Universidad Nacional Autónoma de México

Facultad de Ingeniería

División de Educación Continua

Curso de Metodología Orientada a Objetos y UML

Duración: 26 hrs.
8:00 a 10:00 a.m.

Instructor: Ing. Dan Schmidt Valle

1. Introducción a la Metodología Orientada a Objetos 3
1.1. La Crisis del Software 3
1.2. La Metodología Orientada a Objetos 3
1.3. La Promesa de la Orientación a Objetos 4
2. Conceptos de la Metodología Orientada a Objetos 5
2.1. Conceptos (Objeto, Clase, Encapsulamiento, Herencia) 5
2.2. Ventajas de la Metodología Orientada a Objetos 8
3. El Lenguaje de Modelado Unificado (UML) 11
3.1. Principales beneficios de UML 11
3.2. UML: ¿Método o Lenguaje de Modelado? 12
3.3. Vistas de UML 12
3.4. Diagramas de UML 13
3.5. Elementos de Diagrama 14
3.6. Reglas 14
4. Fases del desarrollo de un sistema 15
4.1. Análisis de Requerimientos 15
4.2. Análisis 15
4.3. Diseño 15
4.4. Programación o Desarrollo 15
4.5. Pruebas 15
4.6. Un proceso iterativo e incremental 16
4.6.1. Desarrollo en pequeños pasos 16
4.6.2. ¿Por qué un desarrollo iterativo e incremental? 17
4.6.3. La aproximación iterativa dirigida por riesgos 18
4.6.4. Iteración genérica 19
4.6.5. Planeación de las iteraciones 20
5. Diagrama de Casos de Uso 22
5.1. Sistema 22
5.2. Actores 22
5.3. Encontrando actores en un diagrama de casos de uso 22
5.4. Casos de uso 22
5.5. Encontrando casos de uso 23
5.6. Relaciones 23
6. Diagrama de Clases 25

6.1. Clase 25
6.2. Atributos 26
6.3. Métodos 26
6.4. Relaciones entre Clases 27
6.4.1. Herencia (Especialización/Generalización) 28
6.4.2. Agregación 29
6.4.3. Asociación 30
6.4.4. Dependencia o Instanciación (uso) 30
6.4.5. Casos Particulares 31
7. Diagramas de Interacción 32
7.1. Secuencia 32
7.2. Colaboración 34
8. Diagramas de Comportamiento 35
8.1. Estados 35
8.2. Actividades 35
9. Diagramas de Implementación 37
9.1. Componentes 37
9.2. Despliegue 39
10. Bibliografía 42

1. Introducción a la Metodología Orientada a Objetos

1.1. La Crisis del Software

La crisis del software es el hecho de que el software que se construye no solamente no satisface los requerimientos ni las necesidades pedidos por el cliente, sino que además excede los presupuestos y los horarios de tiempos. Esta crisis fue el resultado de la introducción de la tercera generación del hardware. El hardware dejo de ser un impedimento para el desarrollo de la informática; redujo los costos y mejoró la calidad y eficiencia en el software producido. La crisis se caracterizó por los siguientes problemas:

- Imprecisión en la planificación del proyecto y estimación de los costos.
- Baja calidad del software.
- Dificultad de mantenimiento de programas con un diseño poco estructurado.

El software es solicitado para ejecutar las tareas demandantes de hoy y está presente en todos los sistemas que van desde los más sencillos hasta los de misión crítica. Las aplicaciones de software son complejas porque modelan la complejidad del mundo real. En estos días, las aplicaciones típicas son muy grandes y complejas para que un individuo las entienda y, por ello, es necesario gran tiempo implementar software. Por otra parte se exige que el software sea eficaz y barato tanto en el desarrollo como en la compra. También se requiere una serie de características como fiabilidad, facilidad de mantenimiento y de uso, eficiencia, etc.

Uno de los principales problemas en el desarrollo de software de hoy en día es que muchos proyectos empiezan la programación tan pronto se definen y concentran mucho de su esfuerzo en la escritura de código. Una de las técnicas generadas para solucionar este problema fue la *Metodología Orientada a Objetos*.

1.2. La Metodología Orientada a Objetos

A medida que se acercaban los años 80, la metodología orientada a objetos comenzaba a madurar como un enfoque de desarrollo de software. Empezamos a crear diseños de aplicaciones de todo tipo utilizando la forma de pensar orientada a los objetos e implementamos (codificamos) programas utilizando lenguajes y técnicas orientadas a los objetos.

La metodología orientada a objetos presenta características que lo hacen idóneo para el análisis, diseño y programación de sistemas; sin embargo, el análisis de requisitos, que es la relación entre la asignación de software al nivel del sistema y el diseño del software, se quedó atrás por lo que empezaron a surgir diferentes métodos de análisis y diseño orientado a objetos, entre los que destacan los métodos *Booch*, *OOSE* (Object Oriented Software Engineering), *Fusion* y *OMT* (Object Modeling Technique). Para poner fin a la "guerra de métodos" que se presentó en ese momento, se creó el *Lenguaje Unificado de Modelado* (UML).

Los métodos orientados a objetos centran su atención en objetos y clases:

- Un objeto es "una cosa".
- Una clase es "una definición de cosas".

Los métodos orientados a objetos crean modelos con los cuales:

- Se pueden crear clases y objetos.
- Definen su estructura, comportamiento y propósito.
- Definen la relación entre clases.
- Definen la relación entre objetos.

Estos modelos son utilizados a través del proyecto.

1.3. La Promesa de la Orientación a Objetos

- Los requerimientos y la arquitectura pueden ser capturados en una manera repetida y razonable.
- Los objetos modelan el comportamiento del mundo real, y son fáciles de entender.
- Los componentes pueden ser reutilizados a través de interfaces bien definidas.
- El cambio de requerimientos es fácil de soportar.
- Los costos de mantenimiento son reducidos considerablemente.

2. Conceptos de la Metodología Orientada a Objetos

2.1. Conceptos (Objeto, Clase, Encapsulamiento, Herencia)

La metodología orientada a objetos ha derivado de las metodologías anteriores a éste. Así como los métodos de diseño estructurado realizados guían a los desarrolladores que tratan de construir sistemas complejos utilizando algoritmos como sus bloques fundamentales de construcción, similarmente los métodos de diseño orientado a objetos han evolucionado para ayudar a los desarrolladores a explotar el poder de los lenguajes de programación basados en objetos y orientados a objetos, utilizando las clases y objetos como bloques de construcción básicos.

Actualmente el modelo de objetos ha sido influenciado por un número de factores no sólo de la *Programación Orientada a Objetos*, *POO* (*Object Oriented Programming*, *OOP* por sus siglas en inglés). Además, el modelo de objetos ha probado ser un concepto uniforme en las ciencias de la computación, aplicable no sólo a los lenguajes de programación sino también al diseño de interfaces de usuario, bases de datos y arquitectura de computadoras por completo. La razón de ello es, simplemente, que una orientación a objetos nos ayuda a hacer frente a la inherente complejidad de muchos tipos de sistemas.

Se define a un objeto como "*una entidad tangible que muestra alguna conducta bien definida*". Un objeto "*es cualquier cosa, real o abstracta, acerca de la cual almacenamos datos y los métodos que controlan dichos datos*".

Los objetos tienen una cierta "integridad" la cual no deberá ser violada. En particular, un objeto puede solamente cambiar estado, conducta, ser manipulado o estar en relación con otros objetos de manera apropiada a este objeto.

Actualmente, el *Análisis Orientado a Objetos* (*AOO*) va progresando como método de análisis de requisitos por derecho propio y como complemento de otros métodos de análisis. En lugar de examinar un problema mediante el modelo clásico de entrada-proceso-salida (flujo de información) o mediante un modelo derivado exclusivamente de estructuras jerárquicas de información, el AOO introduce varios conceptos nuevos. Estos conceptos nuevos le parecen inusuales a mucha gente, pero son bastante naturales.

Una *clase* es una plantilla para objetos múltiples con características similares. Las clases comprenden todas esas características de un conjunto particular de objetos. Cuando se escribe un programa en lenguaje orientado a objetos, no se definen objetos verdaderos sino se definen clases de objetos.

Una *instancia* de una clase es otro término para un objeto real. Si la clase es la representación general de un objeto, una instancia es su representación concreta. A menudo se utiliza indistintamente la palabra objeto o instancia para referirse, precisamente, a un objeto.

En los lenguajes orientados a objetos, cada clase está compuesta de dos cualidades: *atributos* (estado) y *métodos* (comportamiento o conducta). Los atributos son las características individuales que diferencian a un objeto de otro (ambos de la misma clase) y determinan la apariencia, estado u otras cualidades de ese objeto. Los atributos de un objeto incluyen información sobre su estado.

Los métodos de una clase determinan el comportamiento o conducta que requiere esa clase para que sus instancias puedan cambiar su estado interno o cuando dichas instancias son llamadas para realizar algo por otra clase o instancia. El comportamiento es la única manera en que las instancias pueden hacerse algo a sí mismas o tener que hacerles algo. Los atributos se encuentran en la parte interna mientras que los métodos se encuentran en la parte externa del objeto (**figura 1**).

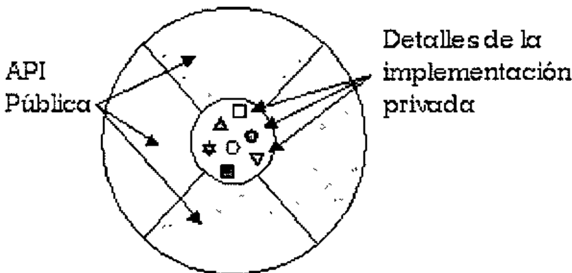


Figura 1. Representación visual de un objeto como componente de software.

Para definir el comportamiento de un objeto, se crean métodos, los cuales tienen una apariencia y un comportamiento igual al de las funciones en otros lenguajes de programación, los lenguajes estructurados, pero se definen dentro de una clase. Los métodos no siempre afectan a un solo objeto; los objetos también se comunican entre sí mediante el uso de métodos. Una clase u objeto puede llamar métodos en otra clase u objeto para avisar sobre los cambios en el ambiente o para solicitarle a ese objeto que cambie su estado.

Cualquier cosa que un objeto no sabe, o no puede hacer, es excluida del objeto. Además, como se puede observar de los diagramas, las variables del objeto se localizan en el centro o núcleo del objeto. Los métodos rodean y esconden el núcleo del objeto de otros objetos en el programa. Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama *encapsulamiento*. Típicamente, el encapsulamiento es utilizado para esconder detalles de la puesta en práctica no importantes de otros objetos. Entonces, los detalles de la puesta en práctica pueden cambiar en cualquier tiempo sin afectar otras partes del programa.

Esta imagen conceptual de un objeto —un núcleo de variables empaquetadas en una membrana protectora de métodos— es una representación ideal de un objeto y es el ideal por el que los diseñadores de sistemas orientados a objetos luchan. Sin embargo, no lo es todo: a menudo, por razones de eficiencia o la puesta en práctica, un objeto puede querer exponer algunas de sus variables o esconder algunos de sus métodos.

El encapsulamiento de variables y métodos en un componente de software ordenado es, todavía, una simple idea poderosa que provee dos principales beneficios a los desarrolladores de software:

- *Modularidad*, esto es, el código fuente de un objeto puede ser escrito, así como darle mantenimiento, independientemente del código fuente de otros objetos. Así mismo, un objeto puede ser transferido alrededor del sistema sin alterar su estado y conducta.

- *Ocultamiento de la información*, es decir, un objeto tiene una "interfaz publica" que otros objetos pueden utilizar para comunicarse con él. Pero el objeto puede mantener información y métodos privados que pueden ser cambiados en cualquier tiempo sin afectar a los otros objetos que dependan de ello.

Los objetos proveen el beneficio de la modularidad y el ocultamiento de la información. Las clases proveen el beneficio de la reutilización. Los programadores de software utilizan la misma clase, y por lo tanto el mismo código, una y otra vez para crear muchos objetos.

En las implantaciones orientadas a objetos se percibe un objeto como un paquete de datos y procedimientos que se pueden llevar a cabo con estos datos. Esto encapsula los datos y los procedimientos. La realidad es diferente: los atributos se relacionan al objeto o instancia y los métodos a la clase. ¿Por qué se hace así? Los atributos son variables comunes en cada objeto de una clase y cada uno de ellos puede tener un valor asociado, para cada variable, diferente al que tienen para esa misma variable los demás objetos. Los métodos, por su parte, pertenecen a la clase y no se almacenan en cada objeto, puesto que sería un desperdicio almacenar el mismo procedimiento varias veces y ello va contra el principio de reutilización de código.

Otro concepto muy importante en la metodología orientada a objetos es el de *herencia*. La herencia es un mecanismo poderoso con el cual se puede definir una clase en términos de otra clase; lo que significa que cuando se escribe una clase, sólo se tiene que especificar la diferencia de esa clase con otra, con lo cual, la herencia dará acceso automático a la información contenida en esa otra clase.

Con la herencia, todas las clases están arregladas dentro de una jerarquía estricta. Cada clase tiene una superclase (la clase superior en la jerarquía) y puede tener una o más subclases (las clases que se encuentran debajo de esa clase en la jerarquía). Se dice que las clases inferiores en la jerarquía, las clases hijas, heredan de las clases más altas, las clases padres.

Las subclases heredan todos los métodos y variables de las superclases. Es decir, en alguna clase, si la superclase define un comportamiento que la clase hija necesita, no se tendrá que redefinir o copiar ese código de la clase padre (**figura 2**).

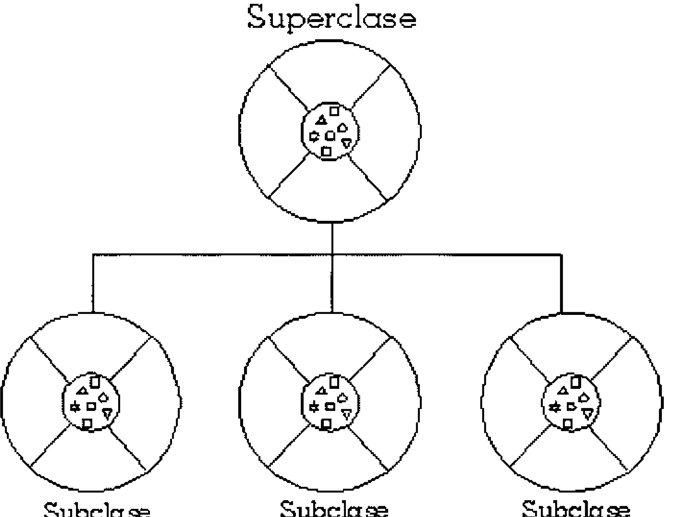


Figura 2. Clases, sub y super.

De esta manera, se puede pensar en una jerarquía de clase como la definición de conceptos demasiado abstractos en lo alto de la jerarquía y esas ideas se convierten en algo más concreto conforme se desciende por la cadena de la superclase.

Sin embargo, las clases hijas no están limitadas al estado y conducta provistos por sus superclases; pueden agregar variables y métodos además de los que ya heredan de sus clases padres. Las clases hijas pueden, también, sobrescribir los métodos que heredan por implementaciones especializadas para esos métodos. De igual manera, no hay limitación a un sólo nivel de herencia por lo que se tiene un árbol de herencia en el que se puede heredar varios niveles hacia abajo y mientras más niveles descienda una clase, más especializada será su conducta.

La herencia presenta los siguientes beneficios:

- Las subclases proveen conductas especializadas sobre la base de elementos comunes provistos por la superclase. A través del uso de herencia, los programadores pueden reutilizar el código de la superclase muchas veces.

- Los programadores pueden implementar superclases llamadas clases abstractas que definen conductas "genéricas". Las superclases abstractas definen, y pueden implementar parcialmente, la conducta pero gran parte de la clase no está definida ni implementada. Otros programadores concluirán esos detalles con subclases especializadas.

2.2. Ventajas de la Metodología Orientada a Objetos

En síntesis, algunas ventajas que presenta son:

- *Reutilización.* Las clases están diseñadas para que se reutilicen en muchos sistemas. Para maximizar la reutilización, las clases se construyen de manera que se puedan

adaptar a los otros sistemas. Un objetivo fundamental de las técnicas orientadas a objetos es lograr la reutilización masiva al construir el software.

- *Estabilidad.* Las clases diseñadas para una reutilización repetida se vuelven estables, de la misma manera que los microprocesadores y otros chips se hacen estables.

- *El diseñador piensa en términos del comportamiento de objetos y no en detalles de bajo nivel.* El encapsulamiento oculta los detalles y hace que las clases complejas sean fáciles de utilizar.

- *Se construyen clases cada vez más complejas.* Se construyen clases a partir de otras clases, las cuales a su vez se integran mediante clases. Esto permite construir componentes de software complejos, que a su vez se convierten en bloques de construcción de software más complejo.

- *Calidad.* Los diseños suelen tener mayor calidad, puesto que se integran a partir de componentes probados, que han sido verificados y pulidos varias veces.

- *Un diseño más rápido.* Las aplicaciones se crean a partir de componentes ya existentes. Muchos de los componentes están construidos de modo que se pueden adaptar para un diseño particular.

- *Integridad.* Las estructuras de datos (los objetos) sólo se pueden utilizar con métodos específicos. Esto tiene particular importancia en los sistemas cliente-servidor y los sistemas distribuidos, en los que usuarios desconocidos podrían intentar el acceso al sistema.

- *Mantenimiento más sencillo.* El programador encargado del mantenimiento cambia un método de clase a la vez. Cada clase efectúa sus funciones independientemente de las demás.

- *Una interfaz de pantalla sugestiva para el usuario.* Hay que utilizar una interfaz de usuario gráfica de modo que el usuario apunte a iconos o elementos de un menú desplegado, relacionados con los objetos. En determinadas ocasiones, el usuario puede ver un objeto en la pantalla. Ver y apuntar es más fácil que recordar y escribir.

- *Independencia del diseño.* Las clases están diseñadas para ser independientes del ambiente de plataformas, hardware y software. Utilizan solicitudes y respuestas con formato estándar. Esto les permite ser utilizadas en múltiples sistemas operativos, controladores de bases de datos, controladores de red, interfaces de usuario gráficas, etc. El creador del software no tiene que preocuparse por el ambiente o esperar a que éste se especifique.

- *Interacción.* El software de varios proveedores puede funcionar como conjunto. Un proveedor utiliza clases de otros. Existe una forma estándar de localizar clases e interactuar con ellas. El software desarrollado de manera independiente en lugares

ajenos debe poder funcionar en forma conjunta y aparecer como una sola unidad ante el usuario.

- *Computación Cliente-Servidor.* En los sistemas cliente-servidor, las clases en el software cliente deben enviar solicitudes a las clases en el software servidor y recibir respuestas. Una clase servidor puede ser utilizada por clientes diferentes. Estos clientes sólo pueden tener acceso a los datos del servidor a través de los métodos de la clase. Por lo tanto los datos están protegidos contra su corrupción.

- *Computación de distribución masiva.* Las redes a nivel mundial utilizarán directorios de software de objetos accesibles. El diseño orientado a objetos es la clave para la computación de distribución masiva. Las clases de una máquina interactúan con las de algún otro lugar sin saber donde residen tales clases. Ellas reciben y envían mensajes orientados a objetos en formato estándar.

- *Mayor nivel de automatización de las bases de datos.* Las estructuras de datos (los objetos) en las bases de datos orientadas a objetos están ligadas a métodos que llevan a cabo acciones automáticas. Una base de datos OO tiene integrada una *inteligencia*, en forma de métodos, en tanto que una base de datos relacional básica carece de ello.

- *Migración.* Las aplicaciones ya existentes, sean orientadas a objetos o no, pueden preservarse si se ajustan a un contenedor orientado a objetos, de modo que la comunicación con ella sea a través de mensajes estándar orientados a objetos.

- *Mejores herramientas CASE.* Las herramientas CASE (Computer Aided Software Engineering, Ingeniería de Software Asistida por Computadora) utilizarán las técnicas gráficas para el diseño de las clases y de la interacción entre ellas, para el uso de los objetos existentes adaptados a nuevas aplicaciones. Las herramientas deben facilitar el modelado en términos de eventos, formas de activación, estados de objetos, etc. Las herramientas OO del CASE deben generar un código tan pronto se definan las clases y permitir al diseñador utilizar y probar los métodos recién creados. Las herramientas se deben diseñar de manera que apoyen el máximo de creatividad y una continua afinación del diseño durante la construcción.

3. El Lenguaje de Modelado Unificado (UML)

En todas las disciplinas de la Ingeniería se hace evidente la importancia de los modelos ya que describen el aspecto y la conducta de "*algo*". Ese "*algo*" puede existir, estar en un estado de desarrollo o estar, todavía, en un estado de planeación. Es en este momento cuando los diseñadores del modelo deben investigar los requerimientos del producto terminado y dichos requerimientos pueden incluir áreas tales como funcionalidad, desempeño y confiabilidad. Además, a menudo, el modelo es dividido en un número de vistas, cada una de las cuales describe un aspecto específico del producto o sistema en construcción.

El modelado sirve no solamente para los grandes sistemas, aun en aplicaciones de pequeño tamaño se obtienen beneficios de modelado, sin embargo es un hecho que entre más grande y más complejo es el sistema, más importante es el papel de que juega el modelado por una simple razón: "El hombre hace modelos de sistemas complejos porque no puede entenderlos en su totalidad".

UML es una técnica para la especificación sistemas en todas sus fases. Nació en 1994 cubriendo los aspectos principales de todos los métodos de diseño antecesores y, precisamente, los padres de UML son Grady Booch, autor del método Booch; James Rumbaugh, autor del método OMT e Ivar Jacobson, autor de los métodos OOSE y Objectory. La versión 1.0 de UML fue liberada en Enero de 1997 y ha sido utilizado con éxito en sistemas construidos para toda clase de industrias alrededor del mundo: hospitales, bancos, comunicaciones, aeronáutica, finanzas, etc.

3.1. Principales beneficios de UML

El UML es una notación gráfica y textual, rica y expresiva, que permite textos si las gráficas pudiesen hacerse difíciles de entender. No tiene un proceso, no se aplica como una receta de cocina. Cada diagrama de UML brinda una vista distinta del sistema en cuestión. El UML no tiene un ciclo de vida. Se utiliza en un ciclo iterativo e incremental. Asimismo, las siguientes ventajas son de importancia:

- Mejores tiempos totales de desarrollo (de 50 % o más).

- Modelar sistemas (y no sólo de software) utilizando conceptos orientados a objetos.

- Establecer conceptos y artefactos ejecutables.

- Encaminar el desarrollo del escalamiento en sistemas complejos de misión crítica.

- Crear un lenguaje de modelado utilizado tanto por humanos como por máquinas.

- Mejor soporte a la planeación y al control de proyectos.

- Alta reutilización y minimización de costos.

3.2. UML: ¿Método o Lenguaje de Modelado?

UML es un lenguaje para hacer modelos y es independiente de los métodos de análisis y diseño. Existen diferencias importantes entre un método y un lenguaje de modelado. Un *método* es una manera explícita de estructurar el pensamiento y las acciones de cada individuo. Además, el método le dice al usuario qué hacer, cómo hacerlo, cuándo hacerlo y por qué hacerlo; mientras que el lenguaje de modelado carece de estas instrucciones. Los métodos contienen modelos y esos modelos son utilizados para describir algo y comunicar los resultados del uso del método.

Un modelo es expresado en un *lenguaje de modelado*. Un lenguaje de modelado consiste de vistas, diagramas, elementos de modelo — los símbolos utilizados en los modelos — y un conjunto de mecanismos generales o reglas que indican cómo utilizar los elementos. Las reglas son sintácticas, semánticas y pragmáticas (**figura 3**).

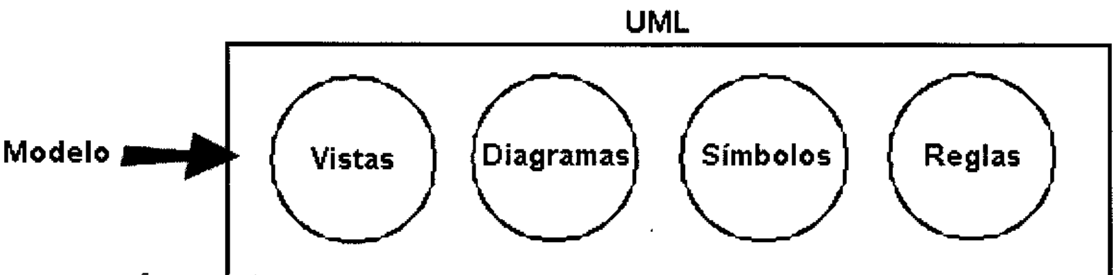


Figura 3. Lenguaje de modelado.

3.3. Vistas de UML

Las vistas muestran diferentes aspectos del sistema modelado. Una vista no es una gráfica, pero sí una abstracción que consiste en un número de diagramas y todos esos diagramas juntos muestran una "fotografía" completa del sistema. Las vistas también ligan el lenguaje de modelado a los métodos o procesos elegidos para el desarrollo. Las diferentes vistas que UML tiene son:

- Vista Use-Case: Una vista que muestra la funcionalidad del sistema como la perciben los actores externos.

- Vista Lógica: Muestra cómo se diseña la funcionalidad dentro del sistema, en términos de la estructura estática y la conducta dinámica del sistema.

- Vista de Componentes: Muestra la organización de los componentes de código.

- Vista Concurrente: Muestra la concurrencia en el sistema, dirigiendo los problemas con la comunicación y sincronización que están presentes en un sistema concurrente.

- Vista de Distribución: muestra la distribución del sistema en la arquitectura física con computadoras y dispositivos llamados *nodos*.

3.4. Diagramas de UML

Los diagramas son las gráficas que describen el contenido de una vista. UML tiene nueve tipos de diagramas que son utilizados en combinación para proveer todas las vistas de un sistema: diagramas de caso de uso, de clases, de objetos, de estados, de secuencia, de colaboración, de actividad, de componentes y de distribución. Una posible organización de estos diagramas es la que sigue:

- Diagrama de Casos de Uso
- Diagrama de Clases
- Diagrama de Objetos
- Diagramas de Comportamiento
  - Diagrama de Estados
  - Diagrama de Actividad
- Diagramas de Interacción
  - Diagrama de Secuencia
  - Diagrama de Colaboración
- Diagramas de implementación
  - Diagrama de Componentes
  - Diagrama de Despliegue

Asimismo, los diagramas pueden clasificarse en dos grandes rubros:

Modelo estático (estructural)

- Diagrama de despliegue
- Diagrama de componentes
- Diagrama de clases
- Diagrama de objetos

Modelo dinámico (comportamiento)

- Diagrama de estados

- Diagrama de actividades
- Diagrama de secuencia
- Diagrama de colaboración
- Diagrama de casos de uso

3.5. Elementos de Diagrama

Los conceptos utilizados en los diagramas son los elementos de modelo que representan conceptos comunes orientados a objetos, tales como clases, objetos y mensajes, y las relaciones entre estos conceptos incluyendo la asociación, dependencia y generalización. Un elemento de modelo es utilizado en varios diagramas diferentes, pero siempre tiene el mismo significado y simbología.

3.6. Reglas

Proveen comentarios extras, información o semántica acerca del elemento de modelo; además proveen mecanismos de extensión para adaptar o extender UML a un método o proceso específico, organización o usuario.

4. Fases del desarrollo de un sistema

Las fases del desarrollo de sistemas que soporta UML son: *Análisis de requerimientos*, *Análisis*, *Diseño*, *Programación* y *Pruebas*.

4.1. Análisis de Requerimientos

UML tiene casos de uso (use-cases) para capturar los requerimientos del cliente. A través del modelado de casos de uso, los actores externos que tienen interés en el sistema son modelados con la funcionalidad que ellos requieren del sistema (los casos de uso). Los actores y los casos de uso son modelados con relaciones y tienen asociaciones entre ellos o éstas son divididas en jerarquías. Los actores y casos de uso son descritos en un diagrama use-case. Cada use-case es descrito en texto y especifica los requerimientos del cliente: lo que él (o ella) espera del sistema sin considerar la funcionalidad que se implementará. Un análisis de requerimientos puede ser realizado también para procesos de negocios, no solamente para sistemas de software.

4.2. Análisis

La fase de análisis abarca las abstracciones primarias (clases y objetos) y mecanismos que están presentes en el dominio del problema. Las clases que se modelan son identificadas, con sus relaciones y descritas en un diagrama de clases. Las colaboraciones entre las clases para ejecutar los casos de uso también se consideran en esta fase a través de los modelos dinámicos en UML. Es importante notar que sólo se consideran clases que están en el dominio del problema (conceptos del mundo real) y todavía no se consideran clases que definen detalles y soluciones en el sistema de software, tales como clases para interfaces de usuario, bases de datos, comunicaciones, concurrencia, etc.

4.3. Diseño

En la fase de diseño, el resultado del análisis es expandido a una solución técnica. Se agregan nuevas clases que proveen de la infraestructura técnica: interfaces de usuario, manejo de bases de datos para almacenar objetos en una base de datos, comunicaciones con otros sistemas, etc. Las clases de dominio del problema del análisis son agregadas en esta fase. El diseño resulta en especificaciones detalladas para la fase de programación.

4.4. Programación o Desarrollo

En esta fase las clases del diseño son convertidas a código en un lenguaje de programación orientado a objetos. Cuando se crean los modelos de análisis y diseño en UML, lo más aconsejable es trasladar mentalmente esos modelos a código.

4.5. Pruebas

Normalmente, un sistema es tratado en pruebas de unidades, pruebas de integración, pruebas de sistema, pruebas de aceptación, etc. Las pruebas de unidades se realizan a clases individuales o a un grupo de clases y son típicamente ejecutadas por el programador. Las pruebas de integración integran componentes y clases en orden para

verificar que se ejecutan como se especificó. Las pruebas de sistema ven al sistema como una "caja negra" y validan que el sistema tenga la funcionalidad final que le usuario final espera. Las pruebas de aceptación conducidas por el cliente verifican que el sistema satisface los requerimientos y son similares a las pruebas de sistema.

4.6. Un proceso iterativo e incremental

Cada fase de desarrollo se compone por una serie de iteraciones e incrementos. El obtener un proceso iterativo e incremental (**figura 4**) se resume en pocas palabras con las tres claves del Proceso Unificado para el desarrollo de software:

- El sistema está dirigido por casos de usos.
- Se centra en una arquitectura.
- Tiene un desarrollo iterativo e incremental.

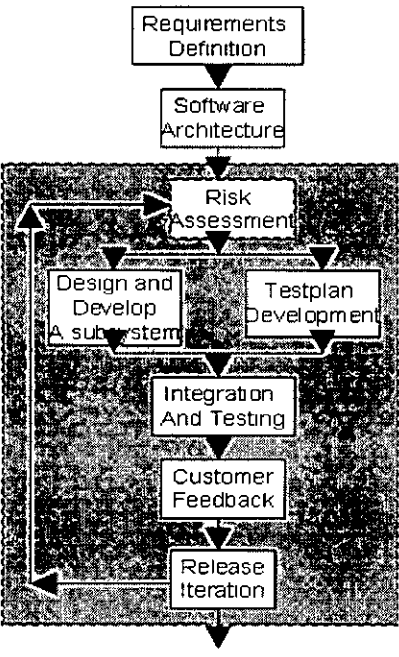


Figura 4. Desarrollo iterativo e incremental.

4.6.1. Desarrollo en pequeños pasos

En las primeras iteraciones se realiza:

- Determinación del ámbito del proyecto.
- Eliminación de riesgos críticos.
- Creación de la línea base de arquitectura.

- Se deben dominar los requisitos, el problema y los riesgos que pueden surgir.

En las iteraciones posteriores:

- Se reducen los riesgos menos graves.

- Se implementan componentes.

Se añaden incrementos hasta llegar a la versión extrema (para el cliente). El ciclo de vida de un proyecto se divide en mini proyectos (iteraciones), cada una compuesta por sus respectivos flujos de trabajo (requisito, análisis, diseño, implementación, prueba). Se les llama mini proyectos porque no es algo que el usuario haya pedido. El jefe de proyecto es quien se encarga de ordenar las iteraciones.

No es una iteración si el desarrollador pasa del ciclo de inicio al de elaboración:

- Sin resolver los riesgos más críticos.

- Sin establecer una línea base de la arquitectura.

- Sin implementar los casos de usos importantes.

Además de ser un atajo, se está construyendo un proyecto no fiable y no vale la pena que siga con él. La iteración NO es aleatoria. Sirve como herramienta para que los directores controlen el proyecto y reduce los riesgos que puedan amenazar al principio del ciclo de vida.

4.6.2. ¿Por qué un desarrollo iterativo e incremental?

- Atenuación de riesgos

Riesgo es una variable que pone en peligro o impide el éxito del proyecto. El Proceso Unificado reconoce los riesgos más importantes en las primeras 2 fases y reduce los mismos. Los que no, pueden poner en peligro el proyecto entero. Los riesgos importantes se tratan en las primeras fases, quedando muy pocas en la de construcción. El proyecto marcha sin inconvenientes hasta el final.

- Obtención de una arquitectura robusta

Es al final de la fase de elaboración donde se evalúa la arquitectura; si aún no está madura se trabaja en una nueva iteración; esto es posible ya que es muy poco lo que se in-vierte en esta fase y las fechas aún no están definidas.

