UDS

ANTOLOGIA

Ingeniería de software

LICENCIATURA EN INGENIERIA EN SISTEMAS COMPUTACIONALES

OCTAVO CUATRIMESTRE

Marco Estratégico de Referencia

ANTECEDENTES HISTORICOS

Nuestra Universidad tiene sus antecedentes de formación en el año de 1979 con el inicio de actividades de la normal de educadoras "Edgar Robledo Santiago", que en su momento marcó un nuevo rumbo para la educación de Comitán y del estado de Chiapas. Nuestra escuela fue fundada por el Profesor de Primaria Manuel Albores Salazar con la idea de traer Educación a Comitán, ya que esto representaba una forma de apoyar a muchas familias de la región para que siguieran estudiando.

En el año 1984 inicia actividades el CBTiS Moctezuma Ilhuicamina, que fue el primer bachillerato tecnológico particular del estado de Chiapas, manteniendo con esto la visión en grande de traer Educación a nuestro municipio, esta institución fue creada para que la gente que trabajaba por la mañana tuviera la opción de estudiar por las tarde.

La Maestra Martha Ruth Alcázar Mellanes es la madre de los tres integrantes de la familia Albores Alcázar que se fueron integrando poco a poco a la escuela formada por su padre, el Profesor Manuel Albores Salazar; Víctor Manuel Albores Alcázar en septiembre de 1996 como chofer de transporte escolar, Karla Fabiola Albores Alcázar se integró como Profesora en 1998, Martha Patricia Albores Alcázar en el departamento de finanzas en 1999.

En el año 2002, Víctor Manuel Albores Alcázar formó el Grupo Educativo Albores Alcázar S.C. para darle un nuevo rumbo y sentido empresarial al negocio familiar y en el año 2004 funda la Universidad Del Sureste.

La formación de nuestra Universidad se da principalmente porque en Comitán y en toda la región no existía una verdadera oferta Educativa, por lo que se veía urgente la creación de una institución de Educación superior, pero que estuviera a la altura de las exigencias de los jóvenes que tenían intención de seguir estudiando o de los profesionistas para seguir preparándose a través de estudios de posgrado.

Nuestra Universidad inició sus actividades el 18 de agosto del 2004 en las instalaciones de la 4ª avenida oriente sur no. 24, con la licenciatura en Puericultura, contando con dos grupos de cuarenta alumnos cada uno. En el año 2005 nos trasladamos a nuestras propias instalaciones en la carretera Comitán – Tzimol km. 57 donde actualmente se encuentra el campus Comitán y el Corporativo UDS, este último, es el encargado de estandarizar y controlar todos los procesos operativos y Educativos de los diferentes Campus, Sedes y Centros de Enlace Educativo, así como de crear los diferentes planes estratégicos de expansión de la marca a nivel nacional e internacional.

Nuestra Universidad inició sus actividades el 18 de agosto del 2004 en las instalaciones de la 4ª avenida oriente sur no. 24, con la licenciatura en Puericultura, contando con dos grupos de cuarenta alumnos cada uno. En el año 2005 nos trasladamos a nuestras propias instalaciones en la carretera Comitán – Tzimol km. 57 donde actualmente se encuentra el campus Comitán y el corporativo UDS, este último, es el encargado de estandarizar y controlar todos los procesos operativos y educativos de los diferentes campus, así como de crear los diferentes planes estratégicos de expansión de la marca.

MISIÓN

Satisfacer la necesidad de Educación que promueva el espíritu emprendedor, aplicando altos estándares de calidad Académica, que propicien el desarrollo de nuestros alumnos, Profesores, colaboradores y la sociedad, a través de la incorporación de tecnologías en el proceso de enseñanza-aprendizaje.

VISIÓN

Ser la mejor oferta académica en cada región de influencia, y a través de nuestra Plataforma Virtual tener una cobertura Global, con un crecimiento sostenible y las ofertas académicas innovadoras con pertinencia para la sociedad.

VALORES

- Disciplina
- Honestidad
- Equidad
- Libertad

ESCUDO

El escudo de la UDS, está constituido por tres líneas curvas que nacen de izquierda a derecha formando los escalones al éxito. En la parte superior está situado un cuadro motivo de la abstracción de la forma de un libro abierto.

