UNIVERSIDAD DE ORIENTE

NÚCLEO DE ANZOÁTEGUI

ESCUELA DE INGENIERÍA Y CIENCIAS APLICADAS

DEPARTAMENTO DE COMPUTACIÓN Y SISTEMAS



**METODOLOGÍA ORIENTADA A OBJETOS**

**Y LENGUAJE UNIFICADO DE MODELADO**

Profesor: Integrantes:

Victor Mujica Díaz Nicolás C.I. 20347421

Caraballo José C.I. 21013733

Núñez Luis C.I. 20874644

Rojas Elhymar C.I. 21174128

Barcelona, Mayo de 2015

**INTRODUCCIÓN**

Las metodologías de desarrollo de software en la ingeniería de software son un framework o marco de trabajo necesario para estructurar, planificar y controlar el proceso de desarrollo en los sistemas de información. A lo largo del tiempo, una gran cantidad de estos métodos han sido desarrollados diferenciándose por sus fortalezas y debilidades. El framework para metodología de desarrollo de software consiste en una filosofía de desarrollo de programas de computación con el enfoque del proceso de desarrollo de software, además de herramientas, modelos y métodos para asistir dicho proceso.

Metodologías de Desarrollo de Software tiene como objetivo presentar un conjunto de técnicas tradicionales y modernas de modelado de sistemas que permitan desarrollar software de calidad, incluyendo heurísticas de construcción y criterios de comparación de modelos de sistemas. Para tal fin se describen, fundamentalmente, herramientas de Análisis y Diseño Orientado a Objetos (UML), sus diagramas, especificación, y criterios de aplicación de las mismas.

**METODOLOGÍA ORIENTADA A OBJETOS (OOM)**

* Introducción a OOM

El modelo y diseño orientado a objetos u OMT (técnica de modelado de objetos) se extiende desde el análisis hasta la implementación pasando por el diseño. Actualmente es una de las metodologías más implantadas.

Las técnicas orientadas a objetos permiten que el software se construya a partir de objetos de compartimiento específico.

Los propios objetos se pueden constituir a partir de otros, que a su vez pueden estar formados por otros objetos. Esto nos recuerda a una maquina compleja construida por partes, subpartes y sub-subpartes, etc.

La metodología de desarrollo de software orientada a objetos es cada día más usada, pues permite desarrollar software fácilmente extensible y reusable. Esto último es sólo posible si los desarrolladores conocen muy bien los fundamentos que estén basados en esta metodología. Por eso, este curso revisa los conceptos más importantes que se encuentran.

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. 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.

La metodología OMT fue creada por James Rumbaugh y Michael Blaha en 1991, mientras James dirigía un equipo de investigación de los laboratorios General Electric. OMT es una de las metodologías de análisis y diseño orientados a objetos, más maduros y eficientes que existen en la actualidad. La gran virtud que aporta esta metodología es su carácter de abierta (no propietaria), que le permite ser de dominio público y, en consecuencia, sobrevivir con enorme vitalidad. Esto facilita su evolución para acoplarse a todas las necesidades actuales y futuras de la ingeniería de software.

Las fases que conforman a la metodología OMT son:

* **Análisis**. El analista construye un modelo del dominio del problema, mostrando sus propiedades más importantes. El modelo de análisis es una abstracción resumida y precisa de lo que debe de hacer el sistema deseado y no de la forma en que se hará. Los elementos del modelo deben ser conceptos del dominio de aplicación y no conceptos informáticos tales como estructuras de datos. Un buen modelo debe poder ser entendido y criticado por expertos en el dominio del problema que no tengan conocimientos informáticos.
* **Diseño del sistema**. El diseñador del sistema toma decisiones de alto nivel sobre la arquitectura del mismo. Durante esta fase el sistema se organiza en subsistemas basándose tanto en la estructura del análisis como en la arquitectura propuesta. Se selecciona una estrategia para afrontar el problema.
* **Diseño de objetos**. El diseñador de objetos construye un modelo de diseño basándose en el modelo de análisis, pero incorporando detalles de implementación. El diseño de objetos se centra en las estructuras de datos y algoritmos que son necesarios para implementar cada clase. OMT describe la forma en que el diseño puede ser implementado en distintos lenguajes (orientados y no orientados a objetos, bases de datos, etc.).
* **Implementación**. Las clases de objetos y relaciones desarrolladas durante el análisis de objetos se traducen finalmente a una implementación concreta. Durante la fase de implementación es importante tener en cuenta los principios de la ingeniería del software de forma que la correspondencia con el diseño sea directa y el sistema implementado sea flexible y extensible. No tiene sentido que utilicemos AOO y DOO de forma que potenciemos la reutilización de código y la correspondencia entre el dominio del problema y el sistema informático, si luego perdemos todas estas ventajas con una implementación de mala calidad.

La metodología OMT emplea tres clases de modelos para describir el sistema:

* **Modelo de objetos**. Describe la estructura estática de los objetos del sistema. El modelo de objetos proporciona el entorno esencial en el cual se pueden situar el modelo dinámico y el modelo funcional. El objetivo es capturar aquellos conceptos del mundo real que sean importantes para la aplicación. Se representa mediante diagramas de objetos.
* **Modelo dinámico**. Describe los aspectos de un sistema que tratan de la temporización y secuencia de operaciones (sucesos que marcan los cambios, secuencias de sucesos, estados que definen el contexto para los sucesos) y la organización de sucesos y estados. Captura el control, aquel aspecto de un sistema que describe las secuencias de operaciones que se producen sin tener en cuenta lo que hagan las operaciones, aquello a lo que afecten o la forma en que están implementadas. Se representa gráficamente mediante diagramas de estado.
* **Modelo funcional**. Describe las transformaciones de valores de datos que ocurren dentro del sistema. Captura lo que hace el sistema, independientemente de cuándo se haga o de la forma en que se haga. Se representa mediante diagramas de flujo de datos

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.

**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.

* Beneficios de OOM
* Permite ensamblar sistemas grandes a partir de módulos reutilizables.
* Reutiliza clases ya probadas y depuradas
* Permite extender las capacidades de módulos y objetos a través de la herencia, aportando flexibilidad.
* Las interfaces entre módulos y sistemas externos es más fácil a través del paso de mensajes.
* La partición en objetos facilita la escalabilidad de los sistemas.
* Se puede partir el desarrollo de un sistema en forma más fácil.
* Mejora la seguridad de los sistemas al ocultar la información de los objetos.
* El análisis orientado a objetos abarca los modelos centrados en datos.
* Métodos formales más claros en MOO
* OO es una herramienta para manejar la complejidad
* Evolución y mantenimiento de los sistemas más fácil.
* OO tiene mayor potencial para captar mayor significado en sus aplicaciones
* GUI, WFS, DSD.

**LENGUAJE UNIFICADO DE MODELADO (UML)**

* Definición

Lenguaje Unificado de Modelado (UML, por sus siglas en inglés, Unified Modeling Language) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; está respaldado por el OMG (Object Management Group)

El término “lenguaje” ha generado bastante confusión respecto a lo que es UML. En realidad el término lenguaje quizás no es el más apropiado, ya que no es un lenguaje propiamente dicho, sino una serie de normas y estándares gráficos respecto a cómo se deben representar los esquemas relativos al software. Mucha gente piensa por confusión que UML es un lenguaje de programación y esta idea es errónea: UML no es un lenguaje de programación. Como decimos, UML son una serie de normas y estándares que dicen cómo se debe representar algo.

El lenguaje unificado de diagrama o notación (UML) sirve para especificar, visualizar y documentar esquemas de sistemas de software orientado a objetos. UML no es un método de desarrollo, lo que significa que no sirve para determinar qué hacer en primer lugar o cómo diseñar el sistema, sino que simplemente le ayuda a visualizar el diseño y a hacerlo más accesible para otros. UML está controlado por el grupo de administración de objetos (OMG) y es el estándar de descripción de esquemas de software.

* Historia

El desarrollo de UML comenzó a finales de 1994 cuando Grady Booch y Jim Rumbaugh de Rational Software Corporation empezaron a unificar sus métodos. A finales de 1995, Ivar Jacob son y su compañía Objectory se incorporaron a Rational en su unificación, aportando el método OOSE.

La notación UML se deriva y unifica las tres metodologías de análisis y diseños más extendidas.

Metodología de Grady Booch para la descripción de conjuntos de objetos y sus relaciones.

Técnica de modelado orientada a objetos de James Rumbaugh (OMT: Object - Modelling Technique).

Aproximación de Ivar Jacobson (OOSE: Object- Oriented Software Engineering) mediante la metodología de casos de uso (use case).

Además, este lenguaje se abrió a la colaboración de otras empresas para que aportaran sus ideas. Todas estas colaboraciones condujeron a la definición de la primera versión de UML.

De las tres metodologías de partida, las de Bco. y Rumbaugh pueden ser descritas como centradas en objetos, ya que sus aproximaciones se enfocan hacia el modelado de los objetos que componen el sistema, su relación y colaboración.

Por otro lado, la metodología de Jacobson es más centrada a usuario, ya que todo en su método se deriva de los escenarios de uso. UML se ha ido fomentando y aceptando como estándar desde el OMG, que es también el origen de CORBA, el estándar líder en la industria para la programación de objetos distribuidos.

En 1997 UML 1.1 fue aprobada por la OMG convirtiéndose en la notación estándar de facto para el análisis y el diseño orientado a objetos.

UML es el primer método en publicar un meta-modelo en su propia notación, incluyendo la notación para la mayoría de la información de requisitos, análisis y diseño. Se trata pues de un meta-modelo auto-referencial (cualquier lenguaje de modelado de propósito general debería ser capaz de modelarse a sí mismo).

**UML 1.x**

Como notación de modelado, la influencia de la OMT domina UML (por ejemplo el uso de rectángulos para clases y objetos). Aunque se quitó la notación de "nubes" de Booch, si se adoptó la capacidad de Booch para especificar detalles de diseño en los niveles inferiores. La notación de Casos de Uso del Objectory y la notación de componentes de Booch fueron integradas al resto de la notación, pero la integración semántica era relativamente débil en UML 1.1, y no se arregló realmente hasta la revisión mayor de UML 2.0.

Conceptos de muchos otros métodos OO fueron integrados superficialmente en UML con el propósito de hacerlo compatible con todos los métodos OO. Además el grupo tomó en cuenta muchos otros métodos de la época, con el objetivo de asegurar amplia cobertura en el dominio de los sistemas en tiempo real. Como resultado, UML es útil en una gran variedad de problemas de ingeniería, desde procesos sencillos y aplicaciones de un sólo usuario a sistemas concurrentes y distribuidos.

El Lenguaje de Modelado Unificado es un estándar internacional:

ISO / IEC 19501:2005 Tecnología de la información - Procesamiento distribuido abierto - Lenguaje de Modelado Unificado (UML) Versión 1.4.2

**UML 2.x**

UML ha madurado considerablemente desde UML 1.1. Varias revisiones menores (UML 1.3, 1.4 y 1.5) han corregido defectos y errores de la primera versión de UML. A estas le ha seguido la revisión mayor UML 2.0 que fue adoptada por el OMG en 2005.

Aunque UML 2.1 nunca fue lanzado como una especificación formal, las versiones 2.1.1 y 2.1.2, aparecieron en 2007, seguidas por UML 2.2 en febrero de 2009. UML 2.3 fue lanzado oficialmente en mayo de 2010. UML 2.4.1 fue lanzado oficialmente en agosto de 2011. UML 2.5 fue lanzado en octubre de 2012 como una versión "En proceso" y todavía tiene que ser formalmente liberada.

* Ciclo de Vida

El ciclo de vida de un proyecto software son todas las etapas por las que debe pasar dicho proyecto, desde la concepción de la idea que hace surgir la necesidad de diseñar un sistema software, pasando por varias etapas intermedias como el análisis, desarrollo, implantación y mantenimiento del mismo y hasta que finalmente muere por ser sustituido por otro sistema. Éste proceso puede ser sencillo o laborioso.

Aunque UML es bastante independiente del proceso, para obtener el máximo rendimiento de este lenguaje se considera que el modelo de proceso de software sea:

* Basados en casos de uso, es decir, que los casos de uso sean un artefacto básico para establecer el comportamiento deseado del sistema, para validar la arquitectura, para las pruebas y para la comunicación entre las personas involucradas en el proyecto.
* Centrado en la arquitectura, de modo que sea el artefacto básico para conceptuar, construir, gestionar y hacer evolucionar el sistema.
* Iterativo, que involucre la gestión del flujo de ejecutables del sistema.
* Incremental, donde cada nueva versión corrige defectos de la anterior e incorpora nueva funcionalidad. Un proceso iterativo e incremental se denomina dirigido por el riesgo, lo que significa que cada nueva versión se ataca y reducen los riesgos más significativos para el éxito del proyecto.

Durante el ciclo de vida de un proyecto software existen cuatro fases por las cuales debe pasar el proceso de desarrollo iterativo, entre las cuales se encuentran las siguientes:

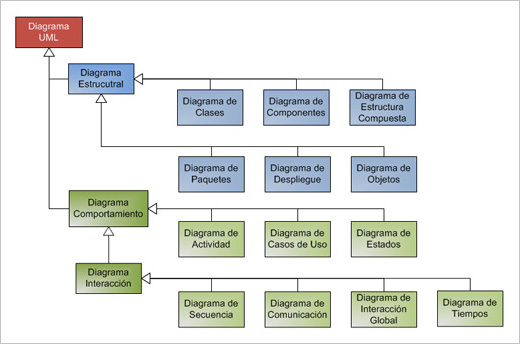
* Iniciación o concepción. En ésta fase se desarrollan las ideas para otorgar un sistema final, es decir, pensar en lo que realizará el sistema, como estará compuesto arquitectónicamente, cual es el plan a seguir para lograrlo y además el costo que tendrá el mismo sistema, tomando en cuenta los riesgos que se puedan presentar durante la elaboración de la aplicación. Un buen sistema debe de pensar en todo esto antes de realizarlo.
* Elaboración. Se deben determinar los requisitos del sistema y las pruebas sobre el mismo. En esta fase se establecen en detalle los casos de uso que se van a implementar para el desarrollo del programa y a la vez se diseña la estructura del sistema, teniendo al final de esta fase las actividades planeadas y los recursos necesarios para que se ejecute el proyecto.
* Construcción, es cuando se pasa de la base arquitectónica ejecutable hasta su disponibilidad para los usuarios. Se reexaminan los requisitos y las pruebas que ha de soportar. Como su nombre lo dice, se comienza a construir (programar) la aplicación tomando en cuenta la estructura que se realizó del proyecto durante la etapa anterior. En esta fase se debe de tener un avance del sistema a crear, para que los usuarios otorguen opiniones de la aplicación y si existe algún error o deficiencia se puedan realizar los cambios necesarios para que este se desarrolle perfectamente.
* Transición. El proceso del software termina en la etapa de transición, ya que es continuamente reexaminado y mejorado erradicando errores y añadiendo nuevas funcionalidades no contempladas. En esta parte de desarrollo se otorga el sistema a los usuarios para que se realicen pruebas del sistema, se pudiese llamar como si fuera una versión beta del programa, para si existiese fallos se puedan corregir con anterioridad y cumplir con el objetivo señalado en tiempo y forma.

La iteración afecta a estas cuatro fases. Una iteración es la repetición de una serie de instrucciones repetitivas que comprenden el proceso para lograr un sistema, como ejemplo de ello en programación se puede mencionar la recursividad, ya que esta permite repetir las instrucciones hasta que se cumpla una acción en específico. Este elemento está presente en UML como un conjunto bien definido de actividades, con un plan y unos criterios de evaluación, que acaban en una versión del producto.

* Modelo de la Arquitectura

El lenguaje UML se expresa con símbolos y/o agrupaciones llamados diagramas. Sirven fundamentalmente para crear diferentes tipos de ellos permitiendo ver desde diferentes perspectivas un sistema software.

En el estándar UML 2.0 existen un grupo de diagramas determinado que son los más comunes y habituales. Se pueden establecer una clasificación de estos dividiéndolos en Estructurales y de Comportamiento, y dentro de los de comportamiento se tiene a los de Interacción.



Los diagramas son de gran utilidad para trabajar en los requisitos, en el análisis del sistema, en la construcción del mismo y en su posterior despliegue.

Si se agrupan los diferentes diagramas se tiene el resultado de lo que es llamado vista. Al desarrollar un sistema con gran cantidad de software se requiere que este sea visto desde diferentes perspectivas y diferentes usuarios. Por esto las vistas permiten describir mejor los sistemas, donde cada una de ellas es una proyección de la organización y la estructura centrada en un aspecto particular del sistema. Entonces, se puede decir que las vistas determinan la arquitectura del sistema.

La arquitectura es un conjunto de decisiones significativas sobre:

* La organización del sistema.
* Selección de elementos estructurales y sus interfaces a través de los cuales se constituye el sistema.
* El comportamiento, como se especifican las colaboraciones entre los componentes del sistema.
* Composición de los elementos estructurales y de comportamiento en subsistemas progresivamente más grandes.
* El estilo arquitectónico que guía la organización son: elementos estáticos, elementos dinámicos y sus interfaces, sus colaboraciones y su composición.

La arquitectura del software no tiene que ver solamente con la estructura y el comportamiento, sino también con el uso, la funcionalidad, el rendimiento, la capacidad de adaptación, la reutilización, la capacidad de ser comprendido, las restricciones económicas y tecnológicas y los compromisos entre alternativas, así como los aspectos estéticos.

Las diferentes vistas que permite la arquitectura para describir mejor el sistema son:

* La vista de casos de uso, comprende la descripción del comportamiento del sistema tal y como es percibido por los usuarios finales, analistas y encargados de las pruebas y se utilizan los diagramas de casos de uso para capturar los aspectos estáticos mientras que los dinámicos son representados por diagramas de interacción, estados y actividades.
* La vista de diseño comprende las clases, interfaces y colaboraciones que forman el vocabulario del problema y de la solución. Esta vista soporta principalmente los requisitos funcionales del sistema, o sea, los servicios que el sistema debe proporcionar. Los aspectos estáticos se representan mediante diagramas de clases y objetos y los aspectos dinámicos con diagramas de interacción, estados y actividades.
* La vista de interacción comprende los hilos y procesos que forman mecanismos de sincronización y concurrencia del sistema cubriendo el funcionamiento, capacidad de crecimiento y el rendimiento del sistema. Con UML, los aspectos estáticos y dinámicos se representan igual que en la vista de diseño, pero con el énfasis que aportan las clases activas, las cuales representan los procesos y los hilos.
* La vista de implementación comprende los componentes y los archivos que un sistema utiliza para ensamblar y hacer disponible el sistema físico. Se ocupa principalmente de la gestión de configuraciones de las distintas versiones del sistema. Los aspectos estáticos se capturan con los diagramas de componentes y los aspectos dinámicos con los diagramas de interacción, estados y actividades.
* La vista de despliegue de un sistema contiene los nodos que forman la topología hardware sobre la que se ejecuta el sistema. Se preocupa principalmente de la distribución, entrega e instalación de las partes que constituyen el sistema. Los aspectos estáticos de esta vista se representan mediante los diagramas de despliegue y los aspectos dinámicos con diagramas de interacción, estados y actividades.

Vista de Diseño

Funcionamiento y vocabulario del sistema.

* Diagramas de clases
* Diagramas de comunicación
* *Diagramas de secuencia*

Vista de Implementación

Ensamblaje, sistema físico y gestión de configuración.

* Diagramas de componentes
* *Diagramas de paquetes*

Vista de Interacción

Capacidad de crecimiento y rendimiento del sistema.

* *Diagramas de actividades*

Vista de Despliegue

Topología hardware, distribución, entrega e instalación del sistema.

* Diagramas de despliegue

**CONCLUSIÓN**

Hoy en día la tecnología orientada a objetos ya no se aplica solamente a los lenguajes de programación, además se viene aplicando en el análisis y diseño con mucho éxito, al igual que en las bases de datos. Es que para hacer una buena programación orientada a objetos hay que desarrollar todo el sistema aplicando esta tecnología, de ahí la importancia del análisis y el diseño orientado a objetos.

UML es un lenguaje de modelado y no de programación. El vocabulario y las reglas de un lenguaje como UML indican cómo crear y leer modelos bien formados, pero no dice que modelos se deben crear ni cuando se deberían crear. Esta tarea corresponde al proceso de desarrollo del software. Detrás de cada símbolo en la notación de UML hay una semántica bien definida, de esta manera un desarrollador puede escribir un modelo en UML, y otro desarrollador o incluso otra herramienta, puede interpretar ese modelo sin ambigüedad. UML está pensado principalmente para sistemas con gran cantidad de software.