblioteca de ventanas como la del paquete awt de Java, todos los objetos de la clase Rectangle pueden ser movidos, redimensionados o consultados sobre sus propiedades. A menudo (pero no siempre), la invocación de una operación sobre un objeto cambia los datos o el estado del objeto. Gráficamente, las operaciones se listan en un compartimento justo debajo de los atributos de la clase. Las operaciones se pueden representar mostrando sólo sus nombres, como se ve en la Figura 4.36.

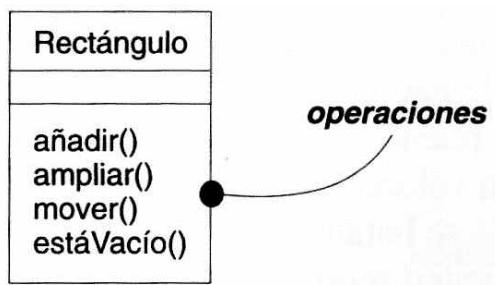


Figura 4.36 Operaciones.

El nombre de una operación puede ser texto, igual que el nombre de una clase. En la práctica, el nombre de una operación es un verbo corto o una expresión verbal que representa un comportamiento de la clase que la contiene. Normalmente, se pone en mayúsculas la primera letra de cada palabra en el nombre de una operación excepto la primera letra, como en mover o estaVacio.

Una operación se puede especificar indicando su signatura, la cual incluye el nombre, tipo y valores por defecto de todos los parámetros y (en el caso de las funciones) un tipo de retorno, como se muestra en la figura 4.37.

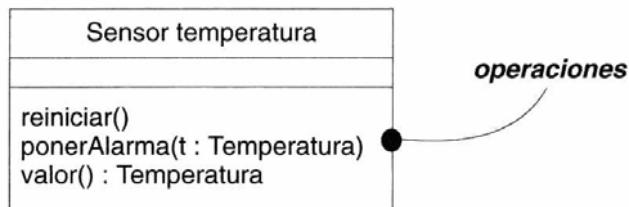


Figura 4.37 Operaciones y sus asignaturas.

4.6 DIAGRAMAS DE SECUENCIA

Un diagrama de secuencia destaca la ordenación temporal de los mensajes. Como se muestra en la figura 4.38 un diagrama de secuencia se forma colocando en primer lugar los objetos que participan en la interacción en la parte superior del diagrama, a lo largo del eje X. Normalmente, se coloca a la izquierda el objeto que inicia la interacción, y los objetos subordinados a la derecha. A continuación, se colocan los mensajes que estos objetos envían y reciben a lo largo del eje Y, en orden de sucesión en el tiempo, desde arriba hasta abajo. Esto ofrece al lector una señal visual clara del flujo de control a lo largo del tiempo.

Los diagramas de secuencia tienen dos características que los distinguen de los diagramas de colaboración.

En primer lugar, está la línea de vida. La línea de vida de un objeto es la línea discontinua vertical que representa la existencia de un objeto a lo largo de un período de tiempo. La mayoría de los objetos que aparecen en un diagrama de interacción existirán mientras dure la interacción, así que los objetos se colocan en la parte superior del diagrama, con sus líneas de vida dibujadas desde arriba hasta abajo. Pueden crearse objetos durante la interacción. Sus líneas de vida comienzan con la recepción del mensaje estereotipado como *create*. Los objetos pueden destruirse durante la interacción. Sus

líneas de vida acaban con la recepción del mensaje estereotipado como `destroy` (además se muestra la señal visual de una gran X que marca el final de sus vidas).

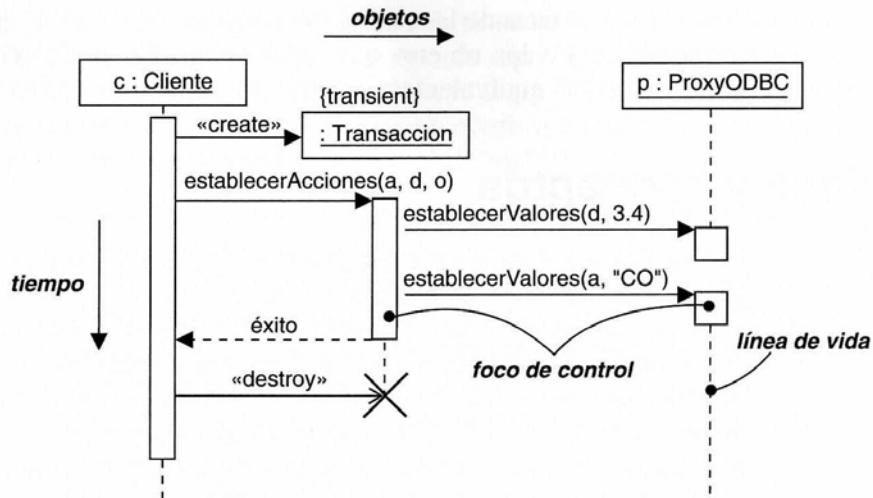


Figura 4.38 Estructura de un diagrama de secuencia

Si un objeto cambia el valor de sus atributos, su estado o sus roles, se puede colocar una copia del icono del objeto sobre su línea de vida en el punto en el que ocurre el cambio, mostrando esas modificaciones.

En segundo lugar, está el foco de control. El foco de control es un rectángulo delgado y estrecho que representa el período de tiempo durante el cual un objeto ejecuta una acción, bien sea directamente o a través de un procedimiento subordinado. La parte superior del rectángulo se alinea con el comienzo de la acción; la inferior se alinea con su terminación (y puede marcarse con un mensaje de retorno). También puede mostrarse el anidamiento de un foco de control (que puede estar causado por recursión, una llamada a una operación propia, o una llamada desde otro objeto) colocando otro foco de control ligeramente a la derecha de su foco padre (esto se puede hacer a cualquier nivel de profundidad). Si se quiere ser especialmente preciso acerca de dónde se encuentra el foco de control, también se puede sombrear la región del rectángulo durante la cual el método del objeto está ejecutándose (y el control no ha pasado a otro objeto).

Al contrario que en un diagrama de secuencia, en un diagrama de colaboración no se muestra explícitamente la línea de vida de un objeto, aunque se pueden representar los mensajes `create` y `destroy`. Además, el foco de control no se muestra explícitamente en un diagrama de colaboración, aunque el número de secuencia de cada mensaje puede indicar el anidamiento.

4.6.1 MENSAJES

Supóngase que se dispone de un conjunto de objetos y un conjunto de enlaces que conectan esos objetos. Si esto es todo lo que se tiene, entonces se está ante un modelo completamente estático que puede representarse mediante un diagrama de objetos. Los diagramas de objetos modelan el estado de una sociedad de objetos en un momento dado y son útiles cuando se quiere visualizar, especificar, construir o documentar una estructura de objetos estática.

Supóngase que se quiere modelar los cambios de estado en una sociedad de objetos a lo largo de un período de tiempo. Se puede pensar en esto como en el rodaje de una película sobre un conjunto de objetos, donde los fotogramas representan momentos sucesivos en la vida de los objetos. Si estos objetos no son totalmente pasivos, se verá cómo se pasan mensajes los unos a los otros, cómo se envían eventos y cómo invocan operaciones. Además, en cada fotograma, se puede mostrar explícitamente el estado actual y el rol de cada instancia individual.

Un **mensaje** es la especificación de una comunicación entre objetos que transmite información, con la expectativa de que se desencadenará una actividad. La recepción de un mensaje puede considerarse una instancia de un evento.

Cuando se pasa un mensaje, la acción resultante es una instrucción ejecutable que constituye una abstracción de un procedimiento computacional. Una acción puede producir un cambio en el estado.