ESLOGAN

"Mi Universidad"

ALBORES

Es nuestra mascota, un Jaguar. Su piel es negra y se distingue por ser líder, trabaja en equipo y obtiene lo que desea. El ímpetu, extremo valor y fortaleza son los rasgos que distinguen.

Ingeniería de Software

Objetivo de la materia:

El principal objetivo de la asignatura es fomentar la capacidad del alumno para aplicar de forma efectiva los principios de la Ingeniería del software al desarrollar un proyecto software completo, desde la fase de especificación hasta la entrega al cliente. Acercaremos al alumno el ciclo de vida de un proyecto de software y a la necesidad de llevarlo a cabo para desarrollar un software de garantía. Para ello, se estudiarán cada una de las fases del ciclo de vida presentando técnicas y herramientas que generen la documentación necesaria para cada fase, tratando las metodologías básicas de especificación de requisitos, diseño, implantación y pruebas.

Índice

Unidad I FUNDAMENTOS DE LA INGENIERÍA DEL SOFTWARE

1.1. DEFINICIÓN Y OBJETIVOS DE LA INGENIERÍA DEL SOFTWARE

1.2. CARACTERISITICAS Y APLICACIONES DEL SOFTWARE

1.3. EVOLUCIÓN HISTÓRICA DEL SOFTWARE

1.4 LEYES DE EVOLUCIÓN DEL SOFTWARE

1.5 PARADIGMAS DE SOFTWARE

1.6 PERSPECTIVA GENERAL DE LA INGENIERIA DEL SOFTWARE

1.7 PROCESOS, MÉTODOS Y HERRAMIENTAS

1.8 MODELO CLÁSICO O LINEAL, MODELO EN CASCADA

1.9 Paradigma de desarrollo de Software

1.10 CONSTRUCCIÓN DE PROTOTIPOS

1.11 Ciclo de Vida de un Sistema basado en Prototipo.

1.12 MODELOS EVOLUTIVOS

UNIDAD II INGENIERÍA DE REQUISITOS

2.1 ANÁLISIS DE REQUERIMIENTOS

2.2 Tipos de requerimientos.

2.3 IDENTIFICACIÓN, ANÁLISIS, NEGOCIACIÓN

2.4 VALIDACIÓN Y GESTIÓN DE REQUISITOS

2.5 Validación de los requisitos de Software

2.6 MODELADO DEL ANÁLISIS, CASOS DE USO

2.7 CONCEPTOS BÁSICOS DE LA ORIENTACIÓN A OBJETOS

2.9 EL LENGUAJE DE MODELADO UNIFICADO (UML)

2.10 MODELADO ESTRUCTURAL

2.11 Grupo de Diseño de Clases en Paquetes

UNIDAD III MODELADO DE COMPORTAMIENTO

3.1 Diagramas

3.2. INTERACCIÓN (SECUENCIA Y COLABORACIÓN)

3.3 Diagramas de secuencia.

3.4 ESTADOS

3.5 ACTIVIDADES

3.6 MODELADO ARQUITECTÓNICO

3.7 LAS 4+1 VISTAS

3.8 CASOS DE USO

3.9 COMPONENTES

3.10. DESPLIEGUE

3.11 Diseño de alto nivel y diseño detallado

3.12 Diseño Estructurado

3.13 Diagrama Entidad-Relación (DER)

3.14 La descomposición (modularización) en el diseño

3.15 Los patrones de arquitectura y diseño

UNIDAD IV MODELO DE IMPLEMENTACIÓN

4.1 Modelos de implementación

4.2DIAGRAMAS DE COMPONENTES

4.3 ELEMENTOS DEL DIAGRAMA DE COMPONENTES

4.4 DIAGRAMAS DE DESPLIEGUE

4.5 MODELOS DE PRUEBA

4.6 IMPLEMENTACIÓN EN JAVA DE LOS DIAGRAMAS DE CLASE

4.7 Constructores y destructores declaración, uso y aplicaciones

4.8 REALIZACIÓN DE LOS CASOS DE USO Y DIAGRAMAS DE INTERACCIÓN

4.9 ARQUITECTURA LÓGICA CON PATRONES: SEPARACIÓN MODELO- VISTA.

.10 MVC Y BASES DE DATOS

4.11 USO EN APLICACIONES WEB

BIBLIOGRAFÍA

Índice

Unidad 1FUNDAMENTOS DE LA INGENIERÍA DEL SOFTWARE 11

1.1. DEFINICIÓN Y OBJETIVOS DE LA INGENIERÍA DEL SOFTWARE 11

1.2. CARACTERISITICAS Y APLICACIONES DEL SOFTWARE 12

1.3. EVOLUCIÓN HISTÓRICA DEL SOFTWARE 14

1.4 LEYES DE EVOLUCIÓN DEL SOFTWARE 15

1.5 PARADIGMAS DE SOFTWARE 16

1.6 PERSPECTIVA GENERAL DE LA INGENIERIA DEL SOFTWARE 17

1.7 PROCESOS, MÉTODOS Y HERRAMIENTAS 19

1.8 MODELO CLÁSICO O LINEAL, MODELO EN CASCADA 20

1.9 Paradigma de desarrollo de Software 23

1.10 CONSTRUCCIÓN DE PROTOTIPOS 26

1.11 Ciclo de Vida de un Sistema basado en Prototipo. 27

1.12 MODELOS EVOLUTIVOS 28

UNIDAD II INGENIERÍA DE REQUISITOS 34

2.1 ANÁLISIS DE REQUERIMIENTOS 34

2.2 Tipos de requerimientos. 35

2.3 IDENTIFICACIÓN, ANÁLISIS, NEGOCIACIÓN 37

2.4 VALIDACIÓN Y GESTIÓN DE REQUISITOS 40

2.5 Validación de los requisitos de Software 42

2.6 MODELADO DEL ANÁLISIS, CASOS DE USO 43

2.7 CONCEPTOS BÁSICOS DE LA ORIENTACIÓN A OBJETOS: 44

2.9 EL LENGUAJE DE MODELADO UNIFICADO (UML) 48

2.10 MODELADO ESTRUCTURAL 51

2.11 Grupo de Diseño de Clases en Paquetes 55

UNIDAD III MODELADO DE COMPORTAMIENTO 58

3.1 Diagramas 58

3.2. INTERACCIÓN (SECUENCIA Y COLABORACIÓN) 58

3.3 Diagramas de secuencia. 59

3.4 ESTADOS 60

3.5 ACTIVIDADES 61

3.6 MODELADO ARQUITECTÓNICO 62
3.7 LAS 4+1 VISTAS 63
3.8 CASOS DE USO 66
3.9 COMPONENTES 69
3.10. DESPLIEGUE 70
3.11 Diseño de alto nivel y diseño detallado 72
3.12 Diseño Estructurado 73
3.13 Diagrama Entidad-Relación (DER) 74
3.14 La descomposición (modularización) en el diseño 75
3.15 Los patrones de arquitectura y diseño 76
UNIDAD IV MODELO DE IMPLEMENTACIÓN 78
4.1 Modelos de implementación 78
4.2DIAGRAMAS DE COMPONENTES 78
4.3 ELEMENTOS DEL DIAGRAMA DE COMPONENTES 79
4.4 DIAGRAMAS DE DESPLIEGUE 80
4.5 MODELOS DE PRUEBA 82
4.6 IMPLEMENTACIÓN EN JAVA DE LOS DIAGRAMAS DE CLASE 84
4.7 Constructores y destructores declaración, uso y aplicaciones 85
4.8 REALIZACIÓN DE LOS CASOS DE USO Y DIAGRAMAS DE INTERACCIÓN 87
4.9 ARQUITECTURA LÓGICA CON PATRONES: SEPARACIÓN MODELO- VISTA. 88
.10 MVC Y BASES DE DATOS 90
4.11 USO EN APLICACIONES WEB 91
BIBLIOGRAFÍA BÁSICA Y COMPLEMENTARIA: 93

Unidad 1FUNDAMENTOS DE LA INGENIERÍA DEL SOFTWARE

1.1. DEFINICIