



MONOGRAFÍA

INGENIERÍA DEL SOFTWARE ORIENTADA A OBJETOS

AISI - ISOO

Contenido

1.	Introducción	1
2.	Historia	1
3.	Modelos de Proceso de Desarrollo de Software	2
3.1	Modelo de Proceso Lineal Secuencial.....	3
3.2	Modelos de Proceso por Incrementos	3
3.2.1	Modelo Incremental	3
3.2.2	Modelo rápido de aplicaciones (RAD).....	5
3.3	Modelos de Proceso Evolutivos.....	6
3.3.1	Modelo de Construcción de Prototipos	6
3.3.2	Modelo en Espiral	7
3.4	Modelo de Proceso Basado en Componentes	8
3.5	Modelos de Proceso de Desarrollo Orientados a Objetos	10
3.5.1	Modelo de Branson y Herness	10
3.5.2	Programación Extrema	12
3.5.3	Proceso Unificado de Desarrollo de Software	13
4.	Análisis Orientado a Objetos	15
4.1	Análisis del dominio.....	16
4.1.1	Proceso del análisis del dominio	17
4.2	Proceso del Análisis Orientado a Objetos	18
4.2.1	Casos de uso.....	18
4.2.2	Modelado de clases, responsabilidades y colaboraciones	18
4.2.3	Definición de estructuras y jerarquías	19
4.2.4	Definición de subsistemas	19
4.3	Modelo Objeto-Relación	19
4.4	Modelo de herencia	20
4.5	Modelo Objeto-Comportamiento	21
5.	Diseño Orientado a Objetos	22
5.1	Métodos de Diseño Orientados a Objeto.....	23
5.2	Diseño del Sistema	24
5.3	Diseño de Clases	26
5.4	Patrones de Diseño	27
5.5	Enfoque convencional vs. OO	28
6.	Programación Orientada a Objetos	28
6.1	Características importantes en la programación orientada a objetos	30
7.	Pruebas Orientadas a Objeto.....	31
7.1	Exactitud de los modelos de AOO y DOO.....	32
7.2	Consistencia de los modelos de AOO y DOO	32

7.3 Estrategias de Pruebas Orientadas a Objetos.....	33
7.3.1 Pruebas de Unidad en el contexto OO.....	33
7.3.2 Pruebas de Integración en el contexto OO	34
7.3.3 Pruebas de validación en el contexto OO.....	34
7.4 Diseño de casos de prueba.....	34
8. Lenguaje de Modelado Unificado (UML).....	35
8.1 Objetivo de UML	35
8.2 Vista Estática	36
8.2.1 Diagrama de clases.....	36
8.3 Vista de Casos de Uso	36
8.3.1 Actor.....	36
8.3.2 Casos de uso.....	36
8.4 Vista de Interacción.....	37
8.4.1 Diagrama de Secuencia	37
8.4.2 Diagrama de Colaboración	38
8.5 Vista de la Máquina de estados.....	38
8.5.1 Diagrama de Estados	38
8.6 Vista de Actividades.....	40
8.6.1 Diagrama de Actividades.....	40
8.7 Vista Física	40
8.8 Vista de Implementación	40
8.8.1 Diagrama de Componentes.....	41
8.9 Vista de Despliegue.....	41
8.9.1 Diagrama de Despliegue.....	41
8.10 Vista de Gestión del Modelo	41
8.11 Construcciones de Extensión	42
Conclusiones.....	42
BIBLIOGRAFÍA	43

1. Introducción

Los objetos que existen en el mundo real, en las entidades creadas por el ser humano, en los negocios y en los productos de uso diario, son las entidades que describen el mundo. Estos objetos pueden ser clasificados, descritos, organizados y creados. Las tecnologías de objetos contienen un número de beneficios que proporcionan ventajas a los niveles de dirección y técnico. Estas tecnologías conducen a la reutilización de componentes del software, las cuales llevan a un desarrollo del software más rápido y a la obtención de programas de mejor calidad. Por otra parte el software orientado a objetos es fácil de mantener debido a su estructura. Esto causa efectos colaterales menores cuando se deben hacer cambios y provoca menos frustración en el ingeniero del software y en el cliente.

En este trabajo se describe de manera global y general los conceptos asociados a la ingeniería del software orientado a objetos, y la tecnología asociada a este paradigma, se brinda una descripción de las etapas que conforman un ciclo de vida orientado a objetos (OO) para obtener un producto, también se desglosa las vistas que conforman el Lenguaje de Modelado Unificado que se ha convertido en el lenguaje de facto para modelar y especificar productos software orientado a objetos.

2. Historia

La ingeniería del software ha evolucionado desde sus primeros días en 1940, las aplicaciones han evolucionado continuamente. La continua meta de mejorar tecnologías y prácticas, busca aumentar la productividad y la calidad de las aplicaciones. El término ingeniería del software fue utilizado por primera vez a finales de los años 50 y comienzos de los años 60, los programadores debatían lo que ingeniería podría significar para el software. En una conferencia en 1968 se dio la primera iniciativa para la ingeniería del software y muchos pensaron que ese era el comienzo de la profesión de la ingeniería del software [Mahoney, 2004][Sommerville, 2005].

La crisis del software fue la causa esencial de la ingeniería del software durante los años 60 al 80, tiempo en el que se identificaron muchos de los problemas del desarrollo de software.

Muchos proyectos de software sobrepasaron su presupuesto y tiempo. Algunos proyectos causaron daños, pérdida de vidas humanas. Muchos pensaban que el término crisis del software se refería a la incapacidad de contratar programadores calificados. Durante décadas, el principal objetivo era resolver la crisis del software por parte de investigadores y compañías que producían herramientas software. Los principales elementos que ayudaron a ese objetivo según cita [Mahoney, 2004] fueron:

- Las herramientas como la programación estructurada, la programación orientada a objetos, las herramientas CASE¹, documentación, estándares.
- La disciplina, debido a que se consideraba que la crisis del software se debía a la falta de disciplina de los programadores.
- Los métodos formales para el proceso fueron determinantes al convertir el desarrollo del software en software de producción.

A la fecha la ingeniería del software continua evolucionando, investigadores y multimillonarias empresas se dedican que proponer metodologías, herramientas que ayuden a desarrollar software cada vez más exigente y complejo debido al surgimiento de Internet, la información tiene que estar presente en todo momento, debe ser confiable para esto se deben aplicar métodos que coadyuve desarrollar software de calidad.

3. Modelos de Proceso de Desarrollo de Software

Los modelos de proceso fueron propuestos originalmente para traer orden al caos en el desarrollo del software. En este entendido un modelo de proceso, define un conjunto claro y completo de actividades, acciones, tareas, hitos y artefactos² que se requiere para transformar los requerimientos del usuario en un producto y por lo tanto construir software de alta calidad [Pressman, 2005] [Jacobson, 2000].

¹ Ingeniería del software asistida por computadora, del inglés Computer Aided Software Engineering

² Artefacto según [Jacobson, 2000] es una pieza de información tangible, puede ser un modelo, un elemento de un modelo, o un documento.

Se selecciona un modelo de proceso para la ingeniería del software, según la naturaleza del proyecto y de la aplicación, los métodos y las herramientas a utilizarse, y los controles y entregas que se requieren. A continuación se muestran los modelos de proceso más conocidos, cada uno de estos modelos sugiere alguna variación de los flujos de proceso, pero que en su ejecución conllevan al conjunto de actividades del marco de trabajo genéricas: comunicación, planificación, modelado, construcción y entrega [Pressman, 2005].

3.1 Modelo de Proceso Lineal Secuencial

El modelo de proceso lineal secuencial³ conocido también como el ciclo de vida clásico sugiere un enfoque sistemático, secuencial, para el desarrollo del software que comienza en un nivel de sistemas y progresa con el análisis, diseño, codificación, pruebas y mantenimiento. Como se muestra en la Figura 1, una etapa de desarrollo debe completarse antes de dar comienzo a la siguiente [Pressman, 2002][Lawrence, 2002].

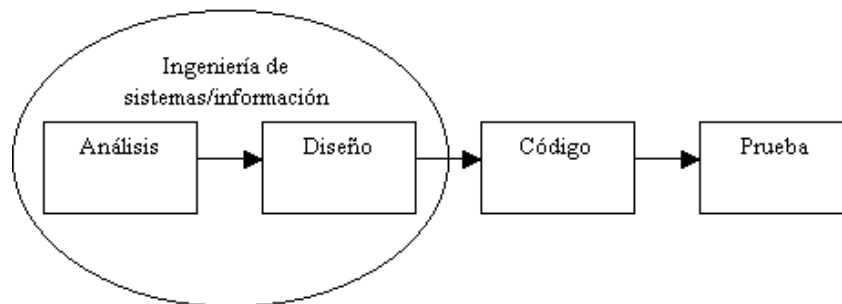


Figura 1: Modelo Lineal Secuencial
Fuente: [Pressman, 2002]

3.2 Modelos de Proceso por Incrementos

3.2.1 Modelo Incremental

El modelo incremental combina elementos del modelo lineal secuencial con la filosofía iterativa. Como se muestra en la Figura 2, el modelo incremental aplica secuencias lineales de

³ Proviene del inglés “Waterfall Model” o Modelo en Cascada.

forma escalonada mientras progresa el tiempo en el calendario. Cada secuencia lineal produce un incremento del software [Pressman, 2002].

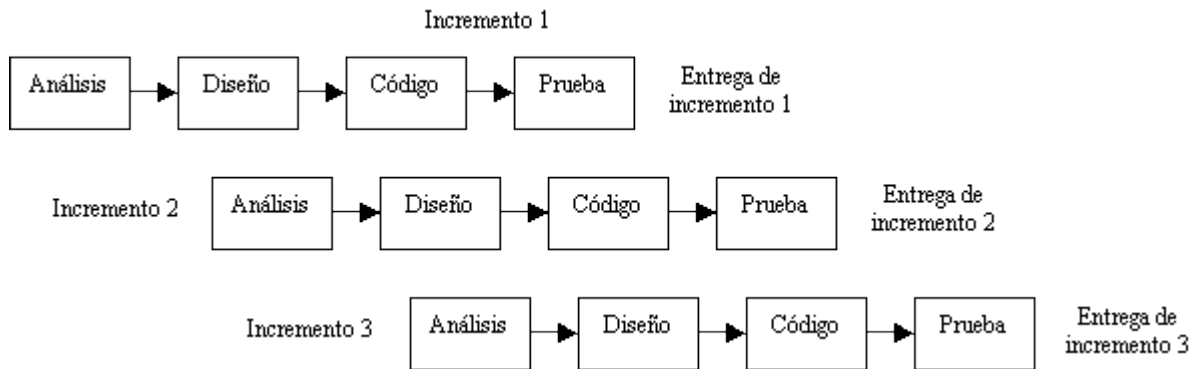


Figura 2: Modelo Incremental
Fuente: [Pressman, 2002]

En el desarrollo incremental, el software, tal como está especificado en los documentos de requerimientos, es particionado de acuerdo con su funcionalidad, el primer incremento a menudo es un producto esencial. Es decir, se afrontan requisitos básicos, pero muchas funciones suplementarias quedan sin extraer. El cliente utiliza el producto central o sufre la revisión detallada. Como un resultado de utilización o de evaluación se desarrolla un plan para el incremento siguiente. El plan afronta la modificación del producto central a fin de cumplir mejor las necesidades del cliente y la entrega de funciones, y características adicionales. Los incrementos⁴ se definen comenzando con un subsistema funcional pequeño y agregando funcionalidad con cada nuevo incremento. Este proceso se repite siguiendo la entrega de cada incremento, hasta que se elabore el producto completo [Pressman, 2002][Lawrence, 2002].

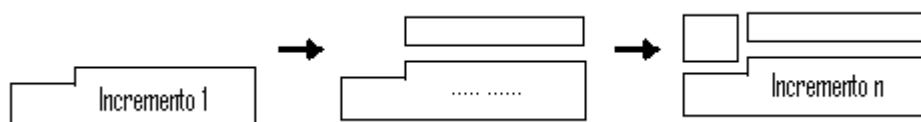


Figura 3: Desarrollo incremental
Fuente: [Lawrence, 2002]

⁴ Versiones

En la Figura 3 se muestra como el desarrollo incremental va construyendo gradualmente su funcionalidad completa con cada nuevo incremento.

3.2.2 Modelo rápido de aplicaciones (RAD)

El modelo rápido de aplicaciones (RAD⁵) es un modelo de proceso de software incremental que enfatiza un ciclo de desarrollo corto. El modelo RAD es una adaptación a alta velocidad del modelo lineal secuencial en el que se logra el desarrollo rápido utilizando una construcción basada en componentes. Si se comprenden bien los requerimientos y se limita el ámbito del proyecto, el proceso RAD permite al equipo de desarrollo crear un sistema completamente funcional dentro de periodos cortos de tiempo⁶ [Pressman, 2005].

Según Kerr(1994) citado en [Pressman, 2005], el enfoque RAD comprende las siguientes fases:

- *Comunicación*, que tiene como objetivo comprender el problema del negocio y las características de información que el software debe satisfacer.
- *Planificación*, es esencial debido a los equipos de desarrollo múltiples que trabajan en paralelo sobre diferentes funciones del sistema.
- *Modelado*, abarca tres fases mayores: el modelado del negocio, modelado de datos y el modelado del proceso. El modelado establece las representaciones del diseño que sirven como base para las actividades de construcción del RAD.
- *Construcción*, hace énfasis en el uso de componentes software ya existentes y en la generación automática de código.
- *Despliegue*, establece una base para iteraciones subsecuentes si es que se requiere.

El modelo RAD es ilustrado en la Figura 4.

⁵ Rapid Application Development

⁶ Según [Martin, 1991] de 60 a 90 días.

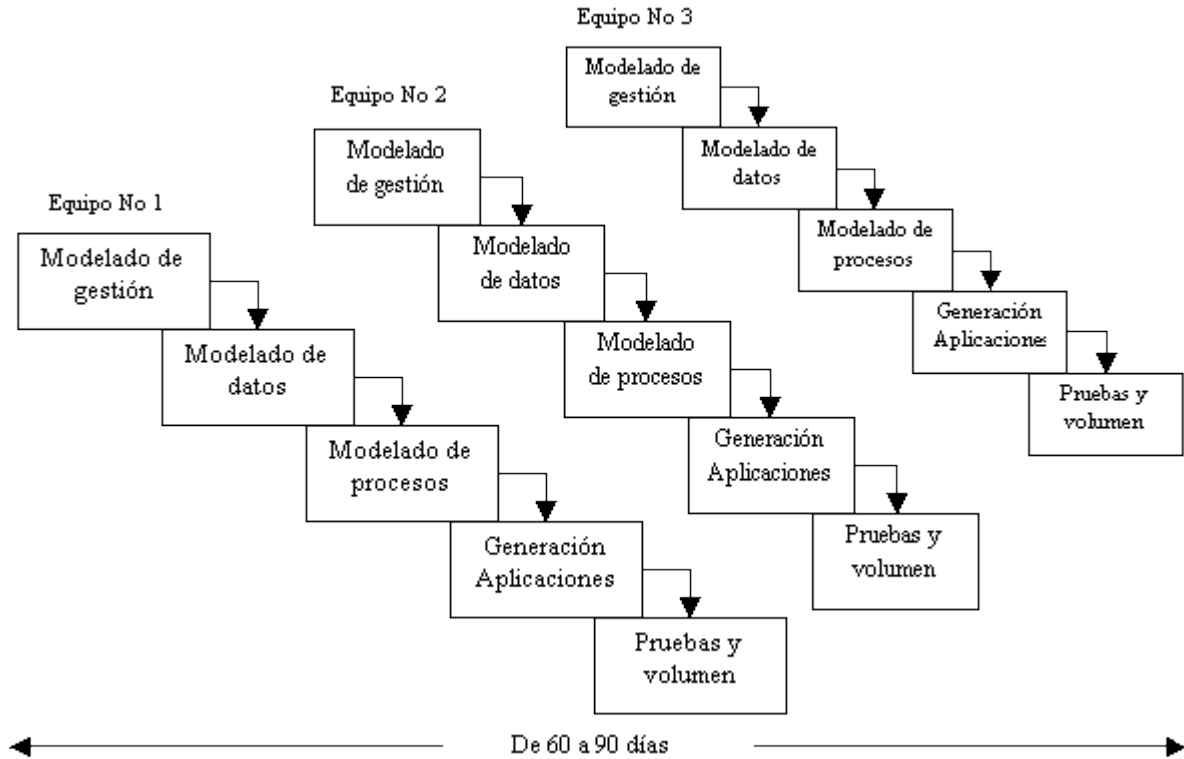


Figura 4: Modelo Rápido de Aplicaciones
Fuente: [Pressman, 2002]

3.3 Modelos de Proceso Evolutivos

3.3.1 Modelo de Construcción de Prototipos

El modelo de construcción de prototipos permite que todo el sistema, o algunas de sus partes, se construyan rápidamente para comprender o aclarar aspectos, donde los requerimientos o el diseño requieren la investigación repetida para asegurar que el equipo de desarrollo, el usuario y el cliente tengan una comprensión unificada tanto de lo que se necesita como lo que se propone como solución. Sin embargo el modelo de construcción de prototipos puede ser usado como un modelo de proceso autónomo, comúnmente usado como una técnica que se implementa dentro el contexto de cualquier modelo de proceso [Lawrence, 2002] [Pressman, 2005].

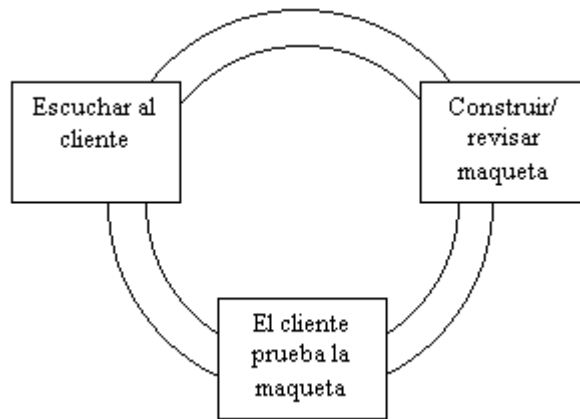


Figura 5: Modelo de Construcción de Prototipos
Fuente: [Pressman, 2002]

El paradigma de construcción de prototipos, presentado en la Figura 5, comienza con la comunicación. El desarrollador y el cliente encuentran y definen los objetivos globales para el software, identifican los requerimientos conocidos y las áreas del esquema en donde es obligatoria más definición. Una iteración del prototipado se planifica rápidamente y ocurre el modelado en la forma de un diseño rápido. El diseño rápido se centra en una representación de aquellos aspectos del software que serán visibles a los clientes y usuarios finales. El diseño rápido conduce a la construcción de un prototipo. El prototipo es entregado y evaluado por los clientes y usuarios. La retroalimentación se utiliza para refinar los requerimientos del software. La iteración ocurre cuando el prototipo se pone a punto para satisfacer las necesidades de los clientes, permitiendo al mismo tiempo que el desarrollador comprenda lo que se necesita hacer [Pressman, 2005].

3.3.2 Modelo en Espiral

El modelo en espiral presentado en la Figura 6, es un modelo de proceso de software evolutivo que conjuga la naturaleza iterativa de construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial. Proporciona el potencial para el desarrollo rápido de versiones incrementales del software [Pressman, 2005].

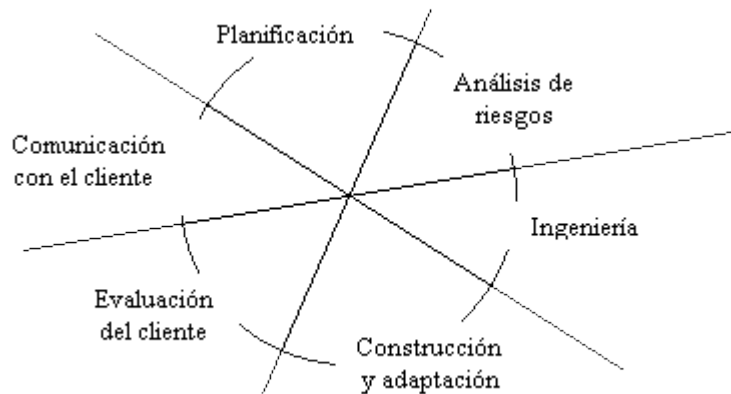


Figura 6: Modelo en Espiral
Fuente: [Pressman, 2002]

Boehm (1998) citado en Lawrence (2002), examinó el proceso de desarrollo del software a la luz de los riesgos involucrados sugiriendo que un modelo en espiral podía combinar las actividades del desarrollo con la gestión del riesgo, para minimizar y controlar el riesgo. En [Boehm, 2001] se describe el modelo de la siguiente manera: “*El modelo de desarrollo en espiral es un modelo de proceso dirigido por los riesgos que se usa para guiar la ingeniería del software concurrente para múltiples clientes de sistemas intensivos. Existen dos principales características distinguibles. Uno es el enfoque cíclico que aumenta incrementalmente un grado de definición e implementación del sistema disminuyendo su grado de riesgo. El otro es un conjunto de hitos para asegurar al cliente soluciones factibles del sistema y mutuamente satisfactorias*”.

3.4 Modelo de Proceso Basado en Componentes

El modelo de proceso basado en componentes esta basado en la reutilización se compone de una gran base de componentes software reutilizables y de marcos de trabajo de integración a éstos. Algunos veces estos componentes son sistemas por si mismos (COST⁷ o sistemas comerciales) que se utilizan para proporcionar una funcionalidad específica, como dar formato al texto o efectuar cálculos numéricos. El modelo incorpora muchas de las características del modelo en espiral. Es evolutivo por naturaleza, y demanda un enfoque iterativo para la

⁷ Del inglés: Commercial off-the-shelf

creación de software [Sommerville, 2005][Pressman, 2005]. En la Figura 5 se muestra el modelo de proceso basado en componentes.

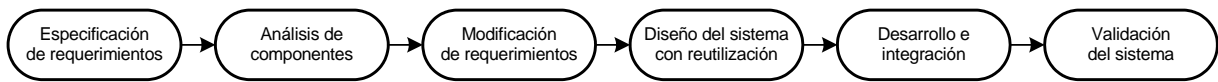


Figura 7: Modelo basado en componentes
Fuente: Modificado de [Sommerville, 2005]

Aunque la etapa de especificación de requerimientos y la validación son comparables con otros procesos, las etapas intermedias en el proceso orientado a la reutilización son diferentes. Estas etapas según [Somerville, 2005] son:

1. *Análisis de componentes.* Dada la especificación de requerimientos, se buscan los componentes para implementar esta especificación. Por lo general, no existe una concordancia exacta y los componentes que se utilizan sólo proporcionan parte de la funcionalidad requerida.
2. *Modificación de requerimientos.* Es esta etapa, los requerimientos se analizan utilizando información acerca de los componentes que se han descubierto. Entonces, estos componentes se modifican para reflejar los componentes disponibles. Si las modificaciones no son posibles, la actividad de análisis de componentes se puede llevar a cabo nuevamente para buscar soluciones alternativas.
3. *Diseño del sistema con reutilización.* En esta fase se diseña o se reutiliza un marco de trabajo para el sistema. Los diseñadores tienen en cuenta los componentes que se reutilizan y organizan el marco de trabajo para que los satisfaga. Si los componentes reutilizables no están disponibles, se puede tener que diseñar nuevo software.
4. *Desarrollo e integración.* Para crear el sistema, el software que no se puede adquirir externamente se desarrolla, y los componentes y los sistemas COTS se integran. En este modelo, la integración de sistemas es parte del proceso de desarrollo, más que una actividad separada.

3.5 Modelos de Proceso de Desarrollo Orientados a Objetos

3.5.1 Modelo de Branson y Herness

Branson y Herness (1992) citado en Kan (2002) propusieron un proceso de desarrollo para proyectos a gran escala que se centra sobre una metodología de ocho pasos apoyada por un mecanismo de seguimiento, una serie inspecciones, un conjunto de tecnologías y reglas de prototipado y prueba. El proceso de ocho pasos son resumidos como sigue:

1. *Modelado esencial del sistema.* El sistema esencial describe aquellos aspectos del sistema requeridos para alcanzar su propósito, sin tomar en cuenta el entorno de hardware y software asignado. Esta compuesto de actividades y datos esenciales. Este paso contiene a su vez cinco tareas:
 - Crear la vista de usuario
 - Actividades del modelo esencial
 - Definir los datos de solución.
 - Refinar el modelo esencial.
 - Construir un análisis detallado.

Este paso se enfoca en los requerimientos del usuario. Los requerimientos son analizados, estudiados, refinados, combinados y organizados dentro de un modelo lógico esencial del sistema.

2. *Derivación de clases candidatas esenciales.* Este paso utiliza una técnica conocida como “carving⁸” para identificar las clases candidatas esenciales y los métodos del modelo esencial de todo el sistema. Un conjunto completo de diagramas de flujo de datos, junto con las especificaciones del proceso de apoyo y las entradas del diccionario de datos, es la base para la selección de clases y métodos. Las clases candidatas y los métodos se encuentran en las entidades externas, datos almacenados, flujos de entrada y especificaciones de proceso.
3. *Restricción el modelo esencial.* El modelo esencial es modificado para trabajar dentro las restricciones del entorno de implementación. Las actividades y datos esenciales son

⁸ No tiene una traducción literal, pero puede ser entendido como arte de trincar, obra de talla, etc.

asignados para varios procesadores y contenedores (repositorios de datos). Las actividades son añadidas al sistema como necesidades, basadas en las limitaciones del entorno de implementación designado. El modelo esencial que crece con las actividades necesarias para el apoyo al entorno designado, es referido como el modelo de personificación.

4. *Derivación de clases adicionales.* Las clases candidatas y los métodos específicos para el entorno de implementación son seleccionados basados en las actividades añadidas durante la restricción del modelo esencial. Estas clases suministran interfaces a las clases esenciales en un nivel consistente.
5. *Sintetización de clases.* Las clases candidatas esenciales y las clases candidatas adicionales son refinadas y organizadas dentro de una jerarquía. Los atributos y operaciones comunes son extraídos para producir superclases y subclases. Las clases finales son seleccionadas para maximizar la reutilización a través de la herencia e importación.
6. *Definición de interfaces.* Las interfaces, declaraciones de tipo de objetos y definiciones de clases son escritas basadas en la documentación de la síntesis de clases.
7. *Completar el diseño:* El diseño del modulo de implementación es completado. El modulo de implementación comprende métodos distintos, cada uno de los cuales proporciona una función cohesiva simple. La lógica, la interacción del sistema y los métodos de invocación a otras clases son usados para lograr el diseño completo de cada método en una clase.
8. *Implementación de una solución:* La implementación de las clases es codificada y la prueba de unidad es llevada a cabo.

Además de la metodología, los requerimientos, el diseño, análisis, implementación, el prototipado y la verificación, Branson y Herness (1993) aseveran que la arquitectura del proceso de desarrollo orientado a objetos debe también dirigir elementos como la reutilización, herramientas CASE, integración, construcción y pruebas, y la gestión del proyecto. El modelo de proceso de Branson y Herness, basado es su experiencia en IBM Rochester, representa un logro para el despliegue de la tecnología orientada a objetos en

grandes organizaciones. Es cierto que surgirán muchas más variaciones antes de que se alcance un modelo de proceso orientado a objetos reconocido comúnmente [Kan, 2002].

3.5.2 Programación Extrema

La programación extrema (XP), un proceso orientado a objetos muy controversial que ha ganado reconocimiento y generado debates vigorosos entre los ingenieros del software, es posiblemente el método ágil más conocido y ampliamente utilizado, el nombre fue acuñado por Kent Beck debido a que el enfoque fue desarrollado utilizando buenas prácticas reconocidas, como el desarrollo iterativo, y con la participación del cliente en niveles “extremos” [Kan, 2002][Somerville, 2005].

En la programación extrema, todos los requerimientos se expresan como escenarios llamados historias de usuario, los cuales se implementan directamente como una serie de tareas. Los programadores trabajan en parejas y desarrollan pruebas para cada tarea antes de escribir código. Todas las pruebas se deben ejecutar satisfactoriamente cuando el código nuevo se integre. Existe un pequeño espacio de tiempo entre las entregas del sistema. Este proceso liviano, iterativo e incremental tiene cuatro valores como piedra angular: comunicación, simplicidad, retroalimentación y coraje [Somerville, 2005] [Kan, 2002]. Con estos principios, XP aboga por las siguientes prácticas resumidas en Somerville (2005):

1. El desarrollo incremental se lleva a cabo a través de entregas del sistema pequeñas y frecuentes y por medio de un enfoque para la descripción de requerimientos basado en las historias del cliente o escenarios que pueden ser la base para el proceso de planificación.
2. La participación del cliente se lleva a cabo a través del compromiso a tiempo completo del cliente en el equipo de desarrollo. Los representantes de los clientes participan en el desarrollo y son los responsables de definir las pruebas de aceptación del sistema.
3. El interés en las personas, en vez de en los procesos, se lleva a cabo a través de la programación en parejas, la propiedad colectiva del código del sistema, y un proceso de desarrollo sostenible que no implique excesivas jornadas de trabajo.

4. El cambio se lleva a cabo a través de las entregas regulares del sistema, un desarrollo previamente probado y la integración continua.
5. El mantenimiento de la simplicidad se lleva a cabo a través de la Refactorización constante para mejorar la calidad del código y la utilización de diseños sencillos que no prevén cambios futuros en el sistema.

Con estas prácticas, el equipo de desarrollo puede “abrigar cambios”. A diferencia de otros modelos de proceso evolutivos, XP disuade la recolección de requerimientos preliminar, el análisis extensivo, y el modelado del diseño. En cambio, limita la planificación intencionalmente para un futuro flexible, promueve una filosofía “You Aren't Gonna Need It”⁹ (YANG I), que enfatiza menos clases y documentación reducida. Lo que aparenta que la filosofía XP y las prácticas pueden ser más aplicables a proyectos pequeños. Para el desarrollo de software grande y complejo, algunos de los principios de XP se convierten en difíciles de implementar y pueden ir incluso en contra de la cordura tradicional, que se construye en los proyectos exitosos. Beck estipula que a la fecha los esfuerzos de XP han trabajado mejor con equipos de diez o menos miembros [Kan, 2002].

3.5.3 Proceso Unificado de Desarrollo de Software

El Proceso de Unificado de Desarrollo de Software, que fue desarrollado por Jacobson, Booch, y Rumbaugh (2000) y es poseído por la Rational Software Corporación, se deriva de metodologías anteriores desarrolladas por estos tres autores, a saber “la metodología Objectory” de Jacobson, la “metodología de Booch” y la “Técnica de Modelado de Objetos” de Rumbaugh [Braude, 2002].

El proceso emplea el Lenguaje de Modelado Unificado (UML¹⁰) para su modelo visual estándar. Es dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental. Los casos de uso son los componentes clave que conducen este modelo de proceso. Un caso de uso puede estar definido como una pieza de funcionalidad que da un resultado de un valor a un

⁹No va a necesitarlo

¹⁰De sus siglas en inglés Unified Modeling Language.

usuario. Todos los casos de uso desarrollados pueden ser combinados dentro de un modelo de casos de uso, que describe la funcionalidad completa del sistema. El modelo de casos de uso es análogo a la especificación funcional en un modelo de proceso tradicional. Los casos de uso son desarrollados con los usuarios y son modelados en UML. Estos representan los requerimientos para el software y son usados a lo largo del modelo de proceso. El Proceso Unificado es también descrito como centrado en la arquitectura. Esta arquitectura es una vista de todo el diseño con características importantes visibles omitiendo detalles. Funciona de la mano con los casos del uso. Subsistemas, clases y componentes son expresados en la arquitectura y también son modelados en UML. Por último el Proceso Unificado es iterativo e incremental. Las iteraciones representan pasos en un flujo de trabajo y los incrementos muestran crecimiento en funcionalidad del producto [Kan, 2002]. La base de los flujos de trabajo para el desarrollo iterativo es:

Requerimientos

Análisis

Diseño

Implementación

Prueba

El Proceso Unificado consiste de ciclos. Cada ciclo resulta en una nueva versión del sistema, y cada versión es un producto entregable. Cada ciclo tiene cuatro fases: inicio, elaboración, construcción y transición. Un número de iteraciones ocurre en cada fase, y los cinco flujos de trabajo esenciales tienen lugar en las cuatro fases. La matriz del proceso se muestra en la Figura 6.

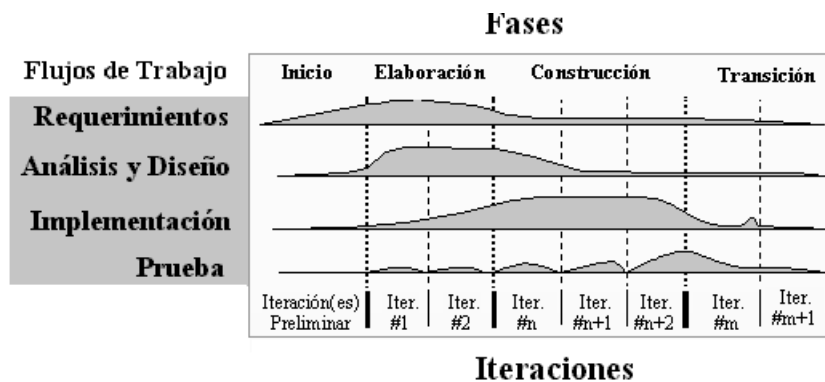


Figura 8: Matriz del Proceso Unificado

Fuente: [Jacobson, 2000]

Durante la fase de *Inicio*, se desarrolla una descripción del producto final a partir de una buena idea y se presenta el análisis de negocio para el producto. Un modelo de casos de uso es creado y los riesgos del proyecto son priorizados [Jacobson, 2000][Kan, 2002].

Durante la fase de *Elaboración*, se especifican en detalle la mayoría de los casos de uso del producto y se diseña la arquitectura del sistema. La gestión del proyecto empieza planificando los recursos y estimando las actividades. Todas las vistas del sistema son entregados, incluyendo el modelo de casos de uso, el modelo de diseño y el modelo de implementación. Estos modelos son desarrollados usando UML y bajo la gestión de configuración [Jacobson, 2000][Kan, 2002].

Durante la fase de *Construcción* se crea el producto, la línea base de la arquitectura crece hasta convertirse en el sistema completo. Se desarrolla el código y el software es probado. Luego el software es evaluado [Kan, 2002].

Finalmente, la fase de *Transición* empieza con un prueba beta. En esta fase, se rastrean los defectos y se los repara para que el software sea transmitido a un equipo de mantenimiento [Kan, 2002].

4. Análisis Orientado a Objetos

El Análisis Orientado a Objetos (AOO) tiene por objetivo definir todas las clases que son relevantes al problema que se resolverá, las operaciones y atributos asociados, las relaciones y comportamientos asociados con las clases, es decir una serie de modelos que describan el software al trabajar para satisfacer un conjunto de requerimientos definidos por el cliente. El modelo de análisis representa información, funcionamiento y comportamiento de los objetos.

En la década de los noventa, se propusieron varios métodos de análisis orientados a objetos, como Coad y Yourdon propusieron el método que lleva su nombre, que es considerado uno de los métodos más sencillos; Rumbaugh propuso el método la Técnica de Modelado de Objetos (OMT¹¹) cuya base fundamental es la definición del modelo de objetos, modelo funcional y modelo dinámico; Jacobson propuso el Objectory¹² que da importancia a la definición de

¹¹ OMT, del inglés Object Modeling Technique.

¹² Objectory, abreviación de Object Factory, Fábrica de Objetos.

casos de uso; y Booch propuso el método que lleva su nombre cuyo método abarca la definición de un micro y macro proceso. Finalmente Rumbaugh, Jacobson y Booch unieron sus métodos y experiencia para obtener un método unificado para el desarrollo de software orientado a objetos, este método que incluye su propio ciclo de proceso se denomina Proceso Unificado de Desarrollo de Software.

Según Pressman se deben ejecutar las siguientes tareas:

1. Los requerimientos básicos del usuario deben comunicarse entre el cliente y el equipo de desarrollo.
2. Identificar las clases, es decir, identificar atributos y métodos para las mismas.
3. Especificar una jerarquía de clases.
4. Representar las relaciones entre objetos.
5. Modelar el comportamiento de los objetos.
6. Repetir las tareas anteriores hasta completar el modelo de análisis orientado a objetos.

El producto final del AOO se compone de una representación gráfica, o basada en lenguaje natural, que define las características, relaciones y comportamiento de las clases identificadas.

4.1 Análisis del dominio

El análisis puede tener varios niveles de abstracción: A nivel de negocios o empresa, se puede definir clases, relaciones, comportamientos que modelen el negocio, usando técnicas del AOO e ingeniería de información. A nivel de área de negocios, donde se define un modelo de objetos que describen al área de aplicación. A nivel de aplicaciones, donde el modelo de objetos se centra en los requerimientos, debido a que la aplicación está en función a los requerimientos. El análisis del dominio se considera un nivel medio de abstracción, es decir, a nivel de área de negocios, y tiene lugar cuando una organización desea crear una biblioteca de clases reutilizables o componentes software ampliamente aplicables a una categoría completa de aplicaciones [Pressman. 2002].

El crear una biblioteca de clases dentro de una organización trae beneficios que son consecuencia de la reutilización, otros beneficios son la consistencia, el tiempo desarrollo, ventajas que están presentes debido a que la tecnología de objetos está fuertemente influenciada por la reutilización y para obtener las ventajas de de un biblioteca de clases la organización tiene que aplicar análisis del dominio.

4.1.1 Proceso del análisis del dominio

El análisis del dominio tiene el siguiente proceso:

1. Definir el dominio a investigar, está tarea implica la definición de elementos que se consideran orientados a objetos y elementos no orientados a objetos. Los elementos OO son las especificaciones existentes, código, interfaces a bases de datos, etc. Los elementos no OO son las políticas, planes, estándares, etc. Estos elementos están presente en un área específica que el analista debe aislar y deben ser comunes a otras áreas del dominio que no son necesariamente iguales.
2. Clasificar los elementos extraídos del dominio, está tarea significa establecer categorías, jerarquías y la definición de nomenclatura para cada elemento.
3. Análisis de los elementos, en esta etapa se deben identificar objetos candidatos reutilizables y justificar las razones por las cuales es considerado reutilizables, definir en que aplicaciones del dominio el objeto puede ser reutilizados.
4. Desarrollar un modelo de análisis para los objetos, este modelo servirá como base para el diseño e implementación de los objetos del dominio.

El objetivo principal del análisis del dominio es desarrollar software dentro de un dominio de aplicación con un alto porcentaje de componentes reutilizables, pero para esto la organización debe políticas para desarrollar componentes y organizarlos en una biblioteca que pueda servir en el futuro [Pressman, 2002].

4.2 Proceso del Análisis Orientado a Objetos

El AOO no comienza con la definición de objetos, sino comienza con la definición de que es lo que el sistema hará y en la manera en que se usará, esta recopilación de información que se obtiene del cliente y usuarios finales se recopila a través de técnicas y que después definen un modelo de AOO.

4.2.1 Casos de uso

Un caso de uso tiene dos componentes principales que son el actor, el caso de uso y la relación entre los mismos. Los casos de uso son fragmentos de funcionalidad que el sistema ofrece para aportar un resultado de valor para sus actores, es decir, los casos de uso modelan el sistema desde el punto de vista del usuario. El caso de uso especifica una secuencia de acciones que el sistema lleva a cabo interactuando con sus actores, incluyendo alternativas dentro de la secuencia. El objetivo de un caso de uso es definir los requisitos funcionales del sistema, diseñar un escenario de uso y proporcionar una descripción clara y sin ambigüedad de cómo el usuario final interactúa con el sistema; otro objetivo de un caso de uso es proporcionar una base para la validación de las pruebas [Pressman, 2002] [Jacobson, 2000].

Los casos de uso se agrupan en un modelo de casos de uso que pueden tener diferentes niveles de abstracción, lo importante es solo identificar requerimientos funcionales y dejar los requerimientos no funcionales para las siguientes etapas del proceso de desarrollo.

4.2.2 Modelado de clases, responsabilidades y colaboraciones

A partir de la identificación de los casos de uso se deben identificar las clases candidatas e indicar sus responsabilidades y colaboraciones. El modelado de clases, responsabilidades y colaboraciones (CRC) aporta un medio de identificar y organizar las clases que resulten relevantes al sistema o requerimientos del software. También sirve como medio para organizar las clases. Las responsabilidades son los atributos y operaciones relevantes para la clase, es decir, una responsabilidad es cualquier cosa que conoce o hace la clase. Los colaboradores son

aquellas clases necesarias para proveer a una clase con la información necesaria para completar una responsabilidad, es decir, una colaboración implica una solicitud de información o una solicitud de alguna acción [Pressman, 2002].

4.2.3 Definición de estructuras y jerarquías

A partir de la definición de las tarjetas CRC, se empieza a establecer la estructura del modelo de clases y la jerarquía entre clases y subclases con la ayuda de estructuras de generalización y especialización para las clases identificadas. También para definir la estructura se debe identificar si existen clases que están formadas por otras clases, las instancias de estas clases u objetos pueden representarse como una estructura de composición y agregación.

Las representaciones estructurales proveen los medios para dividir el modelo de tarjetas CRC y para dividir esta representación gráficamente. La expansión de cada clase aporta los detalles necesarios para la revisión y el diseño [Pressman, 2002].

4.2.4 Definición de subsistemas

Los subconjuntos de clases que colaboran entre sí para llevar a cabo un conjunto de responsabilidades, se les llama subsistemas o paquetes como se los conoce en UML. Los subsistemas o paquetes son abstracciones que aportan a los detalles del modelo de análisis.

Si se considera al subsistema como una caja negra que tiene responsabilidades y colaboraciones entre subsistemas. Un subsistema implementa un conjunto de solicitudes que los colaboradores pueden hacer al subsistema [Pressman, 2002].

4.3 Modelo Objeto-Relación

El modelo Objeto-Relación o modelo de objetos representa la relación entre las clases identificadas en las primeras etapas del análisis, representa tanto datos del sistema como su procesamiento; el modelo de objetos también es útil para mostrar cómo se clasifican las entidades en el sistema y se componen de otras entidades [Sommerville, 2005].

El modelo Objeto-relación puede obtenerse en tres etapas:

- Utilizando las tarjetas CRC, puede definirse una red de objetos colaboradores, es decir, mostrar las relaciones que pueden existir entre los objetos a través de una línea sin etiqueta.
- Revisando las tarjetas CRC, se evalúa las responsabilidades y colaboraciones y se etiqueta cada relación, para evitar ambigüedades, se debe especificar la dirección de la flecha.
- Una vez que las relaciones tienen un nombre, se debe evaluar cada lado de la relación para asignar una cardinalidad a la relación, existen cuatro opciones: 0 a 1, 1 a 1, 0 a muchos, ó 1 a muchos.

Las etapas anteriores se repiten hasta obtener un modelo objeto-relación completo, también se debe hacer notar que en la elaboración del modelo objeto-relación no solo se identifican las relaciones entre los objetos sino que también se identifican todas las vías importantes de mensajes que implican el intercambio de mensajes entre objetos y más adelante o a otro nivel de abstracción, el intercambio de mensajes entre subsistemas en el modelo [Pressman, 2002].

4.4 Modelo de herencia

Una vez que los objetos se identifican, estos deben ordenarse y agruparse en una taxonomía, el cual es un esquema de clasificación que muestra como una clase de objetos está relacionado con otras clases a través de atributos y servicios comunes. Para mostrar una taxonomía, las clases se organizan en una jerarquía de herencia con las clases de objetos más generales al inicio de la jerarquía. Los objetos más especializados pueden tener sus propios atributos y servicios.

El diseño de jerarquías no es fácil, debido a que el analista necesita comprender a detalle el dominio en el que el sistema será implantado. Las principales dificultades son el diseño de un grafo de herencia en donde los objetos no heredan objetos innecesarios, reorganizar el grafo de

herencia cuando se requieren cambios y resolver conflictos de nombres cuando varias clases tienen el mismo nombre pero diferente significado y puede ocasionar al momento de implementar en un lenguaje que no implemente ciertas características orientadas a objeto [Sommerville, 2005].

Una parte fundamental del modelo de herencia es el concepto de agregación, es decir, un objeto es un agregado de un conjunto de otros objetos, o dicho de otra forma, un objeto puede componerse de otros objetos.

4.5 Modelo Objeto-Comportamiento

Para modelar el comportamiento de los objetos, se tiene que mostrar las operaciones proporcionadas por los objetos. Se modelan los comportamientos utilizando escenarios que son representados como casos de uso, una forma de modelarlos es a través de diagramas de secuencia UML, que también incluye diagramas de colaboración que muestran la secuencia de mensajes intercambiados por los objetos [Sommerville, 2005].

El modelo de tarjetas CRC y el de objeto-relación representan elementos estáticos del modelo de análisis OO, se debe hacer una transición al comportamiento dinámico del sistema, es decir, se debe representar el comportamiento del sistema como una función de sucesos específicos y tiempo. El modelo objeto-comportamiento indica como responderá un sistema OO a sucesos externos o estímulos. Para crear un modelo objeto-comportamiento se deben realizar los siguientes pasos [Pressman, 2002]:

- Evaluar todos los casos de uso para comprender la secuencia de interacción dentro del sistema.
- Identificar sucesos de los casos de uso que dirigen la secuencia de interacción y comprender como estos sucesos se relacionan con objetos específicos, se debe entender un suceso entre un actor y el sistema al intercambio de información, pero haciendo notar que el suceso no es la información intercambiada sino al hecho que la información fu intercambiada. Una vez que todos los sucesos han sido identificados, se

asocian a objetos, los actores y objetos pueden responsabilizarse de la generación de sucesos o reconocimiento de sucesos que han ocurrido en otra parte.

- Construir un diagrama de transición de estados para el sistema, se debe considerar dos caracterizaciones de estados: el estado de cada objeto cuando el sistema ejecuta su función, y el estado del sistema observado desde el exterior cuando éste ejecuta su función. El estado de un objeto adquiere en ambos casos características pasivas y activas. Un estado pasivo es simplemente el estado actual de todos los atributos de un objeto, un estado activo es el estado actual cuando éste entra en una transformación continua o proceso. Para forzar la transición de un objeto de estado pasivo a activo debe ocurrir un suceso (a veces llamado disparador). El diagrama muestra una visión interna de la historia de vida de un objeto.
- Revisar el modelo objeto-comportamiento obtenido para verificar exactitud y consistencia.

5. Diseño Orientado a Objetos

Si se compara la ingeniería de software con la construcción de un puente, el proceso de análisis de requerimientos es como decidir donde debe comenzar el puente, dónde debe terminar y qué tipo de carga debe soportar. El diseñador del puente tiene que decidir si elige un puente suspendido, de vigas voladas, de cables o algún otro tipo para satisfacer los requerimientos [Braude, 2003].

El diseño orientado a objetos (DOO) requiere la definición de una arquitectura de software multicapa, la especificación de subsistemas que realizan funciones necesarias y proveen soporte a la infraestructura, la descripción de objetos¹³, que son los bloques de construcción del sistema y la descripción de los mecanismos de comunicación, que permiten que los datos fluyan entre las capas. Se divide en dos partes: el diseño del sistema que crea la arquitectura del producto, definiendo una serie de capas que cumplen funciones específicas del sistema e identifica clases, que son encapsuladas por los subsistemas que residen en cada capa, y el

¹³ clases

diseño de objetos se centra en detalles internos de cada clase, definición de atributos, operaciones y detalles de los mensajes [Pressman, 2002].

5.1 Métodos de Diseño Orientados a Objeto

A continuación se presentan algunos métodos de diseño según el resumen de Pressman (2002) :

Método Booch, abarca un proceso de macro desarrollo que en el contexto del diseño engloba una actividad de planificación arquitectónica, que agrupa objetos similares en particiones arquitectónicas que agrupa objetos similares en particiones arquitectónicas separadas crea un prototipo y valida el prototipo. El micro desarrollo define un conjunto de reglas que regulan el uso de operaciones y atributos y las políticas del dominio específico para la administración de memoria, manejo de errores y otras funciones. Crea un prototipo para cada política, instrumenta y refina el prototipo, y revisa cada política.

Método Rumbaugh, la técnica de modelado de objetos engloba una actividad de diseño que conduce a dos niveles de abstracción. El diseño de sistema se centra en el esquema de los componentes que se necesita para construir un sistema o producto completo. El modelo se divide en subsistemas los cuales se asignan a procesadores y tareas. El diseño de objeto enfatiza en el esquema detallado de un objeto individual. Se representan las estructuras de datos apropiadas para atributos y algoritmos, las clases y atributos son diseñados de manera que se optimice el acceso de los datos.

Método Jacobson, el modelo de diseño enfatiza la planificación para el modelo de análisis ISO 00. Los objetos de diseño primarios, llamados “bloques” son creados y catalogados como bloques de interfaz, bloques de identidades y bloques de control y la comunicación entre bloques se da mediante con la organización de subsistemas.

Método Coad & Yurdon, este método se desarrolla estudiando “como es que los diseñadores Orientado a objetos efectivos” hacen sus trabajo. Se enfoca en la representación de cuatro componentes mayores de sistema: la componente del dominio del problema, la componente de interacción humana, la componente de administración de tareas y la de datos.

5.2 Diseño del Sistema

En esta fase se desarrolla el detalle arquitectónico necesario para la construcción del sistema, las cuales abarca las siguientes actividades:

a) Particionar el modelo de análisis

Al igual que en la Ingeniería de Software convencional, se debe „particionar el modelo de análisis para definir colecciones congruentes de clases, relaciones y comportamiento; cada partición se llamara subsistema. Para entender los límites de cada subsistema, es necesario estratificar el sistema por capas, es decir niveles diferentes de abstracción, estos niveles se determinan por el grado en que el procesamiento asociado con el subsistema es visible al usuario final. Un enfoque utilizado para el diseño por capas es: establecer los criterios sobre la forma en que se agruparán los subsistemas y el número de capas, nombrar las capas y las clases encapsuladas que contienen, diseñar interfaces para cada capa, establecer la estructura de clases para cada capa y el formato de comunicación entre capas. Todos estos pasos aseguran un acoplamiento mínimo entre capas; si es necesario se puede iterar para lograr un diseño más fino [Pressman, 2002].

De acuerdo a lo mencionado anteriormente, la comunicación entre subsistemas se puede diferenciar como un enlace *cliente/servidor* o un enlace *punto a punto (peer to peer)* [Rumbaugh, 1991].

b) Asignación de subsistemas a procesadores y tareas

En algunos casos cuando las clases (o subsistemas) actúan en sucesos asincrónicamente, o bien cada subsistema de ir en procesadores independientes o si están en un mismo procesador se debe proporcionar soporte de concurrencia (tareas concurrentes), definidos gracias al diagrama de estados para cada objeto.

En este caso la utilización del concepto de Hilos es importante, puesto que se crean hilos de control donde cada uno realiza una tarea distinta, y de esta manera evitando la concurrencia.

c) Componente de administración de tareas

En el caso de tener tareas concurrentes, es importante el determinar la prioridad y criticidad de cada tarea, las características de cada una y las clases que tiene asociadas.

Para alcanzar la coordinación y comunicación entre tareas, se definen los atributos y métodos (operaciones) de las clases requeridas.

d) Componente de interfaz de usuario

La interfaz es un subsistema importante, pues tiene varios escenarios o casos de uso y la descripción de los roles que juega el actor.

Utilizando esta información se debe identificar una jerarquía de comandos u ordenes del menú del sistema y las sub funciones que contendrá.

Por último, la sola instanciación de las clases de un GUI son necesarias para desarrollar la interfaz de usuario.

e) Componentes de la Administración de Datos

La administración o gestión de datos trata de la administración de datos críticos para la propia aplicación y la creación de infraestructura para el almacenamiento y recuperación de objetos.

Se manipulan atributos del sistema (para los requerimientos de alto nivel) y estructura de datos (para los requerimientos de bajo nivel)

La manera de almacenar los atributos significativos según Coad y Yourdon (Coad, 1991), es crear una clase objeto – servidor, con el objeto de almacenar el objeto y recuperar los ya almacenados para utilizarlos por otros componentes de diseño y cuando sea requerido por el sistema.

f) Componentes de Gestión de Recursos

Los recursos de un sistema pueden ser las entidades externas o las abstracciones, sea cual fuere, cada entidad debe ser poseída por un “objeto guardián”, que es quien controla el acceso y peticiones para tal entidad.

g) Comunicación entre Subsistemas

Dentro este nivel se puede utilizar un modelo de colaboración entre subsistemas y se debe poder establecer un enlace cliente/servidor o punto a punto.

Un contrato para un subsistema puede seguir las siguientes etapas:

1. Listar cada petición que puede ser realizada por los colaboradores del subsistema.
2. Para cada contrato, anotar las operaciones que se requieran para implementar las responsabilidades que implica el contrato.
3. Crear una tabla con las columnas: nombre del contrato, tipo, colaboradores, clase(s), operación(es), formato de mensaje.
4. Si la forma de interacción entre los subsistemas son complejos, debe crear un diagrama de colaboración entre subsistemas.

5.3 Diseño de Clases

En esta fase definimos el diseño de las clases (atributos, operaciones) y sus interacciones con otras clases.

a) Descripción de Clases

Se define el número de operaciones más específicas de un requerimiento del Análisis Orientado a Objetos.

Dependiendo de lo que necesite el diseñador o desarrollador, se puede describir el objeto a nivel de protocolo o realizar una descripción a nivel de la implementación. La descripción a nivel de protocolo, trata de detallar la interfaz de un objeto, los mensajes que puede recibir y las operaciones que lleva a cabo.

La descripción de la implementación incluye información privada del objeto, la estructura de datos y detalles de los atributos y procedimientos.

b) Diseño de Algoritmos y estructuras de datos

Se diseñan simultáneamente, de la misma manera que en el diseño a nivel de componentes, en donde si una operación es compleja, será necesario modularizar la operación.

Las operaciones pueden manipular los datos, ejecutar cálculos, monitorizar al objeto para la convivencia de un suceso controlado.

5.4 Patrones de Diseño

Los patrones de diseño describen un problema que ocurre repetidas veces en algún contexto determinado de desarrollo de software, y entregan una buena solución ya probada. Esto ayuda a diseñar correctamente en menos tiempo, ayuda a construir problemas reutilizables y extensibles, facilita la documentación, y facilita la comunicación entre los miembros del equipo de desarrollo [Pressman, 2002].

Un patrón de diseño debe proporcionar:

Nombre del patrón

Intención del patrón

Los problemas de diseño que motivan el patrón

La solución que resuelve estos problemas

Las clases necesarias para implementar su solución

Las responsabilidades y colaboraciones entre las clases

Lineamientos que conduzcan a una implementación efectiva

Ejemplos de código fuente

Referencias a otros patrones de diseño

5.5 Enfoque convencional vs. OO

Cada elemento del modelo convencional de análisis se corresponde con uno o más capas del modelo de diseño al igual que el diseño convencional de software. El DOO aplica el diseño de datos cuando los atributos son representados, el diseño de interfaz cuando se desarrolla un modelo de mensajería, y diseño a nivel de componentes. El diseño de subsistemas se deriva considerando los requerimientos del cliente (representados por los casos de uso) y los sucesos y estados que son externamente observables (el modelo de comportamiento de objetos). El diseño de clases y objetos es trazado de la descripción de atributos, operaciones y colaboraciones contenidas en el modelo CRC. El diseño de mensajes es manejado por el modelo objeto-relación y el diseño de responsabilidades es derivado del uso de atributos, operaciones y colaboraciones descritos en el modelo CRC.

Fichman y Kemerer sugieren 10 componentes de diseño modelado que puede usarse para comparar varios métodos convencionales y orientados a objetos:

1. Representación de la jerarquía de módulos.
2. Especificación de las definiciones de datos.
3. Especificación de la lógica procedimental.
4. Indicación de secuencias de proceso final-a-final
5. Representación de estados y transiciones de los objetos.
6. Definición de clases y jerarquías
7. Asignación de operaciones a las clases
8. Definición detallada de operaciones
9. Especificación de conexiones de mensajes
10. Identificación de servicios exclusivos

6. Programación Orientada a Objetos

Los principios deben descubrirse, articularse y probarse primero. Si un principio general sobrevive los rigores de las pruebas y demuestra ser una herramienta útil, entonces resulta

tonto no aplicarlo en la práctica. La programación estructurada presenta un conjunto de principios para la comprensión de los sistemas de software que son el resultado de años de observación y estudio de las técnicas de construcción de programas. *Por tanto, la programación orientada a objetos es otro paso evolutivo en el diseño y construcción de software.*

La programación orientada a objetos es una técnica de estructuración. En ella, los objetos son los principales elementos de construcción. Sin embargo, comprender simplemente lo que es un objeto o utilizar objetos en un programa no significa que usted esté programando en una forma orientada a objetos. Lo que cuenta es la forma cómo los objetos se conectan entre sí, permitiendo que el código fuente sea reutilizable, portable, y fácil de modificar. Por otro lado, la programación orientada a objetos es un paradigma de programación que define los programas en términos de "clases de objetos", objetos que son entidades que combinan estado (es decir, datos), comportamiento (esto es, procedimientos o métodos) e identidad (propiedad del objeto que lo diferencia del resto) [Deitel, 2000].

De esta forma, un objeto contiene toda la información, (los denominados atributos) que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases (e incluso entre objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos). A su vez, dispone de mecanismos de interacción (los llamados métodos) que favorecen la comunicación entre objetos (de una misma clase o de distintas), y en consecuencia, el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separan (ni deben separarse) información (datos) y procesamiento (métodos) [Braider, 2001].

Dos pilares fundamentales de la programación orientada a objetos en la herencia y el polimorfismo. La herencia es una forma de reutilización del software en la que se crean nuevas clases a partir de clases ya existentes. El polimorfismo permite escribir programas de forma general para manejar una amplia variedad de clases interrelacionadas existentes [Deitel, 2000].

6.1 Características importantes en la programación orientada a objetos

Hay un cierto desacuerdo sobre exactamente qué características de un método de programación o lenguaje le definen como "orientado a objetos", pero hay un consenso general en que las características siguientes son las más importantes:

Abstracción: cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar cómo se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos y cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción.

Encapsulamiento: también llamado "ocultación de la información". Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.

Polimorfismo: comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama asignación tardía o asignación dinámica. Algunos lenguajes proporcionan medios más estáticos (en

"tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++ [Deitel, 2000].

Herencia: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementar su comportamiento. Esto suele hacerse habitualmente agrupando los objetos en clases y estas en árboles o enrejados que reflejan un comportamiento común. Cuando un objeto pertenece a más de una clase se llama herencia múltiple; esta característica no está soportada por algunos lenguajes (como Java¹⁴) [Deitel, 2000].

Demonios: Es un tipo especial de métodos, relativamente poco frecuente en los sistemas de POO, que se activa automáticamente cuando sucede algo especial. Es decir, es un programa, como los métodos ordinarios, pero se diferencia de estos porque su ejecución no se activa con un mensaje, sino que se desencadena automáticamente cuando ocurre un suceso determinado: la asignación de un valor a una propiedad de un objeto, la lectura de un valor determinado, etc. Los demonios, cuando existen, se diferencian de otros métodos por que no son heredables y porque a veces están ligados a una de las propiedades de un objeto, mas que al objeto entero [Braider, 2001] [Deitel, 2000].

En la programación o implementación como también se la conoce, implementamos el sistema en términos de componentes, es decir, archivos de código fuente, ejecutables, etc.

7. Pruebas Orientadas a Objeto

El objetivo de las pruebas es encontrar el mayor número posible de errores con una cantidad razonable de esfuerzo, en un determinado tiempo. Otro objetivo esencial de las pruebas es para demostrar al desarrollador y al cliente que el software satisface sus requerimientos, esto

¹⁴ Java, lenguaje de programación orientado a objetos, propuesto por Sun Microsystems Inc.

conduce a las pruebas de validación. La prueba de los sistemas OO presenta un nuevo conjunto de retos al ingeniero del software. La definición de pruebas debe ser ampliada para incluir técnicas que descubran errores, aplicadas para los modelos de AOO y DOO. La integridad y consistencia de las representaciones OO deben ser evaluadas conforme se construyen. Las pruebas de unidad pierden mucho significado, y las estrategias y tácticas de prueba deben tomarse en cuenta para las características propias del software OO. [Pressman, 2002], [Sommerville, 2005]

7.1 Exactitud de los modelos de AOO y DOO

Durante el análisis y diseño, la exactitud semántica debe juzgarse basada en la conformidad del modelo de dominio del problema en el mundo real. Si el modelo refleja con exactitud el mundo real, entonces es semánticamente correcto. Para determinar esto se deben presentar los modelos a expertos en el dominio quienes examinan las definiciones de clases y sus jerarquías, para detectar omisiones o ambigüedades. Las relaciones entre clases se evalúan para determinar si reflejan con exactitud conexiones del mundo real.

7.2 Consistencia de los modelos de AOO y DOO

La consistencia de los modelos de AOO y DOO debe juzgarse considerando las relaciones entre entidades dentro del modelo. Un modelo inconsistente tiene representaciones en un aparte, que no se reflejan correctamente en otras partes del modelo.

Para evaluar la consistencia, se debe examinar cada clase y las relaciones con otras. Un modelo de tarjetas CRC y un diagrama objeto - relación se utilizan para esta tarea. Se recomienda los siguientes pasos:

- Revisar el modelo CRC y objeto - relación, hacer un control para verificar que todas las colaboraciones implicadas en el modelo de AOO hayan sido representadas adecuadamente.

- Inspeccionar la descripción de cada tarjeta CRC para determinar si alguna responsabilidad delegada es parte de la definición del colaborador.
- Invertir la conexión de cada colaborador para asegurarse que solicita un servicio de una fuente razonable.
- Determinar si las responsabilidades muy solicitadas, deben combinarse en una sola responsabilidad.

También deben llevarse a cabo las revisiones del diseño del sistema y del diseño de objetos. El diseño del sistema describe el, producto arquitectónico global, los subsistemas, la asignación de clases a subsistemas y el diseño de la interfaz de usuario. El diseño de objetos presenta los detalles de cada clase, y las actividades de envío de mensajes necesarios para implementar las colaboraciones entre clases. El diseño del sistema se revisa examinando el modelo objeto – comportamiento y la correspondencia del comportamiento del sistema, frente a los subsistemas diseñados para lograra este comportamiento.

El modelado de objetos debe probarse con el modelo objeto – relación, para asegurarse de que todos los objetos del diseño contienen los atributos y operaciones necesarias y para implementar las colaboraciones que se definieron el modelo CRC.

7.3 Estrategias de Pruebas Orientadas a Objetos

La estrategia clásica de prueba del software comienza con las pruebas de unidad, continuando con las pruebas de integración, y termina con las pruebas de validación del sistema.

7.3.1 Pruebas de Unidad en el contexto OO

En un entorno orientado a objetos, el concepto de unidad es distinto, debido a la encapsulación que conduce a la definición de clases y objetos, por la unidad a probar será una clase u objeto y también se centra en el comportamiento de la clase a través del análisis de sus métodos.

7.3.2 Pruebas de Integración en el contexto OO

Este tipo de pruebas implica identificar grupos de clases que proporcionan alguna funcionalidad del sistema e integrar estos para hacer que funcionen conjuntamente. Existen dos estrategias de prueba de integración de los sistemas OO. El primero denominado pruebas basadas en hilos, que integran el conjunto de clases requeridas, para responder una entrada o suceso al sistema. La segunda denominada prueba basada en uso, comienza la construcción del sistema probando aquellas clases independientes que utiliza muy pocas clases servidores, luego se toman las clases dependientes. [Sommerville, 2005]

7.3.3 Pruebas de validación en el contexto OO

La validación del software OO se centra en las acciones visibles al usuario y salidas reconocibles desde el sistema, se deben utilizar casos de uso que proporcionan un escenario para reconocer errores en la interacción del usuario. Los métodos de prueba de caja negra pueden usarse para realizar pruebas de validación. [Pressman, 2005]

7.4 Diseño de casos de prueba

El diseño de casos de prueba consiste en proporcionar entradas y salidas esperadas para probar el sistema, este conjunto de entradas y salidas deben ser efectivos descubriendo defectos en los programas y muestren que el sistema satisface los requerimientos [Sommerville, 2005].

Existen varias aproximaciones que pueden servir para diseñar casos de prueba:

- Pruebas basadas en requerimientos, esta aproximación se utiliza principalmente en la etapa de prueba del sistema, debido a que los requerimientos normalmente se implementan por varios componentes.
- Pruebas estructurales, en donde se utiliza el conocimiento de la estructura del programa para diseñar pruebas que ejecuten todas las partes del sistema, es decir que se debe ejecutar una sentencia al menos una vez.

8. Lenguaje de Modelado Unificado (UML)

UML es el producto de un equipo que comenzó en octubre de 1994, cuando Rumbaugh se unió a Booch en Rational. El objetivo inicial del proyecto fue la unificación de los métodos de Booch y OMT. En octubre de 1995 se hizo la primera publicación del Método Unificado (versión 0.8), en esa época Jacobson se unió a Rational y el alcance del proyecto UML se amplió para incorporar OOSE. En 1997, el grupo inicial de colaboradores se amplió para incluir prácticamente a todas las demás organizaciones que habían enviado alguna propuesta o habían contribuido en alguna medida en las respuestas iniciales al OMG [Rumbaugh, 2000].

8.1 Objetivo de UML

UML tiene varios objetivos el mas importante es el de estandarizar el lenguaje de modelado en la comunidad informática. Es un lenguaje para visualizar, especificar, construir y documentar.

Área	vista	Diagramas	Conceptos Principales
estructural	Vista estática	Diagrama de clases	Clase, asociación, generalización, dependencia, realización, interfaz.
	Vista de casos de uso	Diagrama de casos de uso	Caso de uso, actor, asociación, extensión, inclusión, generalización de casos de uso.
	Vista de implementación	Diagrama de componentes	Componente, interfaz, dependencia, realización
	Vista de despliegue	Diagrama de despliegue	Nodo, componente, dependencia, localización,
Dinámica	Vista de maquina de estados	Diagrama de estados	Estado, evento, transición, acción.
	Vista de actividad	Diagrama de actividad	Estado, actividad, transición de terminación, división, unión.
	Vista de interacciona	Diagrama de secuencia	Interacción, objeto, mensaje, activación.
		Diagrama de colaboración	Colaboración, interacción, rol de colaboración, mensaje.
Gestión del modelo	Vista de gestión del modelo	Diagrama de clases	Paquete, subsistema, modelo.
Extensión del UML	todas	todos	Restricción, estereotipo, valores etiquetados.

Tabla 1. Vistas y diagramas del UML

Fuente: [Rumbaugh, 2000]

8.2 Vista Estática

La vista estática modela los conceptos del dominio de la aplicación, así como los conceptos internos inventados como parte de la implementación de la aplicación. Esta visión es estática porque no describe el comportamiento del sistema dependiendo del tiempo.

8.2.1 Diagrama de clases

Los diagramas de clases son utilizados para modelar el vocabulario del sistema, modelara las colaboraciones o modelar esquemas. Los elementos del diagrama de clases son los siguientes: clases, interfaces, colaboraciones, relaciones de dependencia, generalización y asociación.

8.3 Vista de Casos de Uso

Captura el comportamiento de un sistema, de un subsistema, o de una clase, tal como sé nuestra a un usuario exterior. Los elementos de los casos de uso son: los actores, representados por monigotes, los casos de uso representados por óvalos y el ambiente representado por un rectángulo, que es lo que divide a los casos de uso de los actores [Rumbaugh, 2000].

8.3.1 Actor

Un actor es una idealización de una entidad que interactúa con el sistema a ser modelado, esta entidad puede ser una persona, un sistema, o un proceso. Los actores son los que interactúan con los casos de uso, esta interacción se representa mediante líneas que unen a los actores con los casos de uso. El actor esta representado en el diagrama por un monigote, y lleva debajo su nombre.

8.3.2 Casos de uso

Un caso de uso s una unidad de funcionalidad, externamente visible, proporcionada por una unidad del sistema y expresada por secuencias de mensajes intercambiados por la unidad del

sistema y uno o mas actores. Es importante mencionar que un caso de uso representa lo que se quiere que haga el sistema y no así, como se realizara. El caso de uso esta representado por un ovalo, lleva su nombre dentro o debajo de ella.




Relación	Función	Notación
Asociación	La línea de comunicación entre un actor y un caso de uso en el que participa	
Extensión	La inserción de comportamiento adicional en un caso de uso base que no tiene conocimiento sobre el	Extend 
Generalización de casos de uso	Una relación entre un caso de uso general y un caso de uso mas especifico, que hereda y añade propiedades a aquel	
Inclusión	Inserción de comportamiento adicional en un caso de uso base, que describe explícitamente la inserción.	include

Tabla 2: Tipos de relaciones de casos de uso
Fuente: [Rumbaugh, 2000]

8.4 Vista de Interacción

La vista de interacción describe secuencias de intercambios de mensajes entre los roles que implementan el comportamiento de un sistema. El “rol” es la descripción de un objeto, que desempeña un determinado papel dentro una interacción [Rumbaugh, 2000].

8.4.1 Diagrama de Secuencia

Un diagrama de secuencia muestra un conjunto de mensajes, dispuestos en una secuencia temporal. Cada rol en la secuencia se muestra como una línea de vida, los mensajes se muestran como flechas entre las líneas de vida. Cada mensaje en un diagrama de secuencia corresponde a una operación en una clase, a un evento disparador, o a una transición en una máquina de estados. La información de control implica una *condición* (el cuándo se envía un mensaje) y el *marcador de iteración* que muestra que un mensaje se envía muchas veces a varios objetos receptores.

8.4.2 Diagrama de Colaboración

Una colaboración modela los objetos y los enlaces significativos dentro de una interacción. Un rol describe un objeto, y un rol en la asociación describe un enlace dentro una colaboración. En estos diagramas los objetos se muestran como íconos y las flechas como los mensajes enviados dentro del caso de caso de uso. Un uso de un diagrama de colaboración es mostrar la implementación de una operación. La colaboración muestra los parámetros y las variables locales de la operación, así como asociaciones más permanentes.

8.5 Vista de la Máquina de estados

Una máquina de estados modela las posibles historias de vida de un objeto de una clase, contiene los estados conectados por transiciones. Cada estado modela un período de tiempo, durante la vida de un objeto, en el que satisface ciertas condiciones [Rumbaugh, 2000].

Cuando ocurre un evento, se puede desencadenar una transición que lleve el objeto a un nuevo estado. Cuando se dispara una transición, se puede ejecutar una acción unida a la transición. La máquina de estados se muestra como diagrama de estados. Las máquinas de estados se pueden utilizar para describir mecanismos de control como interfaces de usuario, controladores de dispositivo y otros subsistemas reactivos. También pueden usarse para describir los objetos pasivos que pasan por varias fases cualitativas distintas, durante su tiempo de vida, cada una de las cuales tiene su propio comportamiento especial.

8.5.1 Diagrama de Estados

Los diagramas de estados describen el comportamiento de un sistema. Describen todos los estados posibles en los que puede entrar un objeto particular y la manera en que cambie el estado del objeto, como resultado de los eventos que llegan a él. Cada objeto puede reaccionar de diferente forma al mismo evento cuando están en diferentes estados.

Un *evento* es una ocurrencia significativa que tiene una localización en tiempo y espacio (no tiene duración). Los eventos representan las clases de cambios que un objeto puede detectar: la recepción de llamadas o señales explícitas desde un objeto a otro, un cambio en ciertos valores, o el paso del tiempo.

Un *estado* describe un período de tiempo durante la vida de un objeto de una clase, caracterizado como un conjunto de valores cualitativos similares, como un período de tiempo donde un objeto espera cierta actividad o realiza cierta actividad.

Un *estado compuesto* es un estado que se ha descompuesto en sub estados secuenciales o sub estados concurrentes. Puede tener estados anidados, la transición es elegible para ser disparada siempre que cualquier estado anidado esté dentro. Los estados compuestos son útiles para expresar condiciones de excepción y de error, porque las transiciones en ellos se aplican a todos los estados anidados sin necesidad de que cada estado anidado maneje la excepción explícitamente.

Una *transición* conecta dos estados (o más, si hay una división o unión de control); cuando deja un estado define la respuesta de un objeto en ese estado a la ocurrencia de un evento, es decir, un evento disparador, una condición de guarda, una acción (externa o de finalización), y un estado destino, activan ese estado.

Entre los tipos de estados más comunes están:



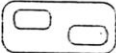





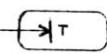
<i>Tipo de estado</i>	<i>Descripción</i>	<i>Notación</i>
estado simple	Un estado sin estructura	
estado concurrente compuesto	Un estado que está dividido en dos o más estados concurrentes, todos los cuales están activos concurrentemente cuando el estado compuesto está activo	
estado secuencial compuesto	Un estado que contiene uno o más subestados disjuntos, exactamente uno de los cuales está activo en cada momento cuando el estado compuesto está activo	
estado inicial	Un pseudoestado que indica el estado inicial cuando es invocado el estado que lo engloba	
estado final	Un estado especial cuya activación indica que el estado que lo engloba ha completado su actividad	
estado de unión o conjunción	Un pseudoestado que encadena segmentos de transición en una sola transición atómica	
estado de historia	Un pseudoestado cuya activación restaura el estado activado previamente dentro de un estado compuesto	
estado de referencia a submáquina	Un estado que referencia a una submáquina, la cual está implícitamente insertada en el lugar del estado de referencia a submáquina	
estado abreviado externo	Un pseudoestado, dentro de un estado de referencia a submáquina, que identifica un estado en la referida	

Tabla 3: Tipos de estados
Fuente: [Rumbaugh, 2000]

8.6 Vista de Actividades

La vista de actividades muestra las actividades de computación implicadas en la ejecución de un cálculo. Un estado de actividad representa una actividad: un paso en el flujo de trabajo o la ejecución de una operación (una sentencia en un procedimiento). Un grafo de actividades describe grupos secuenciales y concurrentes de actividades. Los grafos de actividades se muestran en diagramas de actividades.

8.6.1 Diagrama de Actividades

Las flechas muestran dependencias secuenciales y las barras horizontales representan bifurcaciones o uniones de control que se ejecutan concurrentemente. A menudo es útil organizar las actividades en un modelo según su responsabilidad, para ello se utilizan „calles que separan por líneas en el diagrama las actividades en regiones.

Un diagrama de actividades puede contener bifurcaciones o divisiones de control. Entre los usos de un diagrama de actividades están el modelar procesos de negocios o actividades software, entender el comportamiento de alto nivel de la ejecución de un sistema.

8.7 Vista Física

Las vistas físicas modelan la estructura de la implementación de la aplicación por sí misma, su organización en componentes, y su despliegue en nodos ejecución. Hay dos vistas físicas: la vista de implementación y la vista de despliegue.

8.8 Vista de Implementación

La vista de implementación modela los componentes de un sistema – a partir de los cuales se construye la aplicación – así como las dependencias entre los componentes, para poder determinar el impacto de un cambio propuesto. También modela la asignación de clases y de otros elementos del modelo a los componentes. La vista de implementación se modela en diagramas de componentes.

8.8.1 Diagrama de Componentes

El diagrama de componentes muestra los tipos de componentes del sistema, es decir, muestra la organización y las dependencias entre un conjunto de componentes. Un círculo pequeño con un nombre es una interfaz; una línea sólida que va desde un componente a una interfaz, indica que el componente proporciona los servicios de la interfaz.

Una flecha de guiones de un componente a una interfaz indica que el componente requiere los servicios proporcionados por interfaz.

8.9 Vista de Despliegue

La vista de despliegue representa la disposición de las instancias de componentes de ejecución en instancias de nodos. Un nodo es un recurso de ejecución, tal como una computadora, un dispositivo o memoria. Esta vista permite determinar las consecuencias de la distribución y de la asignación de recursos. La vista de despliegue se representa en diagramas de despliegue.

8.9.1 Diagrama de Despliegue

Los diagramas de despliegue muestran la configuración de nodos que participan en la ejecución y de los componentes que residen en ellos. Son importantes para visualizar, especificar y documentar sistemas empotrados, sistemas cliente/servidor y sistemas distribuidos, sino para gestionar sistemas ejecutables mediante ingeniería inversa y directa.

8.10 Vista de Gestión del Modelo

La vista de gestión del modelo modela la organización del modelo en sí mismo. Un modelo abarca un conjunto de paquetes que contienen los elementos del modelo, tales como clases, máquinas de estados, y casos de uso. Los paquetes pueden contener otros paquetes, por lo tanto, un modelo señala un paquete raíz, que contiene indirectamente todo el contenido del modelo. Los paquetes son unidades para manipular el contenido de un modelo, así como unidades para el control de acceso y el control de configuración. Cada elemento del modelo pertenece a un paquete o a otro elemento.

Un modelo es una descripción completa de un sistema, con una determinada precisión, se lo representa como una clase especial de paquete. Un subsistema es otro paquete especial. Representa una porción de un sistema, con una interfaz perfectamente determinada que puede ser implementado como un componente distinto. Generalmente, la información de gestión del modelo se representa en diagrama de clases.

8.11 Construcciones de Extensión

UML incluye tres construcciones principales de extensión: restricciones, estereotipos y valores etiquetados.

Una restricción es una declaración textual de una relación semántica expresada en un cierto lenguaje formal o en lenguaje natural. Un estereotipo es una nueva clase de elemento del modelo, ideada por el modelador, y basada en un tipo existente de elemento del modelo. Un valor etiquetado es una porción de información con nombre, unida a cualquier elemento del modelo.

Conclusiones

El paradigma orientado a objetos es un enfoque que ha tomado fuerza a medida que las exigencias del mercado crecen, cada vez se necesita software más complejo, que administre todo tipo de datos desde texto hasta datos multimedia, la presente monografía pretende dar una introducción al estado del arte en el que investigadores y desarrolladores se encuentran. Nuevos enfoques empiezan a surgir como las metodologías ágiles que están pensadas para ciertos tipos de proyectos; los modelos de proceso y las etapas que se mostraron constituyen lo que se denomina metodologías pesadas, pero está pensada para proyectos de desarrollo de gran magnitud en donde los requerimientos no están bien definidos, los equipos de desarrollo son grandes, los costos son altos y el software es complejo. No se pretende que lo descrito sea tomado al pie de la letra, investigadores continúan cuestionado y favoreciendo a ciertas técnicas que pueden adaptarse al proyecto, lo que se pretende es mostrar un panorama de lo se considera debería estudiar un desarrollador orientado a objetos al momento de encarar un proyecto.

BIBLIOGRAFÍA

[Boehm, 2001], Boehm, Barry, "The Spiral Model as a Tool for Evolutionary Software Acquisition". Disponible en <http://www.stcs.hill.af.mil/crosstalk/2001/05/boehm.html>

[Braider, 2001], Blaider Correa, Eddy, "Programación en Java 2", Alfaomega.

[Braude, 2003], Braude, Eric, "Ingeniería de Software: Una perspectiva orientada a objetos", Alfaomega.

[Deitel, 2000], Deitel, H.M., "Cómo programar en Java", Primera Edición, Prentice Hall.

[Fowler, 1997], Fowler, Martin "UML gota a gota". Addison Wesley Longman, 1997.

[Kan, 2002], Kan, H., Stephen, "Metrics and Models in Software Quality Engineering-Second Edition", Addison Wesley.

[Jacobson, 2000], Jacobson, Ivar, Booch, Grady, Rumbaugh James, "El Proceso Unificado de Desarrollo de Software", Pearson Addison Wesley.

[Lawrence, 2002], Lawrence, Pfleeger, Shari "Ingeniería del Software Teoría y práctica", Prentice Hall.

[Mahonet, 2004], Mahoney, Michael, "Finding a History for Software Engineering".

[Pressman, 2002], Pressman, Roger, "Ingeniería de Software: Un enfoque práctico", McGrawhill, 5ª. Edición.

[Pressman, 2005], Pressman, Roger, "Software Engineering: A Practitioners Approach", MacGrawhill, Sixth Edition.

[Rumbaugh, 2000], Rumbaugh, James, Jacobson, Ivar, Booch, Grady, "El lenguaje unificado del modelado". Addison Wesley Longman, 1999.

[Rumbaugh, 2000], Rumbaugh, James, Jacobson, Ivar, Booch, Grady, "El lenguaje unificado del modelado. Manual de referencia". Addison Wesley Longman, 1999.

[Sommerville, 2005], Sommerville, Ian, "Ingeniería del Software", Pearson Addison Wesley, 7ª. Edición.

Enlaces Visitados:

<http://www.rational.com/uml/>. Sitio oficial de Rational Software Inc Fecha de Acceso: Julio, 2006