En UML, se pueden modelar varios tipos de acciones.

- Llamada Invoca una operación sobre un objeto; un objeto puede enviarse un mensaje a sí mismo, lo que resulta en la invocación local de una operación.
 - Retorno Devuelve un valor al invocador.
 - Envío Envía una señal a un objeto.
 - Creación Crea un objeto.
 - Destrucción Destruye un objeto; un objeto puede "suicidarse" al destruirse a sí mismo.

En UML también se pueden modelar acciones complejas. Además de los cinco tipo: básicos de acciones listados anteriormente, a un mensaje se le puede asociar una cadena de texto arbitraria, con la cual se pueden introducir expresiones complejas. UML no especifica a sintaxis ni la semántica de tales cadenas de caracteres.

UML permite distinguir visualmente los diferentes tipos de mensajes, como se muestra en la figura 4.39.

El tipo más común de mensaje que se modelará será la llamada, en la cual un objeto invoca una operación de otro objeto (o de él mismo). Un objeto no puede invocar cualquier operación

Los lenguajes como C++ tienen comprobación estática de tipos (aunque sean polimórficos), lo que significa que la validez de una llamada se comprueba en tiempo de compilación. Los lenguajes como Smalltalk, sin embargo, tienen comprobación dinámica de tipos lo que significa que no se puede determinar la validez del envío de un mensaje a un objeto hasta el momento de su ejecución. En UML, un modelo bien formado puede, en general, ser comprobado estáticamente por una herramienta, ya que el desarrollador normalmente conoce al receptor de la operación en tiempo de modelado.

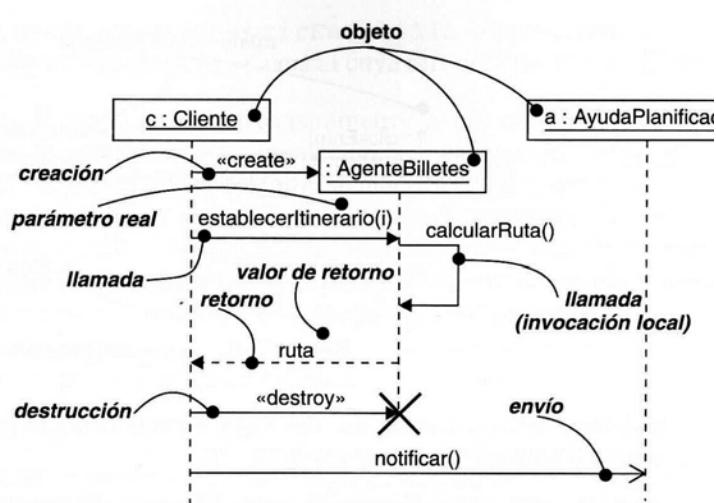


Figura 4.6.2 Tipos de mensajes en un diagrama de secuencia.

Es posible proporcionar parámetros reales al mensaje enviado cuando un objeto invoca una operación o envía una señal a otro objeto. Análogamente, cuando un objeto devuelve el control a otro, también es posible representar el valor de retorno.

4.7 DIAGRAMAS DE COMUNICACIÓN (DIAGRAMAS DE COLABORACIÓN EN UML 1.X)

Un **diagrama de colaboración** en las versiones de UML 1.x es esencialmente un diagrama que muestra interacciones organizadas alrededor de los roles. A diferencia de los diagramas de secuencia, los diagramas de comunicación muestran explícitamente las relaciones de los roles. Por otra parte, un diagrama de comunicación no muestra el tiempo como una dimensión aparte, por lo que resulta necesario etiquetar con números de secuencia tanto la secuencia de mensajes como los hilos concurrentes.

Muestra cómo las instancias específicas de las clases trabajan juntas para conseguir un objetivo común. Implementa las asociaciones del diagrama de clases mediante el paso de mensajes de un objeto a otro. Dicha implementación es llamada "enlace". En la figura 4.40 se muestra un ejemplo de un diagrama de colaboración, donde se puede observar su estructura.

Un **diagrama de comunicación** es también un diagrama de clases que contiene roles de clasificador y roles de asociación en lugar de sólo clasificadores y asociaciones. Los roles de clasificador y los de asociación describen la configuración de los objetos y de los enlaces que pueden ocurrir cuando se ejecuta una instancia de la comunicación. Cuando se instancia una comunicación, los objetos están ligados a los roles de clasificador y los enlaces a los roles de asociación. El rol de asociación puede ser desempeñado por varios tipos de enlaces temporales, tales como argumentos de procedimiento o variables locales del procedimiento. Los símbolos de enlace pueden llevar estereotipos para indicar enlaces temporales. En la figura 4.41 se muestra un ejemplo de un diagrama de comunicación.

Un uso de un diagrama de colaboración es mostrar la implementación de una operación. La comunicación muestra los parámetros y las variables locales de la operación, así como asociaciones más permanentes. Cuando se implementa el comportamiento, la secuencia de los mensajes corresponde a la estructura de llamadas anidadas y el paso de señales del programa.

Un diagrama de secuencia muestra secuencias en el tiempo como dimensión geométrica, pero las relaciones son implícitas. Un diagrama de comunicación muestra relaciones entre roles geométricamente y relaciona los mensajes con las relaciones, pero las secuencias temporales están menos claras.

Es útil marcar los objetos en cuatro grupos: los que existen con la interacción entera; los creados durante la interacción (restricción {new}); los destruidos durante la interacción (restricción {destroyed}); y los que se crean y se destruyen durante la interacción (restricción {transient}).

Aunque las comunicaciones muestran directamente la implementación de una operación, pueden también mostrar la realización de una clase entera. En este uso, muestran el contexto necesario para implementar todas las operaciones de una clase. Esto permite que el modelador vea los roles múltiples que los objetos pueden desempeñar en varias operaciones.

Los *mensajes* se muestran como flechas etiquetadas unidas a los enlaces. Cada mensaje tiene un número de secuencia, una lista opcional de mensajes precedentes, una condición opcional de guarda, un nombre y una lista de argumentos y un nombre de valor de retorno opcional. El nombre de serie incluye el nombre (opcional) de un hilo. Todos los mensajes del mismo hilo se ordenan secuencialmente. Los mensajes de diversos hilos son concurrentes a menos que haya una dependencia secuencial explícita.

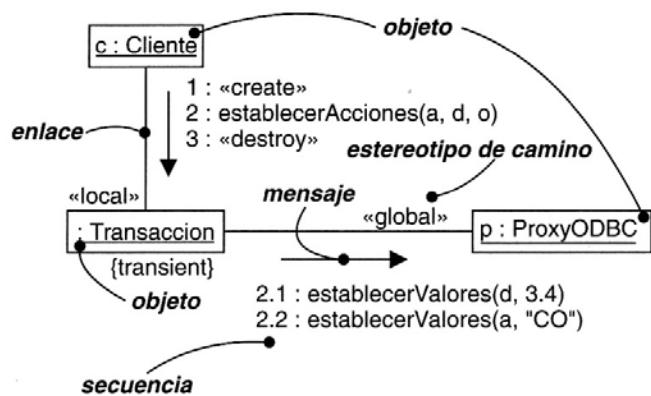


Figura 4.40 Estructura de un diagrama de colaboración (UML 1.x)

Generalmente, un diagrama de comunicación contiene un símbolo para un objeto durante una operación completa. Sin embargo, a veces, un objeto contiene diferentes estados que se deban hacer explícitos. Por ejemplo, un objeto pudo cambiar de localización o sus asociaciones pudieron diferenciarse.

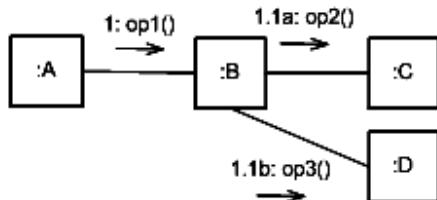


Figura 4.41 Diagrama de comunicación (UML 2.0)

Los diferentes símbolos de objeto que representan un objeto se pueden conectar usando flujos "become" o "conversion". Un flujo "become" es una transición, a partir de un estado de un objeto a otro. Se dibuja como una flecha de línea discontinua con el estereotipo "become" o "conversión" y puede ser etiquetado con un número de serie para mostrar cuando ocurre. Un flujo de conversión también se utiliza para mostrar la migración de un objeto a partir de una localización a otra distinta para otro lugar también se deben marcar con el numero en secuencias.

4.8 DIAGRAMAS DE DESPLIEGUE (DISTRIBUCIÓN)

Los diagramas de despliegue son uno de los dos tipos de diagramas que aparecen cuando se modelan los aspectos físicos de los sistemas orientados a objetos. Un diagrama de despliegue muestra la configuración de nodos que participan en la ejecución y de los componentes que residen en ellos.

Los diagramas de despliegue se utilizan para modelar la vista de despliegue estática de un sistema. La mayoría de las veces, esto implica modelar la topología del hardware sobre el que se ejecuta el sistema. Los diagramas de despliegue son fundamentalmente diagramas de clases que se ocupan de modelar los nodos de un sistema. En la figura 4.42 se muestra un ejemplo de diagrama de despliegue empleando la notación gráfica de UML.

Los diagramas de despliegue no sólo son importantes para visualizar, especificar y documentar sistemas empotrados, sistemas cliente/servidor y sistemas distribuidos, sino también para gestionar sistemas ejecutables mediante ingeniería directa e inversa.

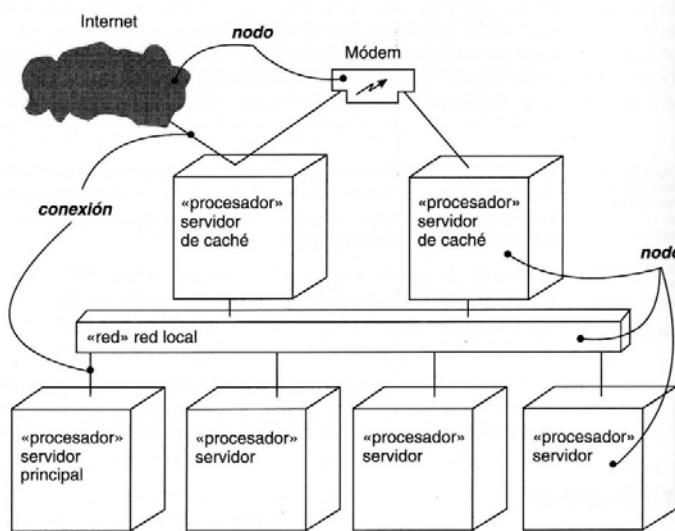


Figura 4.42 Diagrama de despliegue.

Un diagrama de despliegue es un diagrama que muestra la configuración de los nodos que participan en la ejecución y de los componentes que residen en ellos. Gráficamente, un diagrama de despliegue es una colección de nodos y arcos.

Un diagrama de despliegue es un tipo especial de diagrama y comparte las propiedades comunes al resto de los diagramas (un nombre y un contenido gráfico que es

una proyección de un modelo). Lo que distingue a un diagrama de despliegue de los otros tipos de diagramas es su contenido particular.

Normalmente, los diagramas de despliegue contienen:

- Nodos.
- Relaciones de dependencia y asociación.

Al igual que los demás diagramas, los diagramas de despliegue pueden contener notas y restricciones

Los diagramas de despliegue también pueden contener componentes, cada uno de los cuales debe residir en algún nodo. Los diagramas de despliegue también pueden contener paquetes o subsistemas, los cuales se utilizan para agrupar elementos del modelo en bloques más grandes. A veces también se colocarán instancias en los diagramas de despliegue, especialmente cuando se quiera visualizar una instancia de una familia de topologías hardware.

En muchos sentidos, un diagrama de despliegue es un tipo especial de diagrama de clases que se ocupa de modelar los nodos de un sistema.

Los diagramas de despliegue se utilizan para modelar la vista de despliegue estática de un sistema. Esta vista cubre principalmente la distribución, entrega e instalación de las partes que configuran el sistema físico.

Hay varios tipos de sistemas para los que son innecesarios los diagramas de despliegue. Si se desarrolla un software que reside en una máquina e interactúa sólo con dispositivos estándar en esa máquina, que ya son gestionados por el sistema operativo (por ejemplo, el teclado, la pantalla y el módem de un computador personal), se pueden ignorar los diagramas de despliegue.

Por otro lado, si se desarrolla un software que interactúa con dispositivos que normalmente no gestiona el sistema operativo o si el sistema está distribuido físicamente sobre varios procesadores, entonces la utilización de los diagramas de despliegue ayudará a razonar sobre la correspondencia entre el software y el hardware del sistema.

Cuando se modela la vista de despliegue estática de un sistema, normalmente se utilizarán los diagramas de despliegue de una de las tres siguientes maneras.

4.8.1 ESTADOS

Un estado es una condición o situación en la vida de un objeto durante la cual satisface alguna condición, realiza alguna actividad o espera algún evento. Un objeto permanece en un estado durante una cantidad de tiempo finita. Por ejemplo, un

Calentador en una casa puede estar en cualquiera de los cuatro estados siguientes: Inactivo (esperando una orden para calentar la casa), En preparación (el gas está abierto, pero esta esperando a que se eleve la temperatura), Activo (el gas está abierto y el ventilador está encendido) y Apagado (el gas se ha cerrado, pero el ventilador continúa activad arrojando el calor residual del sistema).

Cuando la máquina de estados de un objeto se encuentra en un estado dado, dice que el objeto está en ese estado. Por ejemplo, una instancia de Calentador podrí estar Inactivo o quizá Apagando.

Partes de un estado

1. Nombre	Una cadena de texto que distingue al estado de otros estados; un estado puede ser anónimo, es decir, no tiene nombre.
2. Acciones de entrada/salida	Acciones ejecutadas al entrar y salir del estado, respectivamente.
3. Transiciones internas	Transiciones que se manejan sin causar un cambio en el estado.
4. Subestados	Estructura anidada de un estado, que engloba subestados disjuntos (activos secuencialmente) o concurrentes (activos concurrentemente).
5. Eventos diferidos	Una lista de eventos que no se manejan en este estado sino que se posponen y se añaden a una cola para ser manejados por el objeto en otro estado.

El nombre de un estado puede ser texto formado por cualquier número de letras, dígitos y ciertos signos de puntuación (excepto signos como los dos puntos) y puede extenderse a lo largo de varias líneas. En la práctica, los nombres de estados son nombres cortos o expresiones nominales extraídos del vocabulario del sistema que se está modelando. Normalmente, en el nombre de un estado se pone en mayúsculas la primera letra de cada palabra, como en Inactivo o Apagando. En la figura 4.43 se muestra gráficamente el concepto de estado descrito en esta sección.