- Gestión de requisitos cambiantes

Es más fácil para el usuario ver un sistema ejecutable en funcionamiento que leer cientos de páginas de documentos. Esto permite a que los usuarios opinen y sugieran modificar o agregar cosas que se nos pasó de largo. En método cascada los usuarios ven al sistema funcionando recién en la integración y prueba, y si desea cambiar algo deberán aumentar presupuesto y atrasar las fechas.

- Permitir cambios tácticos

Con método iterativo los directores se encargan de ver al final de cada iteración. Si hubo un incremento y se han resueltos los problemas; entonces autorizará a los desarrolladores a seguir con la siguiente iteración. Si el éxito fue parcial, se ampliará la iteración hasta poder cumplir con lo requerido. Si el resultado es negativo puede llegar a cancelarse el proyecto.

- Conseguir una iteración continua

Ya desde un principio se hacen frecuentes construcciones y, con éstas, aparecen los errores que se tratarán a lo largo de todo el proyecto. No habrá sorpresas para el final.

- Conseguir un aprendizaje temprano

Se ingresa gente nueva a medida que se pasa de una iteración a otra. Puede empezar con unas cinco a diez personas para pasar a veinticinco y hasta a cien. Los nuevos tienen una formación adecuada y trabajan con gente con experiencia, rápidamente se ajustan a la velocidad adecuada.

4.6.3. La aproximación iterativa dirigida por riesgos

Se analizan los riesgos, luego se priorizan y se organizan las iteraciones para acabar con los riesgos en una iteración temprana. Se manejan por iteraciones para no tener que tratar con todos los errores a la vez. Otras ventajas que se obtienen son:

- Tratar los requisitos pedido por los usuarios
- Obtener una arquitectura robusta.
- Tratar otros aspectos como rendimiento, disponibilidad, portabilidad: éstos se ven cuando se implementa y se prueba el software.

Las iteraciones alivian los riesgos técnicos. Hay cuatro formas de tratar a un riesgo según su prioridad:

- **Evitarlo**: Quizás se tenga que replanear el proyecto o hacer un cambio de requisitos.
- **Limitarlo**: Achicarlo para que afecte una parte pequeña del proyecto.

- **Atenuarlo**: Probar repetidas veces hasta ver si aparece o no.

- **Controlarlo**: Ver si el proyecto puede convivir con ésta. Caso contrario no se podrá continuar: algo que no es tan malo ya que se detectó en un principio y los gastos fueron mínimos.

4.6.4. Iteración genérica

Una iteración (**figura 5**) es un mini proyecto, donde se tiene como resultado una versión interna. Está compuesto por cinco flujos de trabajo: requisitos, análisis, diseño, desarrollo y pruebas. Los trabajadores y artefactos pueden trabajar en más de un flujo de trabajo.

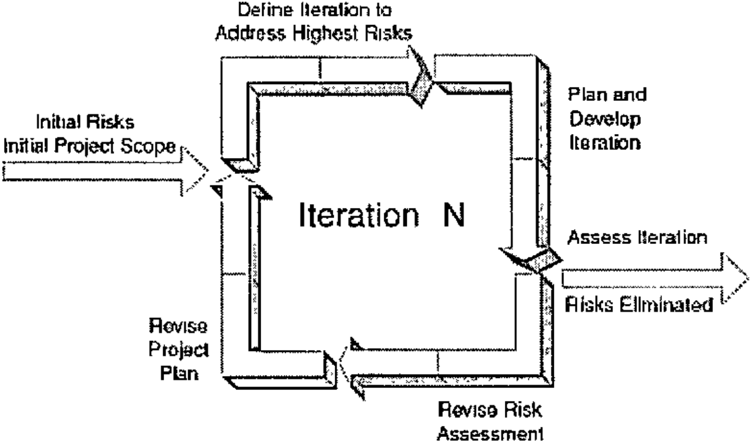


Figura 5. Iteración.

Las fases están divididas en N iteraciones. Descripción de cada fase:

- **Inicio**: Hacer análisis del negocio y reducir los riesgos más importantes.

- **Elaboración**: Obtener línea base de la arquitectura, capturar requisitos, reducir demás riesgos.

- **Construcción**: Desarrollar el sistema entero. Ofrecer funcionalidad operativa a clientes.

- **Transición**: Tener el producto preparado para la entrega. Se enseña a usuarios a utilizar el software.

Cada iteración se analiza cuando termina y se ve si cambiaron o aparecieron nuevos requisitos que modificarán a la siguiente iteración. Una herramienta muy útil a utilizar ocasionalmente al finalizar una iteración es la prueba de regresión, que sirve para saber si no se han estropeado iteraciones anteriores. Se aplica al antes de terminar con la iteración actual.

4.6.5. Planeación de las iteraciones

No se planea el proyecto entero en fase de inicio, sólo unos pasos. Es al final de la fase de elaboración, donde se tiene una base para planificar la mayor parte, cuando se lleva a cabo esta planeación. Dentro de las responsabilidades del planeador de iteraciones se encuentra el determinar la secuencia de iteraciones. Dado que los casos de usos establecen una meta y la arquitectura establece un patrón, la búsqueda de una secuencia idónea para las iteraciones es una de las piedras angulares para el exitoso desenvolvimiento del proceso generativo.

En las primeras iteraciones se conocen mejor los requisitos, riesgos y soluciones. Las iteraciones siguientes dan como resultado incrementos aditivos que terminan en una versión externa. La planificación y trabajo de una iteración empieza cuando la anterior se está por entregar.

El resultado de una iteración es un incremento. Definición de incremento: Diferencia entre la versión interna de la iteración anterior y la siguiente o entre dos líneas bases sucesivas. Hay más incrementos a medida que se acerca la fase de transición. La integración del último incremento se convierte en el sistema final.

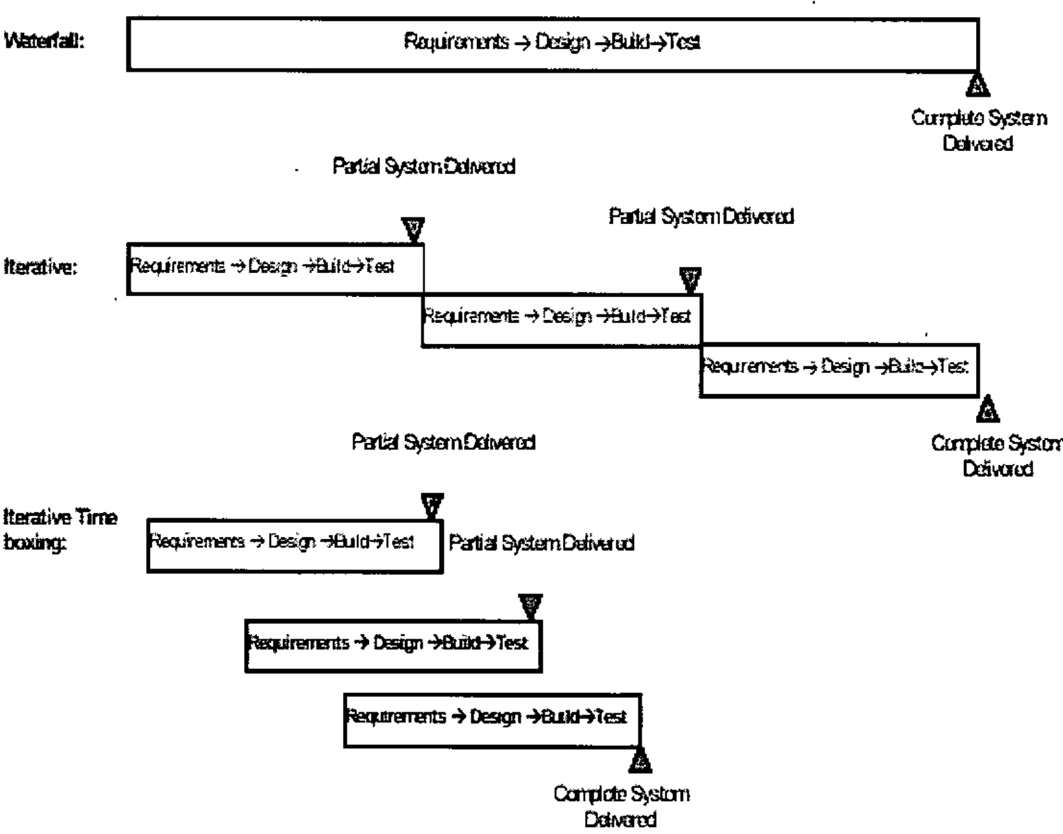


Figura 6. Comparación de procesos.

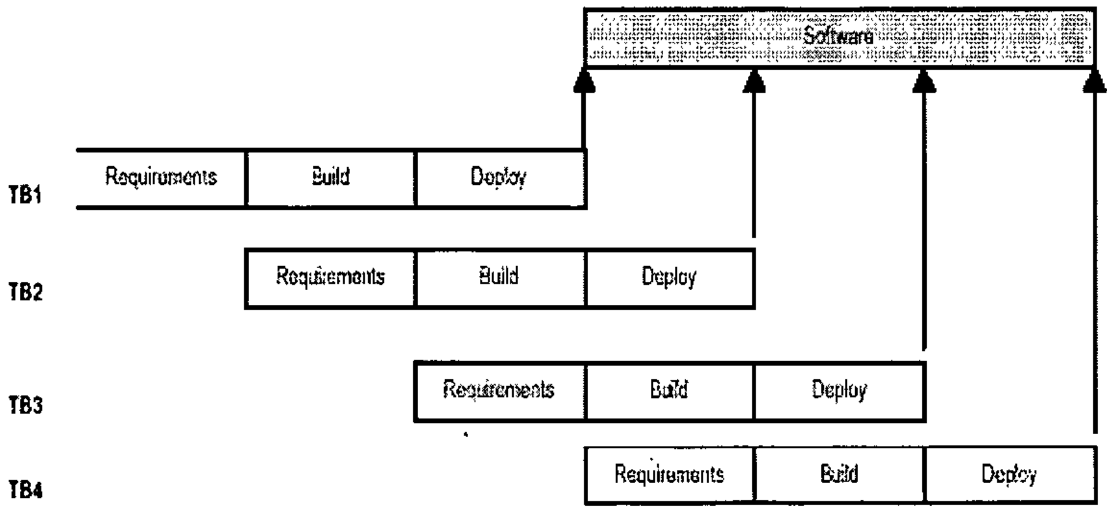


Figura 7. Comparación de procesos.

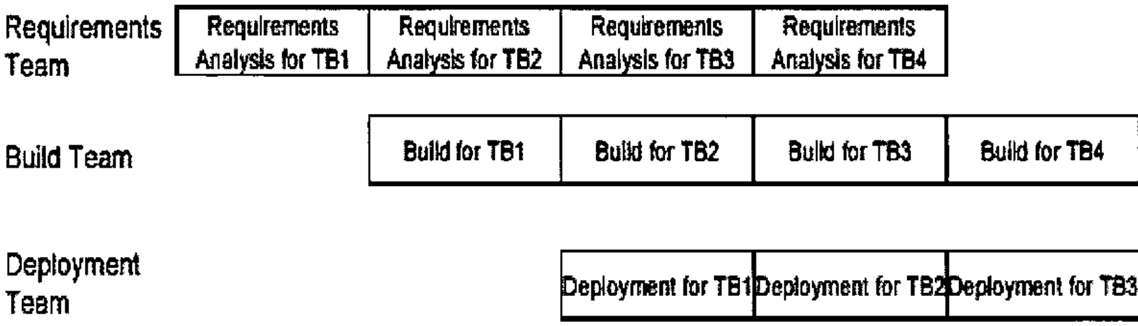


Figura 8. Tareas por equipo.

5. Diagrama de Casos de Uso

5.1. Sistema

Un sistema en un diagrama de caso de uso es descrito como una caja; el nombre del sistema aparece arriba o dentro de la caja. Ésta también contiene los símbolos para los casos de uso del sistema.

5.2. Actores

Un actor es alguien o algo que interactúa con el sistema; es quien utiliza el sistema. Por la frase "interactúa con el sistema" se debe entender que el actor envía a o recibe del sistema unos mensajes o intercambia información con el sistema. En pocas palabras, el actor lleva a cabo los casos de uso. Un actor puede ser una persona u otro sistema que se comunica con el sistema a modelar.

Un actor es un tipo (o sea, una clase), no es una instancia y representa a un rol. Gráficamente se representa con la figura de "stickman" (**figura 9**).

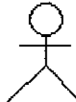


Figura 9. Stickman (El hombre de palitos).

5.3. Encontrando actores en un diagrama de casos de uso

Es posible obtener a los actores de un diagrama de casos de uso a través de las siguientes preguntas:

- ¿Quién utilizará la funcionalidad principal del sistema (actores primarios)?
- ¿Quién necesitará soporte del sistema para realizar sus actividades diarias?
- ¿Quién necesitará mantener, administrar y trabajar el sistema (actores secundarios)?
- ¿Qué dispositivos de hardware necesitará manejar el sistema?
- ¿Con qué otros sistemas necesitará interactuar el sistema a desarrollar?
- ¿Quién o qué tiene interés en los resultados (los valores) que el sistema producirá?

5.4. Casos de uso

Un caso de uso (**figura 10**) representa la funcionalidad completa tal y como la percibe un actor. Un caso de uso en UML es definido como un conjunto de secuencias de acciones que un sistema ejecuta y que permite un resultado observable de valores

para un actor en particular. Gráficamente se representan con una elipse y tiene las siguientes características:

- Un caso de uso siempre es iniciado por un actor.
- Un caso de uso provee valores a un actor.
- Un caso de uso es completo.

Figura 10. Caso de Uso.

5.5. Encontrando casos de uso

El proceso para encontrar casos de uso inicia encontrando al actor o actores previamente definidos. Por cada actor identificado, hay que realizar las siguientes preguntas:

- ¿Qué funciones del sistema requiere el actor? ¿Qué necesita hacer el actor?
- ¿El actor necesita leer, crear, destruir, modificar o almacenar algún tipo de información en el sistema?
- ¿El actor debe ser notificado de eventos en el sistema o viceversa? ¿Qué representan esos eventos en términos de funcionalidad?
- ¿El trabajo diario del actor podría ser simplificado o hecho más eficientemente a través de nuevas funciones en el sistema? (Comúnmente, acciones actuales del actor que no estén automatizadas)

Otras preguntas que nos ayudan a encontrar casos de uso pero que no involucran actores son:

- ¿Qué entradas/salidas necesita el sistema? ¿De dónde vienen esas entradas o hacia dónde van las salidas?
- ¿Cuáles son los mayores problemas de la implementación actual del sistema?

5.6. Relaciones

- **Asociación** ——————>

Es el tipo de relación más básica que indica la invocación desde un actor o caso de uso a otra operación (caso de uso). Dicha relación se denota con una flecha simple.

- **Dependencia o Instanciación**

Es una forma muy particular de relación entre clases, en la cual una clase depende de otra, es decir, se instancia (se crea). Dicha relación se denota con una flecha punteada.

- **Generalización**

  - Este tipo de relación es uno de los más utilizados, cumple una doble función dependiendo de su estereotipo, que puede ser de **Uso** (<<uses>>) o de **Herencia** (<<extends>>).

  - Este tipo de relación esta orientado exclusivamente para casos de uso (y no para actores).

  - **extends**: Se recomienda utilizar cuando un caso de uso es similar a otro (características).

  - **uses**: Se recomienda utilizar cuando se tiene un conjunto de características que son similares en más de un caso de uso y no se desea mantener copiada la descripción de la característica.

De lo anterior cabe mencionar que tiene el mismo paradigma en diseño y modelado de clases, en donde esta la duda clásica de usar o heredar.

6. Diagrama de Clases

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia, de uso y de contención.

Un diagrama de clases esta compuesto por los siguientes elementos:

- Clase: atributos, métodos y visibilidad.

- Relaciones: Herencia, Composición, Agregación, Asociación y Uso.

6.1. Clase

Es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una Casa, un Auto, una Cuenta Corriente, etc.).

En UML, una clase (**figura 11**) es representada por un rectángulo (figura 6) que posee tres divisiones:

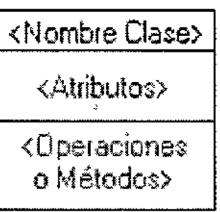


Figura 11. Clase.

En donde:

- **Superior**: Contiene el nombre de la Clase

- **Intermedio**: Contiene los atributos (o variables de instancia) que caracterizan a la Clase (pueden ser private, protected o public).

- **Inferior**: Contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: private, protected o public).

Para el ejemplo (**figura 12**), una Cuenta Corriente que posee como característica:

- Balance

Puede realizar las operaciones de:

- Depositar
- Girar
- y Balance

El diseño asociado es:

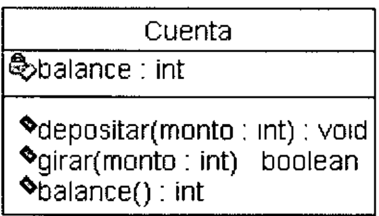


Figura 12. Clase Cuenta.

6.2. Atributos

Los atributos o características de una Clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno, estos son:

- **public** (+,): Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
- **private** (-,): Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden acceder).
- **protected** (#,): Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase además de las subclases que se deriven (ver herencia).

6.3. Métodos

Los métodos u operaciones de una clase son la forma en como ésta interactúa con su entorno, éstos pueden tener las características:

- **public** (+,): Indica que el método será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
- **private** (-,): Indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden acceder).

- **protected** (#,): Indica que el método no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase además de métodos de las subclases que se deriven (ver herencia).

6.4. Relaciones entre Clases

Ahora ya definido el concepto de Clase, es necesario explicar como se pueden interrelacionar dos o más clases (cada uno con características y objetivos diferentes). Antes es necesario explicar el concepto de cardinalidad de relaciones. En UML, la cardinalidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y éstas pueden ser:

a. **uno o muchos**: 1..* (1..n)

b. **0 o muchos**: 0..* (0..n)

c. **número fijo**: m (m denota el número).

6.4.1. Herencia (Especialización/Generalización)

Indica que una subclase hereda los métodos y atributos especificados por una Super Clase, por ende la Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la Super Clase (public y protected), ejemplo (**figura 13**):

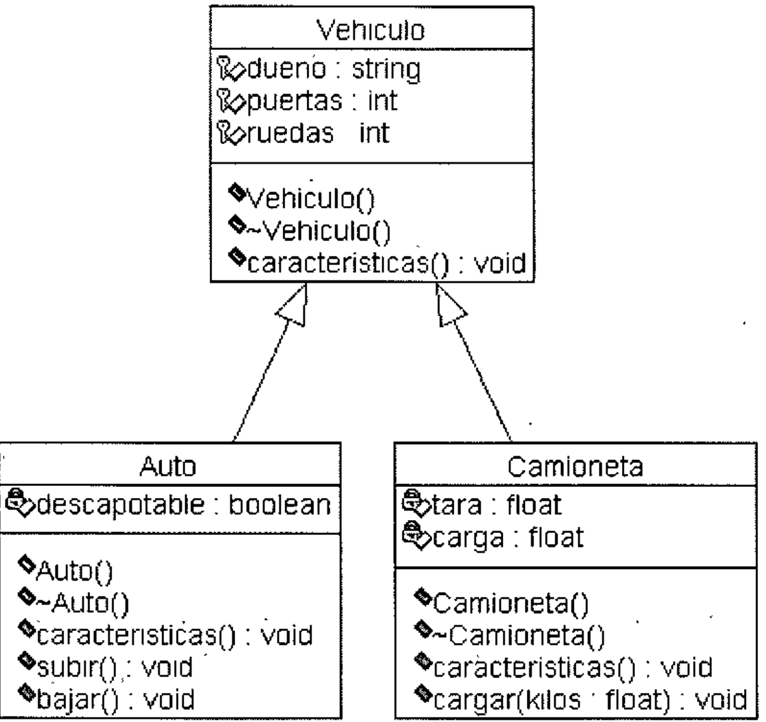


Figura 13. Ejemplo de herencia.

En la figura se especifica que Auto y Camión heredan de Vehículo, es decir, Auto posee las Características de Vehículo (Precio, VelMax, etc) además posee algo particular que es Descapotable, en cambio Camión también hereda las características de Vehiculo (Precio, VelMax, etc) pero posee como particularidad propia Acoplado, Tara y Carga.

Cabe destacar que fuera de este entorno, lo único "visible" es el método Características aplicable a instancias de Vehículo, Auto y Camión, pues tiene definición publica, en cambio atributos como Descapotable no son visibles por ser privados.

6.4.2. Agregación

Para modelar objetos complejos, n bastan los tipos de datos básicos que proveen los lenguajes: enteros, reales y secuencias de caracteres. Cuando se requiere componer objetos que son instancias de clases definidas por el desarrollador de la aplicación, tenemos dos posibilidades:

- **Por Valor**: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comúnmente llamada **Composición** (el Objeto base se construye a partir del objeto incluido, es decir, es "parte/todo").

- **Por Referencia**: Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada **Agregación** (el objeto base utiliza al incluido para su funcionamiento).

Un ejemplo **(figura 14)** es el siguiente:

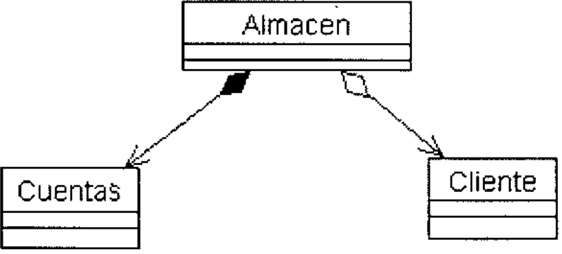


Figura 14. Relaciones por valor y por referencia.

En donde se destaca que:

- Un Almacén posee Clientes y Cuentas (los rombos van en el objeto que posee las referencias).

- Cuando se destruye el Objeto Almacén también son destruidos los objetos Cuenta asociados, en cambio no son afectados los objetos Cliente asociados.

- La composición (por Valor) se destaca por un rombo relleno.

- La agregación (por Referencia) se destaca por un rombo transparente.

La flecha en este tipo de relación indica la navegabilidad del objeto referenciado. Cuando no existe este tipo de particularidad la flecha se elimina.

6.4.3. Asociación

La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre si. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro. Ejemplo (**figura 15**):

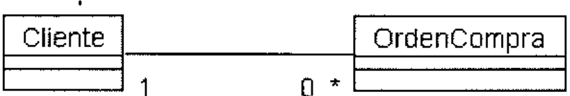
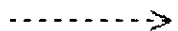


Figura 15. Asociación.

Un cliente puede tener asociadas muchas Ordenes de Compra, en cambio una orden de compra solo puede tener asociado un cliente.

6.4.4. Dependencia o Instanciación (uso)



Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase). Se denota por una flecha punteada. El uso más particular de este tipo de relación es para denotar la dependencia que tiene una clase de otra, como por ejemplo (**figura 16**) una aplicación grafica que instancia una ventana (la creación del Objeto Ventana esta condicionado a la instanciación proveniente desde el objeto Aplicacion):

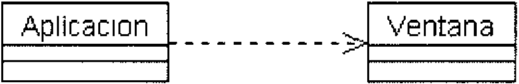


Figura 16. Dependencia.

Cabe destacar que el objeto creado (en este caso la Ventana gráfica) no se almacena dentro del objeto que lo crea (en este caso la Aplicación).

6.4.5. Casos Particulares

Clase Abstracta

Una clase abstracta (**figura 17**) se denota con el nombre de la clase y de los métodos con letra "itálica". Esto indica que la clase definida no puede ser instanciada pues posee métodos abstractos (aún no han sido definidos, es decir, sin implementación). La única forma de utilizarla es definiendo subclases, que implementan los métodos abstractos definidos.

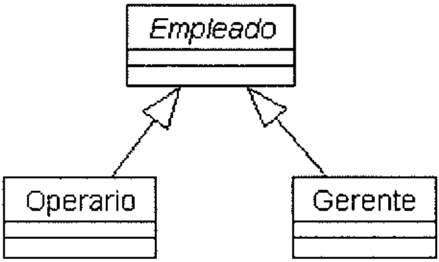


Figura 17. Clase abstracta.

Clase parametrizada

Una clase parametrizada (**figura 18**) se denota con un subcuadro en el extremo superior de la clase, en donde se especifican los parámetros que deben ser pasados a la clase para que esta pueda ser instanciada. El ejemplo más típico es el caso de un Diccionario en donde una llave o palabra tiene asociado un significado, pero en este caso las llaves y elementos pueden ser genéricos. La calidad de genérico puede venir dada de una plantilla (como en el caso de C++) o bien de alguna estructura predefinida (especialización a través de clases). En el ejemplo no se especificaron los atributos del Diccionario, pues ellos dependerán exclusivamente de la implementación que se le quiera dar.

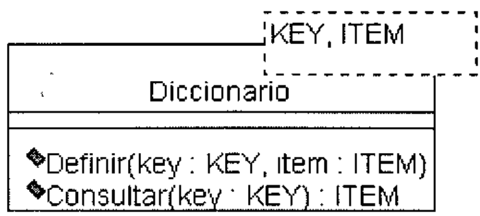


Figura 18. Clase parametrizada.

7. Diagramas de Interacción

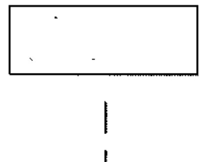
7.1. Secuencia

El diagrama de secuencia representa la forma en como un Cliente (Actor) u Objetos (Clases) se comunican entre si en petición a un evento. Esto implica recorrer toda la secuencia de llamadas, de donde se obtienen las responsabilidades claramente. Dicho diagrama puede ser obtenido de dos partes, desde el Diagrama Estático de Clases o el de Casos de Uso (son diferentes).

Los componentes de un diagrama de interacción son:

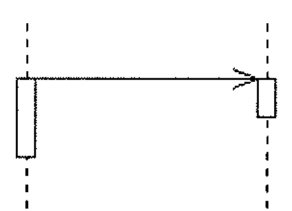
- Un Objeto o Actor.
- Mensaje de un objeto a otro objeto.
- Mensaje de un objeto a si mismo.

Objeto/Actor



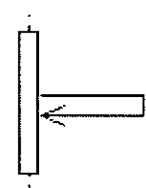
El rectángulo representa una instancia de un Objeto en particular, y la línea punteada representa las llamadas a métodos del objeto.

Mensaje a Otro Objeto



Se representa por una flecha entre un objeto y otro, representa la llamada de un método (operación) de un objeto en particular.

Mensaje al Mismo Objeto



No solo llamadas a métodos de objetos externos pueden realizarse, también es posible visualizar llamadas a métodos desde el mismo objeto en estudio.

En el presente ejemplo, tenemos el diagrama de interacción proveniente del siguiente modelo estático (**figura 19**):

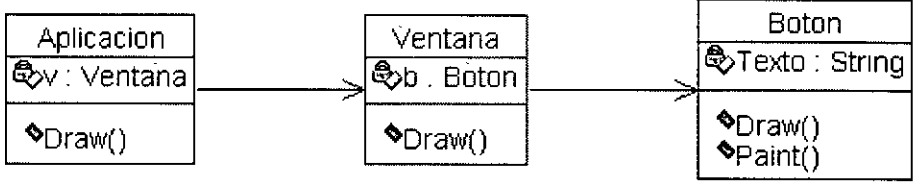


Figura 19. Diagrama estático.

Aquí se representa una aplicación que posee una Ventana gráfica, y ésta a su vez posee internamente un botón.

Entonces el diagrama de secuencia (**figura 20**) para dicho modelo es:

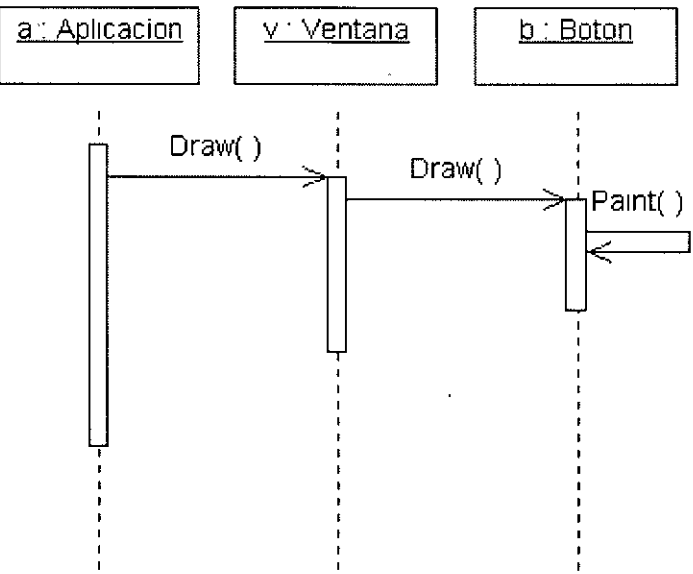


Figura 20. Diagrama de secuencia.

En donde se hacen notar las sucesivas llamadas a Draw() (entre objetos) y la llamada a Paint() por el objeto Botón.

En este diagrama se observa que sólo se consideran algunos objetos y es importante aclarar que estos no serán todos los objetos a considerar dentro del sistema, ya que todavía es posible agregar nuevos objetos que no se habían considerado en el dominio del análisis así como los objetos técnicos, como se mencionó anteriormente. Los objetos considerados se representan en rectángulos con el nombre subrayado y cada uno cuenta con su línea de vida vertical que muestra la vida del objeto.

Nótese que hasta el momento no se ha hecho énfasis en el orden que se pretende para estos diagramas. No obstante, siempre es necesario tomar en cuenta procesos completos para cada diagrama. Para completar una secuencia es perentorio regresar al

punto de partida. Es decir, cada vez habrá que asegurar que el camino de retorno para toda secuencia sea ininterrumpido hasta el objeto que la haya iniciado.

7.2. Colaboración

Así mismo, se cuenta con el diagrama de colaboración (**figura 21**), el cual se centra tanto en las interacciones y las ligas entre un conjunto de objetos colaborando entre ellos (una liga es una instancia de una asociación). Ambos, el diagrama de secuencia y el diagrama de colaboración, muestran interacciones, pero el diagrama de secuencia se centra en el tiempo mientras que el diagrama de colaboración se centra en el espacio. Las ligas muestran los objetos actuales y cómo ellos se relacionan unos con otros. Así como los diagramas de secuencia, los diagramas de colaboración pueden ser utilizados para ilustrar la ejecución de una operación, una ejecución de un use-case o simplemente un escenario de interacción dentro del sistema. En este diagrama también se representa a los objetos en cajas rectangulares y con el nombre subrayado. Las ligas se dibujan con líneas y se puede agregar una etiqueta para un mensaje y un número que define la secuencia de las ligas.

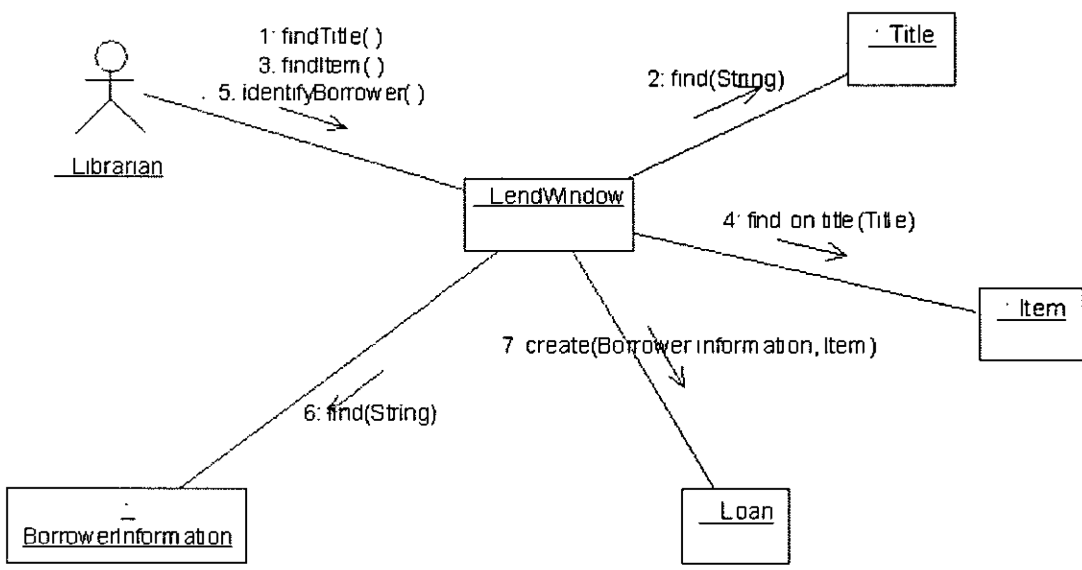


Figura 21. Diagrama de colaboración.

8. Diagramas de Comportamiento

8.1. Estados

Posteriormente se realiza el diagrama de estados (**figura 22**) el cual captura el ciclo de vida de los objetos, subsistemas y sistemas. Dicho diagrama determina los estados que un objeto puede tener y cómo los eventos afectan esos estados a través del tiempo. Un diagrama de estado debe abarcar todas las clases que tengan estados y conducta definidos claramente.

Todos los objetos tienen un estado y éste es el resultado de actividades previas ejecutadas por el objeto. Ese estado está determinado por los valores de los atributos de este objeto y sus relaciones con otros objetos. Una clase puede tener un atributo que especifique el estado, o el estado puede ser determinado por los valores de los atributos "normales" del objeto.

Cada diagrama posee un inicio y un fin. Si bien estos no pueden considerarse estados en sí mismos, es indispensable conceptualizarlos como la generación y destrucción de un objeto, proceso extremadamente necesario en cualquier diseño.

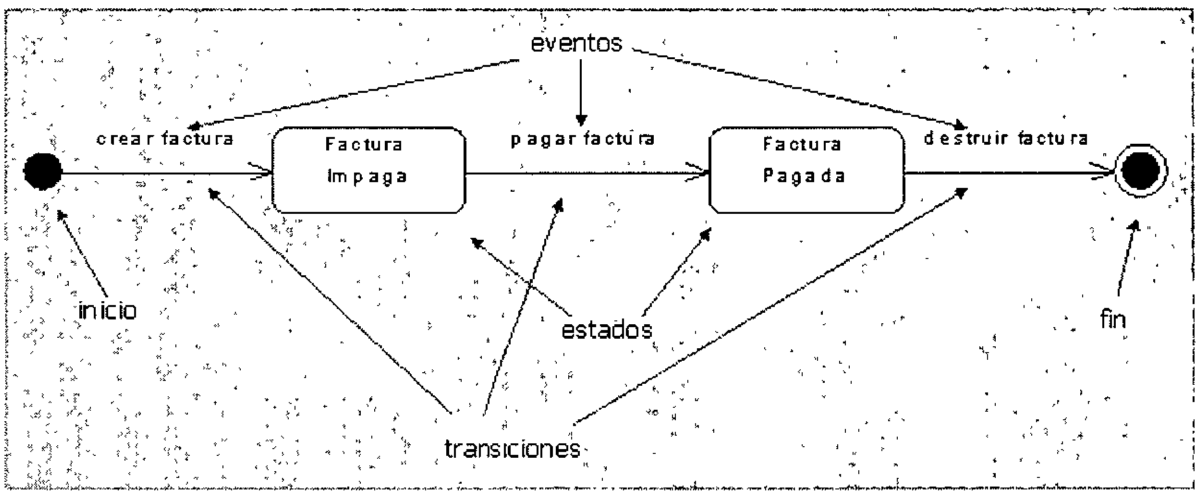


Figura 22. Diagrama de estados.

8.2. Actividades

Un diagrama de actividades (**figura 23**) puede considerarse como un caso especial de un diagrama de estados en el cual casi todos los estados son estados acción (identifican una acción que se ejecuta al estar en él) y casi todas las transiciones evolucionan al término de dicha acción (ejecutada en el estado anterior). Un diagrama de actividades puede dar detalle a un caso de uso, un objeto o un mensaje en un objeto. Permiten representar transiciones internas al margen de las transiciones o eventos externos. En la figura 5.7 se presenta un ejemplo de diagrama de actividades para un mensaje de un objeto.

La interpretación de un diagrama de actividades depende de la perspectiva considerada: en un diagrama conceptual, la actividad es alguna tarea que debe ser realizada; en un diagrama de especificación o de implementación, la actividad es un método de una clase. Generalmente se suelen utilizar para modelar los pasos de un algoritmo.

Un estado de acción representa un estado con acción interna, con por lo menos una transición que identifica la culminación de la acción (por medio de un evento implícito). No deben tener transiciones internas ni transiciones basadas en eventos, ya que si fuera este el caso, se representaría con un diagrama de estados. Los estados de acción se representan por un rectángulo con bordes redondeados, y permiten modelar un paso dentro de un algoritmo. Las flechas dirigidas entre estados de acción representan transiciones con evento implícito que, en el caso de decisiones, pueden tener una condición o guarda asociada (que al igual que en los diagramas de estado evalúa a verdadero o a falso).

Las decisiones se representan mediante una transición múltiple que sale de un estado y donde cada camino tiene una etiqueta distinta. Se representa mediante un rombo al cual llega la transición del estado origen y del cual salen las múltiples transiciones de los estados destino. En un diagrama de actividades también pueden existir barras de sincronización, a las que se encuentran asociadas varios caminos salientes. Cada camino saliente se dirige a una actividad, realizándose dichas actividades en paralelo. Esto quiere decir que el orden en que se realicen dichas actividades es irrelevante, siendo válido cualquier orden entre ellas.

Otro elemento característico de los diagramas de actividades es el carril (swim lane). Este permite separar los distintos sectores, rubros o procesos por los que atravesará una parte del sistema durante su accionar. Pueden existir tantos carriles como sea necesario, pero la conformación de un diagrama lógico y comprensible se complica a medida que se añaden los mismos.

Dado que el diagrama de actividades permite expresar el orden en que se realizan las cosas, resulta adecuado para el modelado de organizaciones y el de programas concurrentes (permiten representa gráficamente los hilos de ejecución). Como la mayoría de las técnicas de modelado de comportamiento, los diagramas de actividades tienen sus puntos fuertes y sus puntos débiles, de forma que es necesario utilizarlos en combinación con otras técnicas. Su principal aportación al modelado del comportamiento es que soportan el comportamiento paralelo, lo que resulta adecuado para el modelado de flujo de trabajo y programación multihilos. Por contra, su principal desventaja es que no muestran de una forma clara los enlaces existentes entre las acciones y los objetos, siendo mucho más apropiado para ello los diagramas de interacción.

En general resulta adecuado utilizar diagramas de actividades para:

- Análisis de casos de uso: Durante el análisis de los casos de uso no estamos interesados en asociar acciones a objetos, sino en entender qué

acciones se necesitan llevar a cabo y cuales son las dependencias en el comportamiento.

- Comprensión del flujo de trabajo a lo largo de diferentes casos de uso.
- Modelado de aplicaciones multihilos.

Por contra, resultan en general del todo inadecuados a la hora de mostrar la colaboración entre objetos y la evolución del comportamiento de los objetos durante su tiempo de vida.

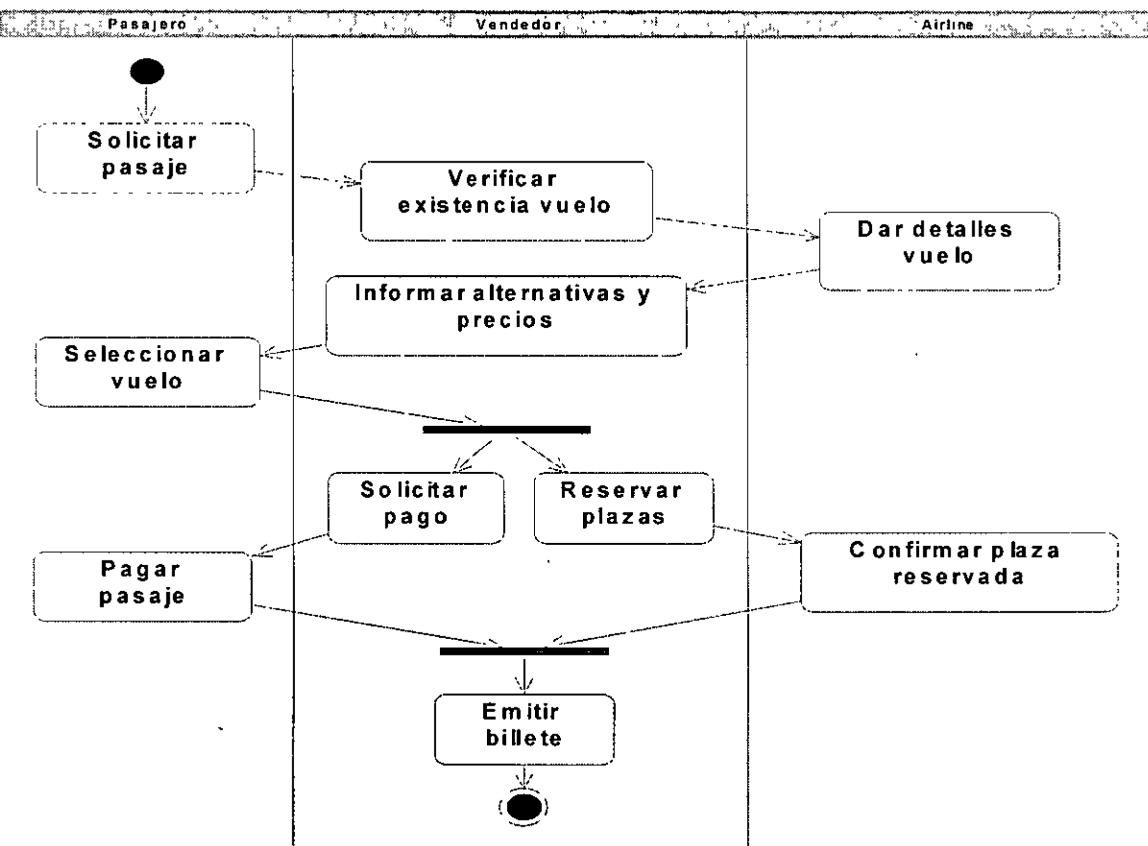


Figura 23. Diagrama de actividades con carriles.

9. Diagramas de Implementación

9.1. Componentes

Lo que distingue a un diagrama de componentes (**figura 24**) de otros tipos de diagramas es su contenido. Normalmente contienen componentes, interfaces y relaciones entre ellos. Y como todos los diagramas, también puede contener paquetes utilizados para agrupar elementos del modelo.

Un diagrama de componentes muestra las organizaciones y dependencias lógicas entre componentes software, sean éstos componentes de código fuente, binarios o

ejecutables. Desde el punto de vista del diagrama de componentes se tienen en consideración los requisitos relacionados con la facilidad de desarrollo, la gestión del software, la reutilización, y las restricciones impuestas por los lenguajes de programación y las herramientas utilizadas en el desarrollo. Los elementos de modelado dentro de un diagrama de componentes serán componentes y paquetes. En cuanto a los componentes, sólo aparecen tipos de componentes, ya que las instancias específicas de cada tipo se encuentran en el diagrama de despliegue.

Dado que los diagramas de componentes muestran los componentes software que constituyen una parte reutilizable, sus interfaces, y su interrelaciones, en muchos aspectos se puede considerar que un diagrama de componentes es un diagrama de clases a gran escala. Cada componente en el diagrama debe ser documentado con un diagrama de componentes más detallado, un diagrama de clases, o un diagrama de casos de uso.

Un paquete en un diagrama de componentes representa un división física del sistema. Los paquetes se organizan en una jerarquía de capas donde cada capa tiene una interfaz bien definida. Un ejemplo típico de una jerarquía en capas de este tipo es: Interfaz de usuario; Paquetes específicos de la aplicación; Paquetes reutilizables; Mecanismos claves; y Paquetes hardware y del sistema operativo.

Un diagrama de componentes se representa como un grafo de componentes software unidos por medio de relaciones de dependencia (generalmente de compilación). Puede mostrar también que un componente software contiene una interfaz, es decir, la soporta. Normalmente los diagramas de componentes se utilizan para modelar código fuente, versiones ejecutables, bases de datos físicas, entre otros:

- Código fuente: En el modelado de código fuentes se suelen utilizar para representar las dependencias entre los ficheros de código fuente, o para modelar las diferentes versiones de estos fichero. Para ello se deben identificar el conjunto de archivos de código fuente de interés y estereotiparlos como archivos file, agruparlos en paquetes, utilizar valores etiquetados para la información de versiones y modelar las dependencias de compilación.

- Código ejecutable: Se utiliza para modelar la distribución de una nueva versión a los usuarios. Para tal propósito se identifican el conjunto de componentes ejecutables que intervienen, se utilizan estereotipos para los diferentes tipos de componentes (ejecutables, bibliotecas, tablas, archivos, documentos, etc.), se consideran las relaciones entre dichos componentes que la mayoría de las veces incluirán interfaces que son exportadas (realizadas) por ciertos componentes e importadas (utilizadas) por otros.

- Bases de datos física: UML permite el modelado de bases de datos físicas así como de los esquemas lógicos de bases de datos.

Así si se tiene en cuenta las dependencias asociadas al proceso de compilación un componente podría ser:

- Código fuente que depende de otros componentes (no necesariamente código fuente) que deben estar disponibles durante la compilación del componente.

- Código objeto binario, como por ejemplo una librería, que puede depender de algún código objeto con el que se liga.

- Código ejecutable que puede depender de otros programas ejecutables con los que interacciones en tiempo de ejecución.

Se pueden utilizar estereotipos como <<link>> o <<compile>> para distinguir la distinta naturaleza de las dependencias. Igualmente se pueden definir estereotipos para las distintas clases de componentes. UML proporciona algunos estereotipos predefinidos como: <<file>>, <<library>>, <<executable>>, <<table>> y document>>.

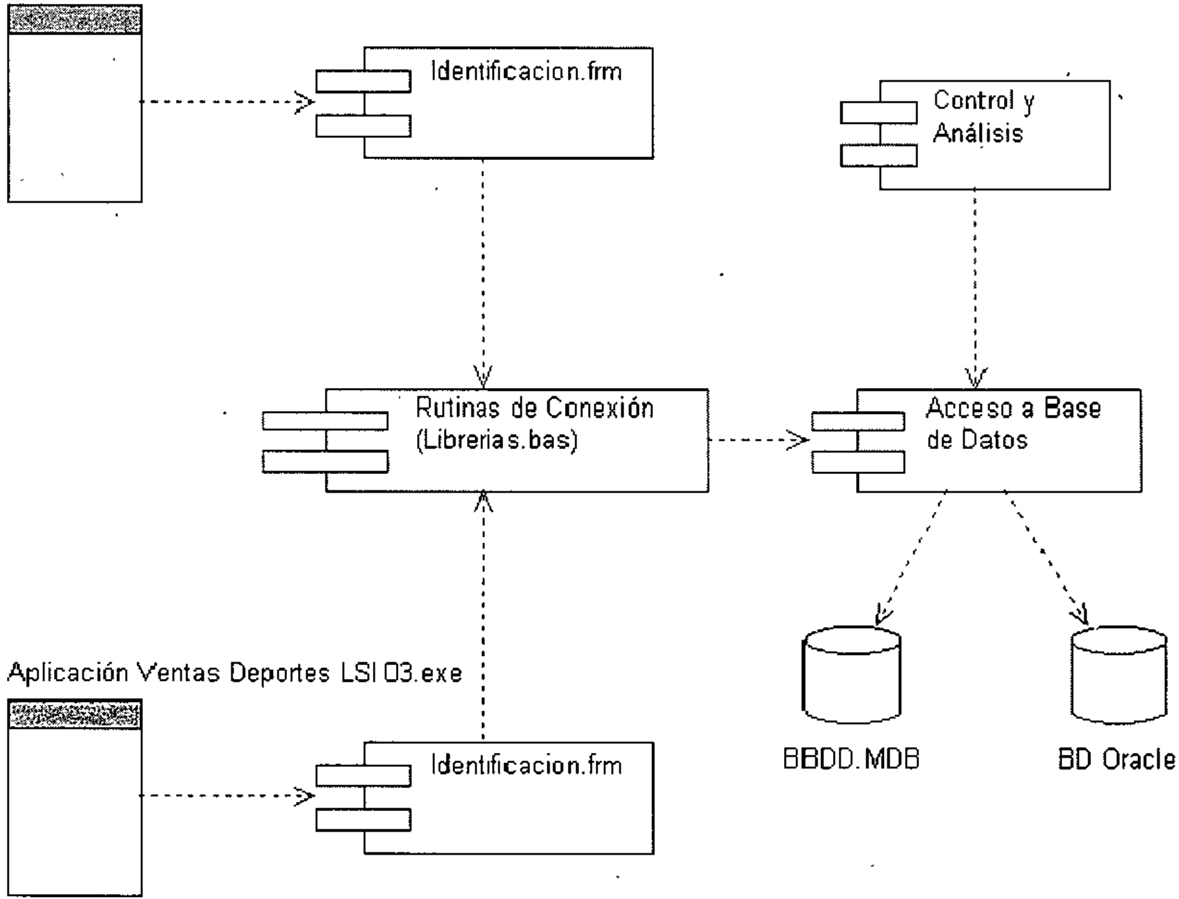


Figura 24. Diagrama de componentes.

9.2. Despliegue

Un diagrama de despliegue **(figura 25)** muestra las relaciones físicas entre los componentes hardware y software en el sistema final, es decir, la configuración de los elementos de procesamiento en tiempo de ejecución y los componentes software

(procesos y objetos que se ejecutan en ellos). Estarán formados por instancias de los componentes software que representan manifestaciones del código en tiempo de ejecución (los componentes que sólo sean utilizados en tiempo de compilación deben mostrarse en el diagrama de componentes).

Un diagrama de despliegue es un grafo de nodos unidos por conexiones de comunicación. Un nodo puede contener instancias de componentes software, objetos, procesos (caso particular de un objeto). En general un nodo será una unidad de computación de algún tipo, desde un sensor a un mainframe. Las instancias de componentes software pueden estar unidas por relaciones de dependencia, posiblemente a interfaces (ya que un componente puede tener más de una interfaz).

Los diagramas de despliegue muestran la configuración en funcionamiento del sistema, incluyendo su hardware y su software. Para cada componente de un diagrama de despliegue se deben documentar las características técnicas requeridas, el tráfico de red esperado, el tiempo de respuesta requerido,.etc.

La mayoría de las veces el modelado de la vista de despliegue estática 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. Aunque UML no es un lenguaje de especificación hardware de propósito general, se ha diseñado para modelar muchos de los aspectos hardware de un sistema a un nivel suficiente para que un ingeniero software pueda especificarla plataforma sobre la que se ejecuta el software del sistema y para que un ingeniero de sistemas pueda manejar la frontera entre el hardware y el software cuando se trata de la relación entre hardware y software se utilizan los diagramas de despliegue para razonar sobre la topología de procesadores y dispositivos sobre los que se ejecuta el software.

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

- Sistemas empotrados: Un sistema empotrado es un colección de hardware con una gran cantidad de software que interactúa con el mundo físico. Los sistemas empotrados involucran software que controla dispositivo (motores, actores) que a su ves están controlados por estímulos externos como sensores.

- Sistemas cliente-servidor: Los sistemas cliente-servidor son un extremos del espectro de los sistemas distribuidos y requieren tomar decisiones sobre la conectividad de red de los clientes a los servidores y sobre la distribución física de los componentes software del sistemas a través de nodos.

- Sistemas completamente distribuidos: En el otro extremo encontramos aquellos sistemas que son ampliamente o totalmente distribuidos y que normalmente incluyen varios niveles de servidores Tales sistemas contienen a menudo varias versiones de componentes software, alguno de los cuales pueden incluso migrar de un nodo a otro. El diseño de tales sistemas requiere

tomar decisiones que permitan un cambio continuo de la topología del sistema.

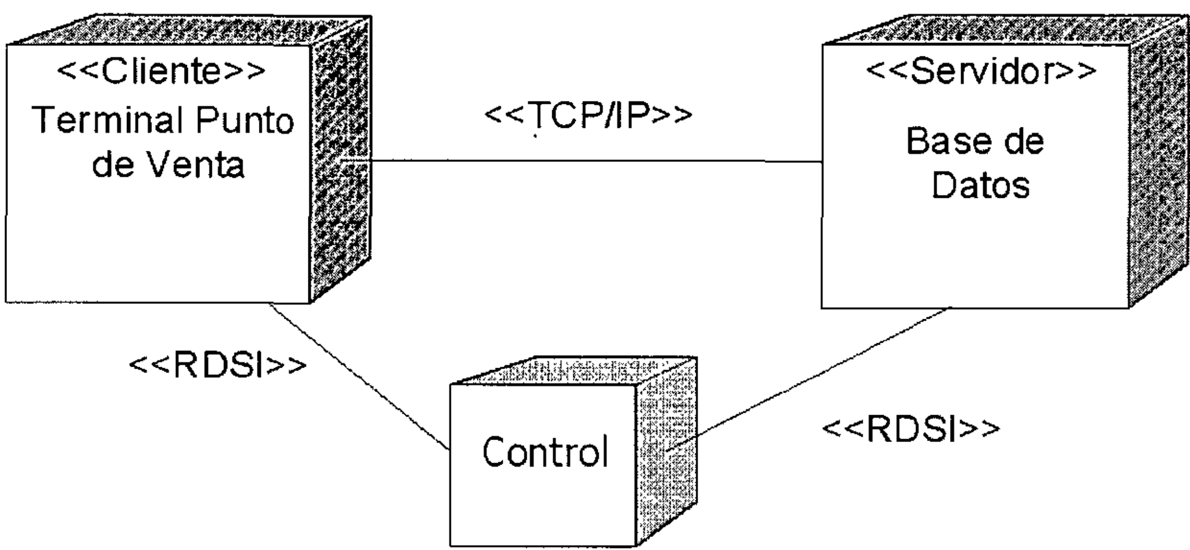


Figura 25. Diagrama de despliegue.

10.Bibliografía

http://www.fi-b.unam.mx/pp/profesores/carlos/aydoo/intro.html

http://www.monografias.com/trabajos5/inso/inso.shtml

http://www.dcc.uchile.cl/~psalinas/uml/casosuso.html

http://www.dcc.uchile.cl/~psalinas/uml/modelo.html

http://www.dcc.uchile.cl/~cc61j/dinamica/sld005.htm

http://www.omg.org/uml/

http://www.celigent.com/uml/

http://www.cetus-links.org/oo_uml.html

http://www.enteract.com/~bradapp/docs/patterns-intro.html

http://www.monografias.com/trabajos22/desarrollo-software/desarrollo-software.shtml

G. Booch, J. Rumbaugh, I. Jacobson. "The Unified Modeling Language Reference Manual". Addison Wesley.

G. Booch, J. Rumbaugh, I. Jacobson. "The Unified Modeling Language User Guide". Addison Wesley

Craig Larman. UML y Patrones, Introducción al análisis y diseño orientado a objetos. Prentice Hall. Primera versión en Español, 1999.

Pierre-Alain Muller, "Instant UML"

Martin Fowler, "UML Destilled" ("UML Gota a Gota")

Terry Quatrani, "Visual Modeling ...", un caso de estudio

Manuales Sun MicroSystems