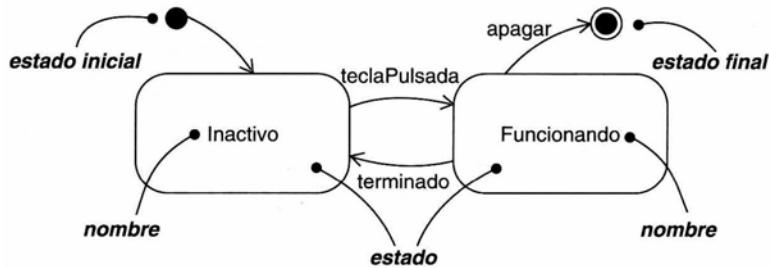


Figura 4.43 Representación de las partes de un estado en un Diagrama de Estados en UML.

4.8.2 TRANSICIONES

Una transición es una relación entre dos estados que indica que un objeto que esté en el primer estado realizará ciertas acciones y entrará en el segundo estado cuando ocurra un evento especificado y se satisfagan unas condiciones especificadas. Cuando produce ese cambio de estado, se dice que la transición se ha disparado. Hasta que se dispara la transición, se dice que el objeto está en el estado origen; después de dispararse, se dice que está en el estado destino. Por ejemplo, un Calentador podría pasar del estado Inactivo al estado EnPreparacion cuando ocurriera un evento como demasiadoFrio (con el parámetro tempDeseada).

Elementos que definen una transición

- | | |
|------------------------|---|
| 1. Estado origen | El estado afectado por la transición; si un objeto está en el estado origen, una transición de salida puede dispararse cuando el objeto reciba el evento de disparo de la transición, y si la condición de guarda, si la hay, se satisface. |
| 2. Evento de disparo | El evento cuya recepción por el objeto que está en el estado origen provoca el disparo de la transición si se satisface su condición de guarda. |
| 3. Condición de guarda | Una expresión booleana que se evalúa cuando la transición se activa por la recepción del evento de disparo: si la expresión toma el valor verdadero, la transición se puede disparar; si la expresión toma el valor falso, la transición no se dispara y si no hay otra transición que pueda ser disparada por el mismo evento, éste se pierde. |

4. Acción Una computación atómica ejecutable que puede actuar directamente sobre el objeto asociado a la máquina de estados, e indirectamente sobre otros objetos visibles al objeto.
5. Estado destino El estado activo tras completarse la transición.

Una transición puede tener múltiples orígenes, en cuyo caso representa una unión (*join*) de muchos estados concurrentes, así como múltiples destinos, en cuyo caso representa una división (*fork*) a múltiples estados concurrentes.

En la figura 4.44 se muestra un ejemplo donde se pueden observar los elementos de una transición y las variantes de una transición.

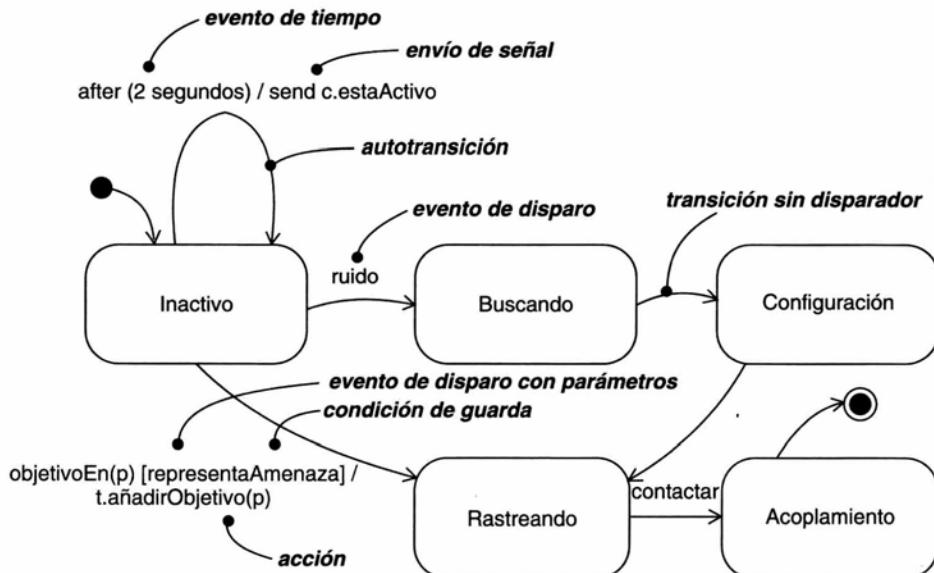


Figura 4.44 Transiciones en un Diagrama de Estados en UML.

Evento de disparo. Un evento es la especificación de un acontecimiento significativo que ocupa un lugar en el tiempo y en el espacio. En el contexto de las máquinas de estados, un evento es la aparición de un estímulo que puede disparar una transición de estado. Como se representa en la figura anterior, los eventos pueden incluir señales, llamadas, el paso del tiempo, o un cambio en el estado. Una señal o una llamada pueden tener parámetros cuyos valores estén disponibles para la transición, incluyendo expresiones para la condición de guarda y la acción.

También es posible tener una transición sin disparador, representada por una transición sin evento de disparo. Una transición sin disparador (también llamada transición de terminación) se dispara implícitamente cuando su estado origen ha completado su actividad.

Un evento de disparo puede ser polimórfico. Por ejemplo, si se ha especificado una familia de señales, entonces una transición cuyo evento de disparo sea *s* puede ser activada por *s*, así como por cualquier evento hijo de *S*.

Condición de guarda. Como se muestra en la figura anterior, una condición de guarda se representa como una expresión booleana entre corchetes y se coloca tras el evento de disparo. Una condición de guarda sólo se evalúa después de ocurrir el evento de disparo de la transición.

Por lo tanto, es posible tener muchas transiciones desde el mismo estado origen y con el mismo evento de disparo, siempre y cuando sus condiciones no se solapen.

Una condición de guarda se evalúa sólo una vez por cada transición, cuando ocurre el evento, pero puede ser evaluada de nuevo si la transición se vuelve a disparar. Dentro de la expresión booleana se pueden incluir condiciones sobre el estado de un objeto (por ejemplo, la expresión *unCalentador* in *Inactivo*, que toma el valor *verdadero* objeto *un Calentador* está actualmente en el estado *Inactivo*).

Aunque una condición de guarda sólo se evalúa una vez cada vez que se activa: transición, un evento de cambio se evalúa potencialmente de forma continua.

Acción. Una acción es una computación atómica ejecutable. Las acciones pueden incluir llamadas a operaciones (sobre el objeto que contiene la máquina de estados, como sobre otros objetos visibles), la creación o destrucción de otro objeto, o el envío una señal a un objeto. Como se muestra en la figura anterior, existe una notación especial para el envío de una señal (el nombre de la señal se precede de la palabra clave *send* como una indicación visual).

Una acción es atómica, es decir, no puede ser interrumpida por un evento y, por tanto, se ejecuta hasta su terminación. Esto contrasta con una actividad, que puede ser ininterrumpida por otros eventos.

Se puede representar explícitamente el objeto al que se envía una señal con una dependencia estereotipada como *send*, cuyo origen sea el estado y cuyo destino sea el objeto.

4.9 CASOS PRÁCTICOS

HOTEL

El dueño de un hotel le pide a usted desarrollar un programa para consultar sobre las piezas disponibles y reservar piezas de su hotel.

El hotel posee tres tipos de piezas: simple, doble y matrimonial, y dos tipos de clientes: habituales y esporádicos. Una reserva almacenará datos del cliente, de la pieza reservada, la fecha de comienzo y el número de días que será ocupada la pieza.

El recepcionista del hotel debe poder hacer las siguientes operaciones:

- Obtener un listado de las piezas disponibles de acuerdo a su tipo
- Preguntar por el precio de una pieza de acuerdo a su tipo
- Preguntar por el descuento ofrecido a los clientes habituales
- Preguntar por el precio total para un cliente dado, especificando su número de RUT, tipo de pieza y número de noches.
- Dibujar en pantalla la foto de una pieza de acuerdo a su tipo
- Reservar una pieza especificando el número de la pieza, RUT y nombre del cliente.
- Eliminar una reserva especificando el número de la pieza

El administrador puede usar el programa para:

- Cambiar el precio de una pieza de acuerdo a su tipo
- Cambiar el valor del descuento ofrecido a los clientes habituales
- Calcular las ganancias que tendrán en un mes especificado (considere que todos los meses tienen treinta días).

El hotel posee información sobre cuales clientes son habituales. Esta estructura puede manejarla con un diccionario, cuya clave sea el número de RUT y como significado tenga los datos personales del cliente.

El diseño a desarrollar debe facilitar la extensibilidad de nuevos tipos de pieza o clientes y a su vez permitir agregar nuevas consultas.

Diagrama de Casos de Uso

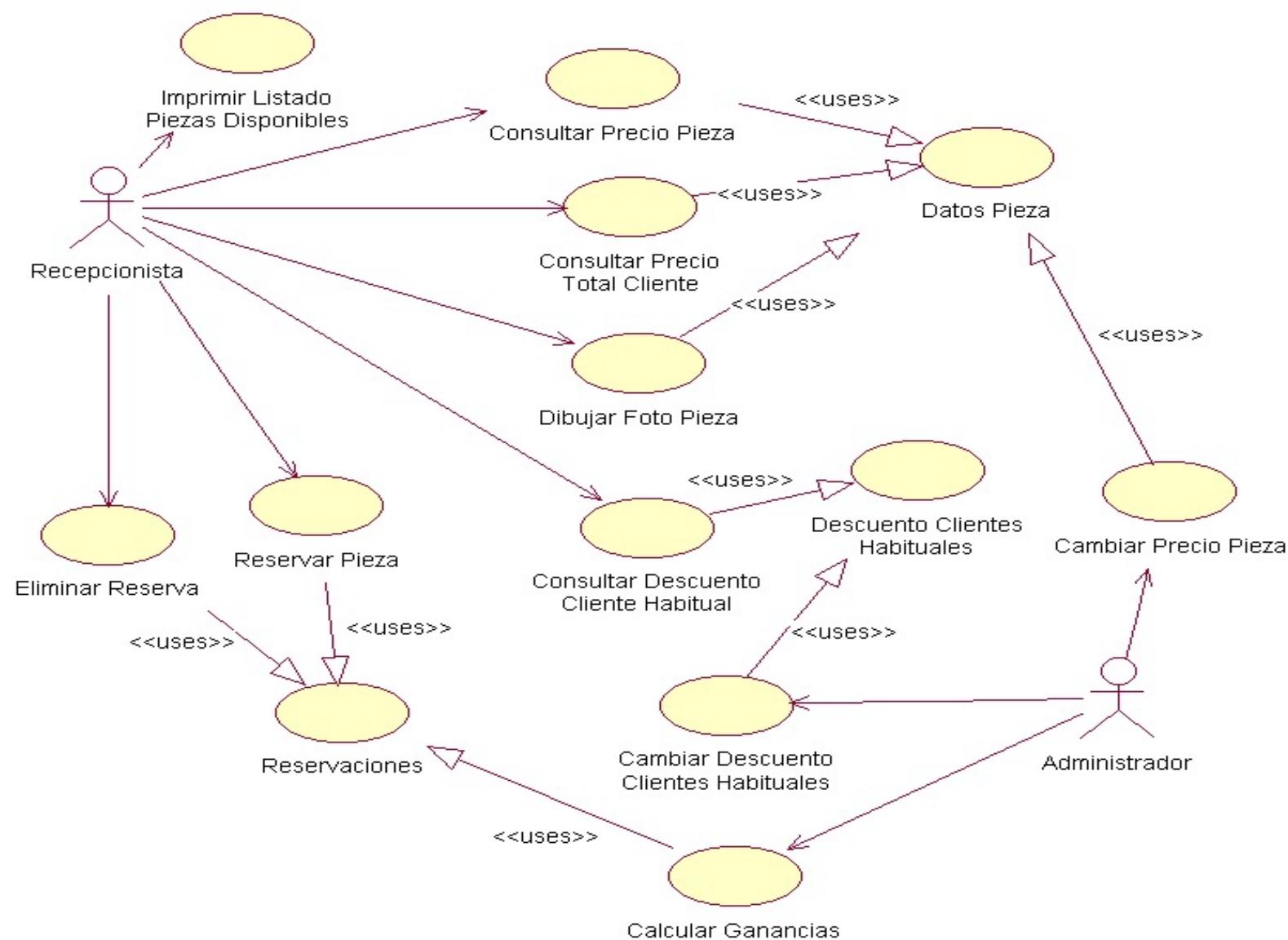


Diagrama de Clases

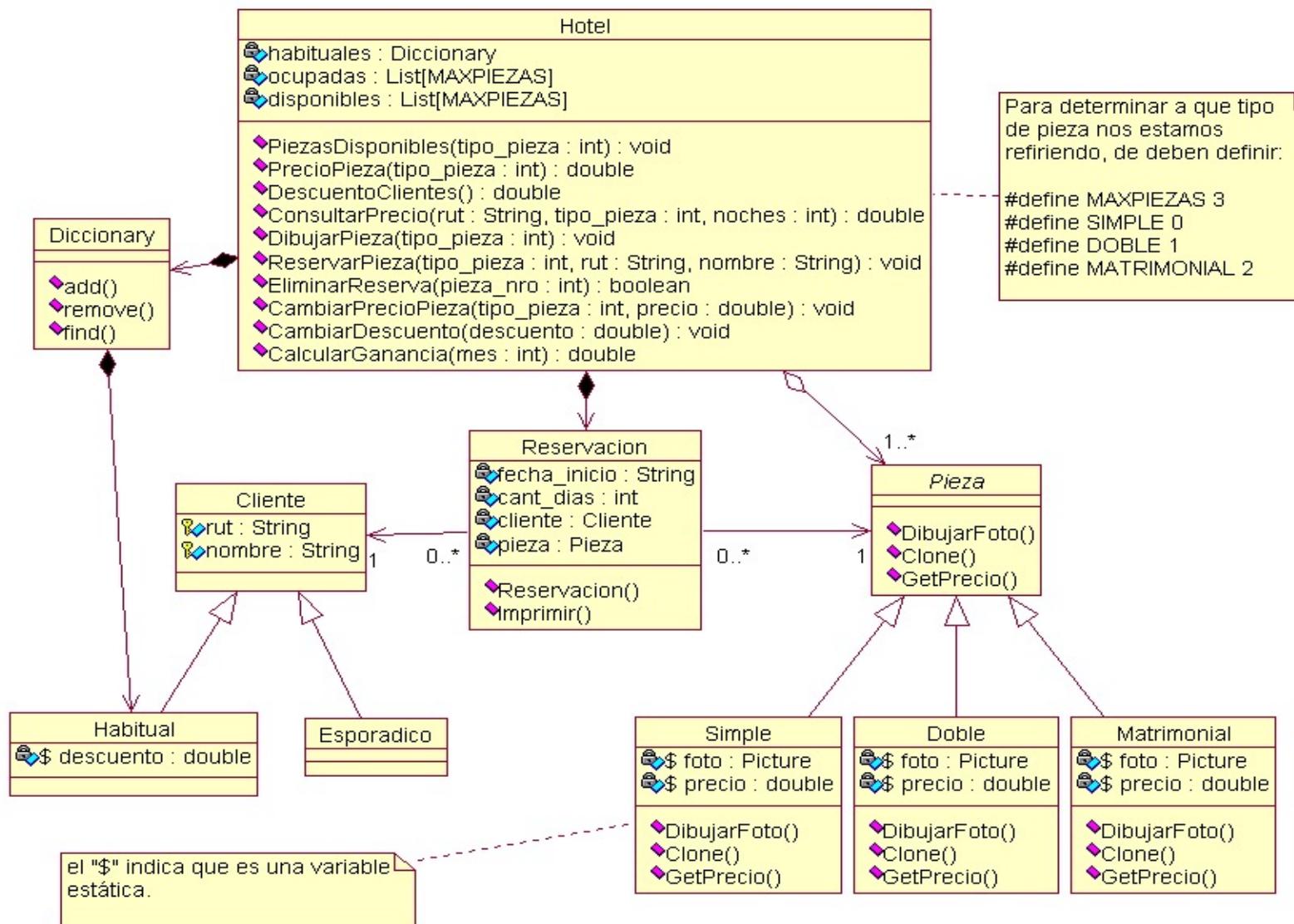
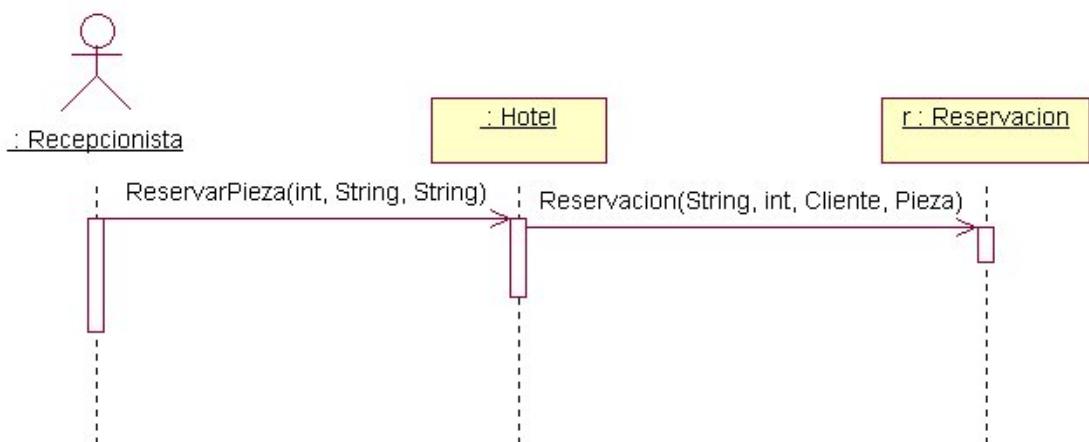


Diagrama de Interacción del escenario reservación



CLASIFICACIÓN DE LAS MANZANAS

En la actualidad el cultivo a gran escala de manzanas de calidad se lleva a cabo con poca actividad en el país, siendo uno de los mercados menos desarrollados. Este tipo de cultivos se lleva a cabo de una forma muy artesanal.

Viendo la necesidad del agricultor de manzanas de establecer unos parámetros que le hagan más fácil su labor; se decide estudiar el problema de la clasificación de manzanas desde la perspectiva y enfoque que le brinda UML.

Presentando el problema planteado, se desarrollará una visión general vista a través de los siguientes diagramas UML:

- Diagrama de casos de uso
- Diagrama de clases
- Diagrama de actividades
- Diagrama de secuencia
- Diagrama de estados
- Diagrama de paquetes
- Diagrama de objetos
- Diagrama de colaboración

Diagrama de Casos de Uso

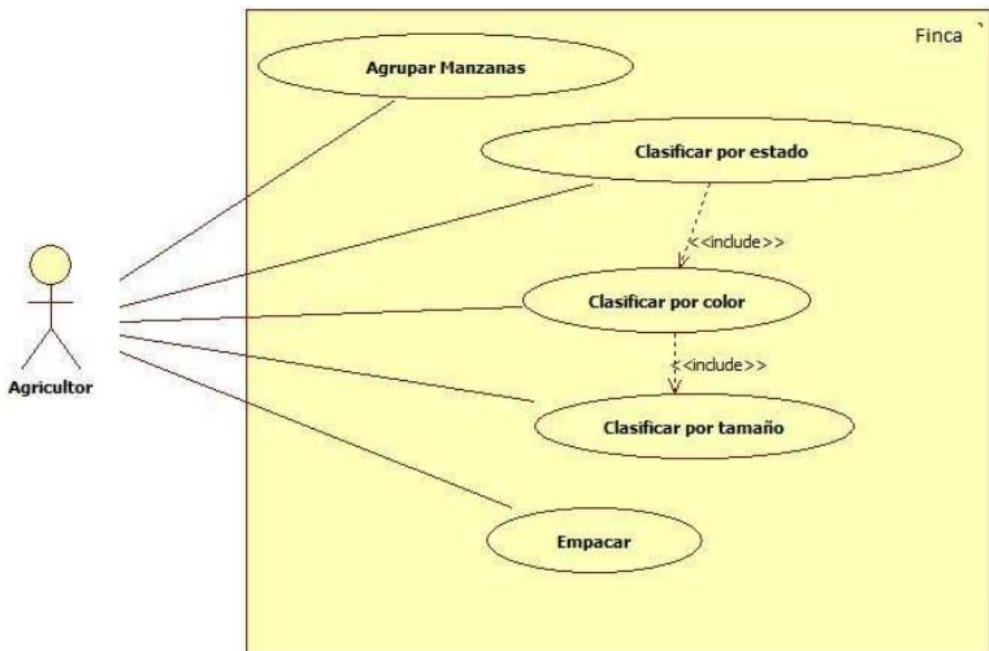


Diagrama de Clases

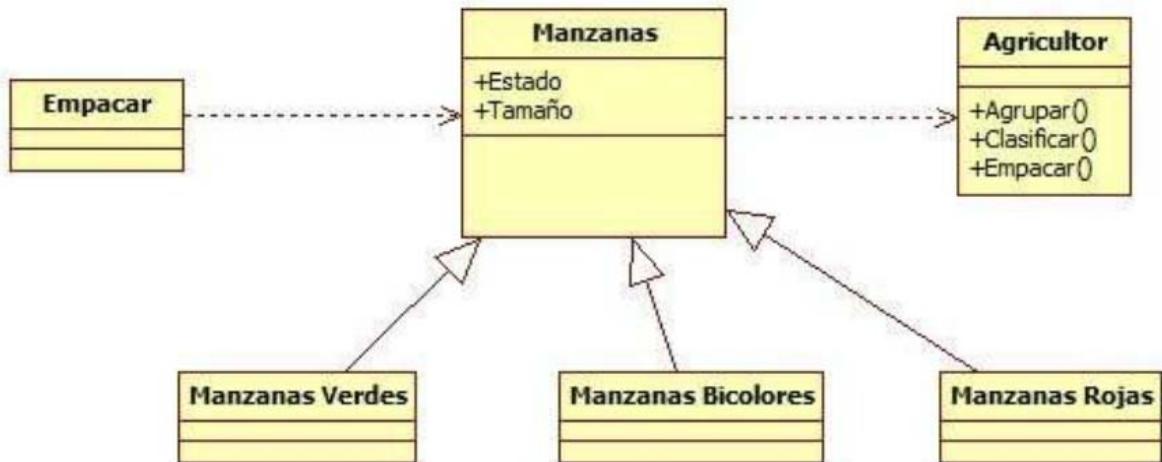


Diagrama de actividades

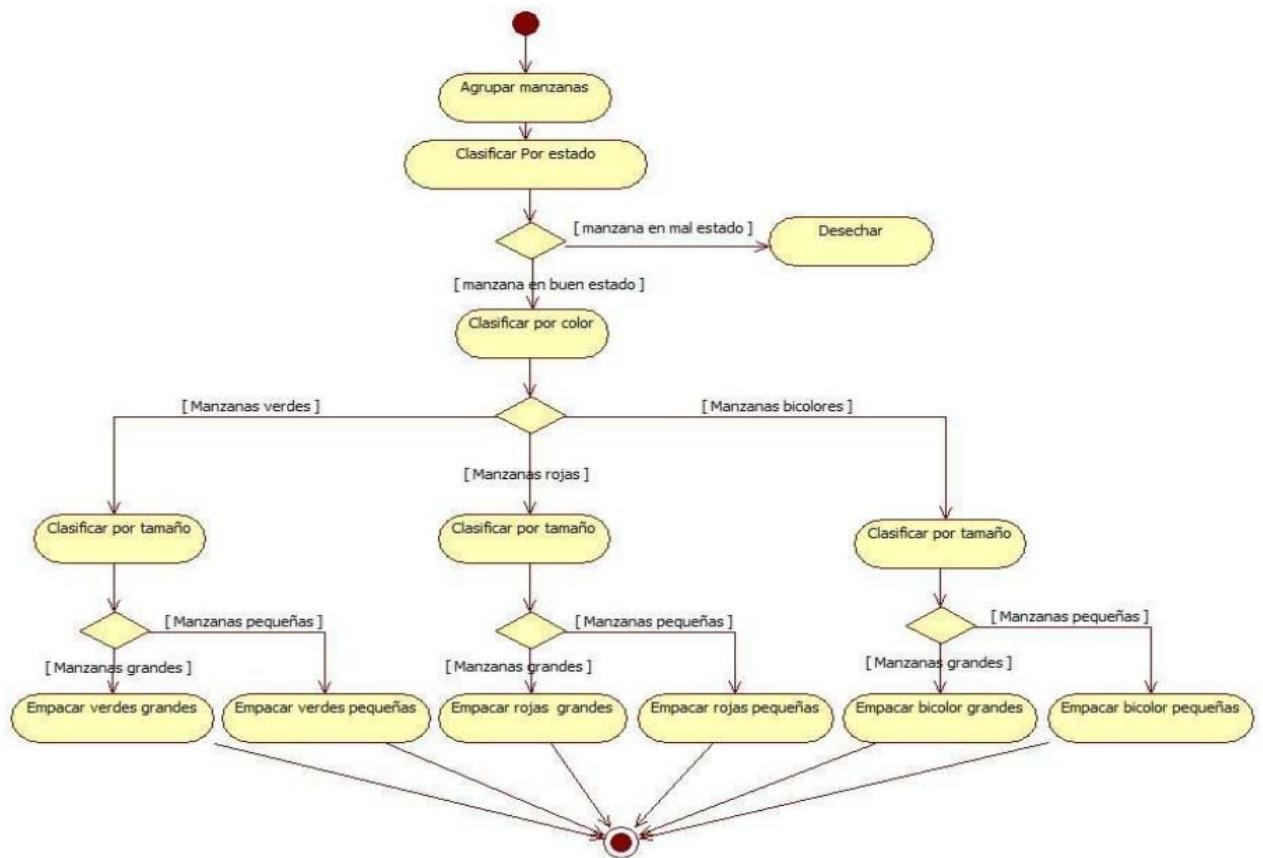


Diagrama de secuencia

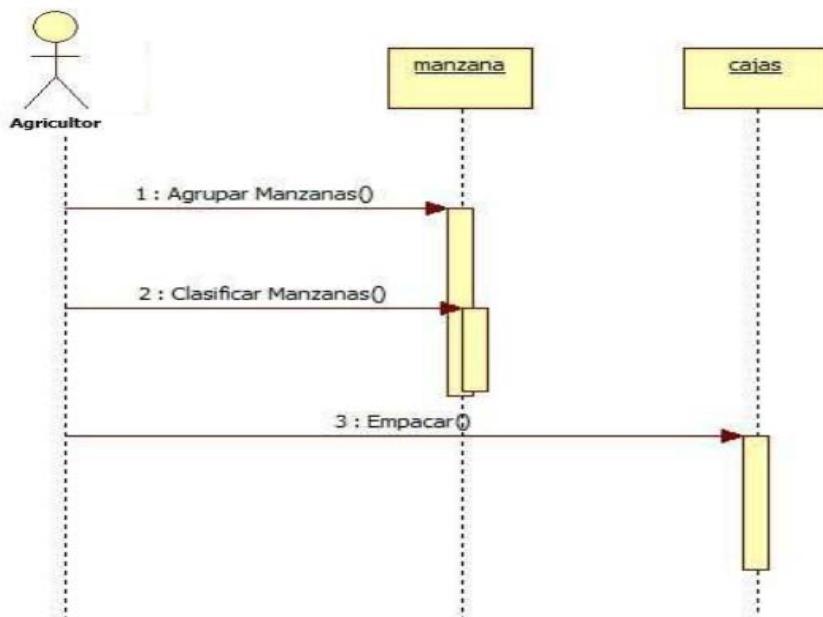


Diagrama de estados

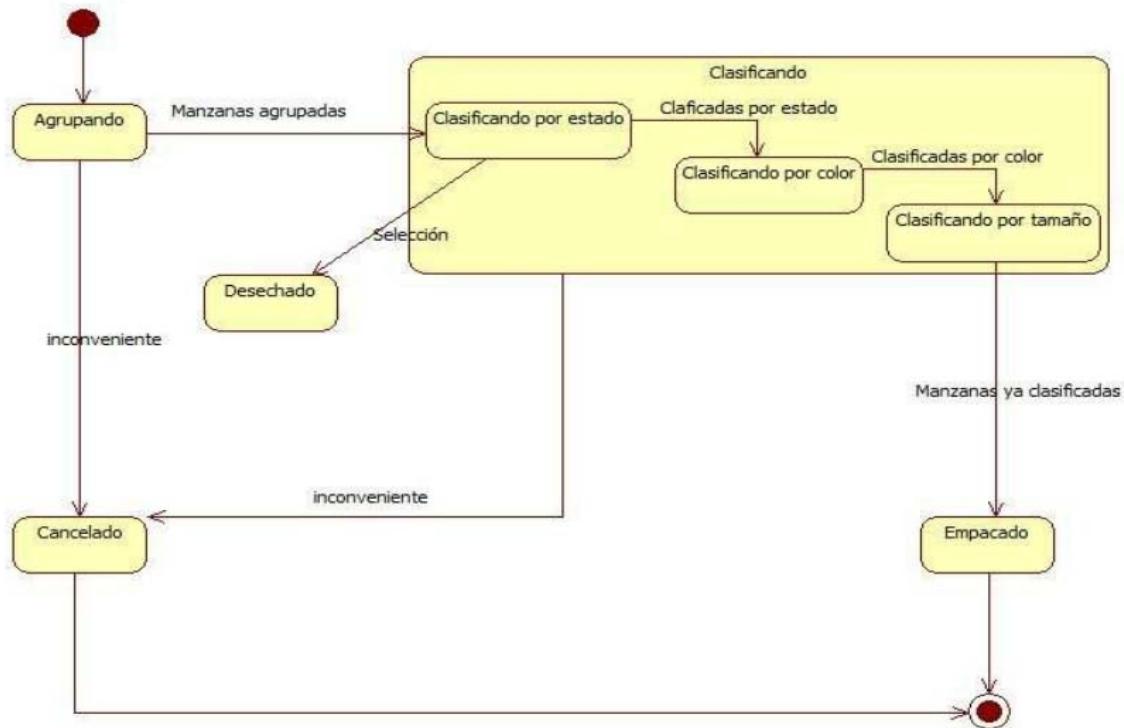


Diagrama de paquetes

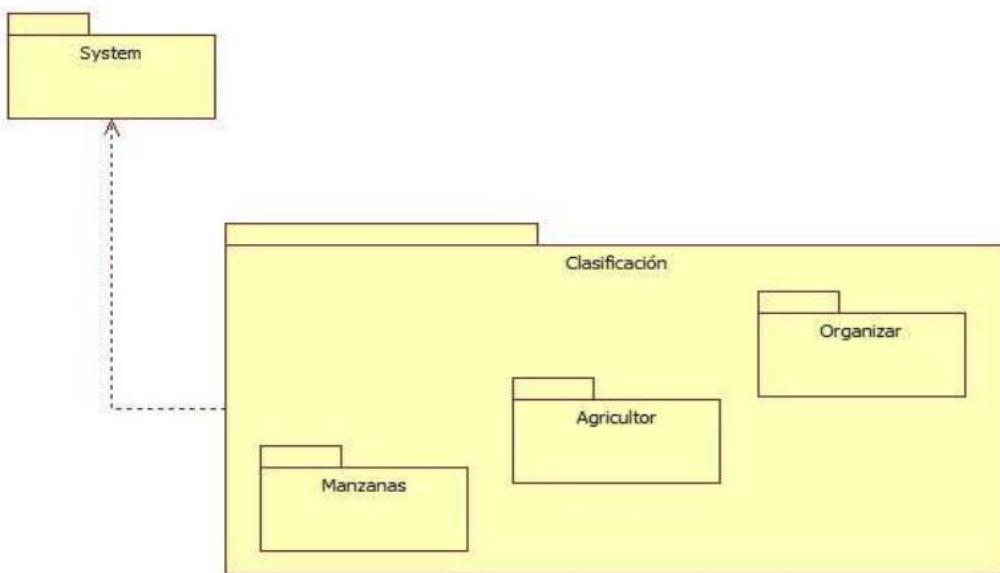


Diagrama de objetos

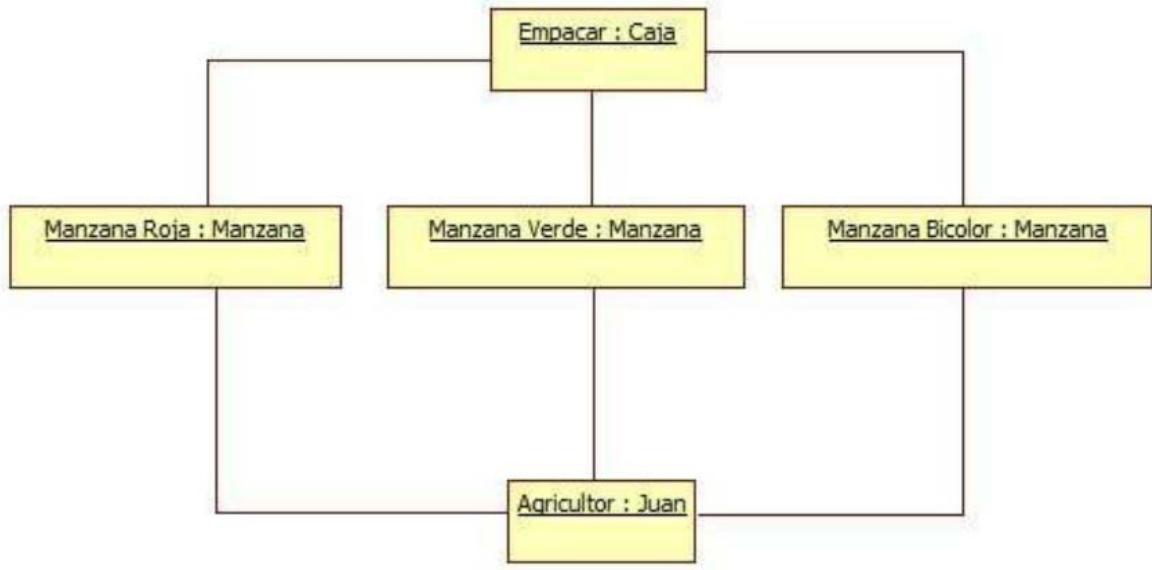
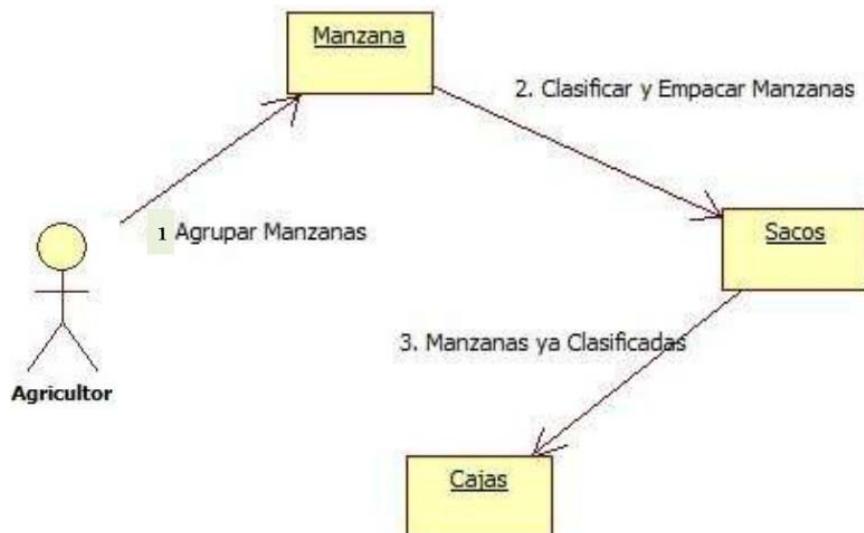


Diagrama de colaboración



BIBLIOGRAFIA

Booch Grady, Analisis y Diseño Orientado a Objetos con aplicaciones, 2da Edición, Editorial Addison Wesley Longman, Páginas: 629, ISBN: 968-444-352-8

Craig larman, UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado, Editorial Prentice Hall, ISBN: 8420534382

Debrauwer , Heyde, Fien Van Der, UML 2: Iniciación, Ejemplos Y Ejercicios Corregidos (RECURSOS INFORMATICOS), Primera Edición, 2005

Deitel Harvey M, Deitel Paul J, Como programar en C / C++, 2da edición, Prentice Hall, Páginas: 900, ISBN: 0-13-226119-7

Deitel Harvey, C++ cómo programar, 4ta edición, Editorial Pearson Prentice Hall,

Gómez Álvaro, Suarez Carlos, Sistemas de Información Herramientas prácticas para la gestión, 3ra Edición, Editorial Alfaomega, Paginas: 351, ISBN: 978-607-7854-45-6.

Kendall Kenneth, Kendall Julie, Análisis y Diseño de Sistemas, 8va Edición, Editorial Prentice Hall, Páginas: 566, ISBN: 978-607-32-0577-1

Martin Fowler, Kendall Scott, UML Gota a Gota, Editorial Addison Wesley

Neustadt Ila, Arlow Jim, Uml 2, 1ra edición, Editorial Anaya

REFERENCIAS A PÁGINAS WEB

Object Management Group

<http://www.omg.org/spec/UML/2.4/>

Recuperado por ultima vez el 5 de Octubre 2011

Tutorial de UML

<http://www.dcc.uchile.cl/~psalinas/uml/introduccion.html>

Recuperado por ultima vez el 5 de Octubre 2011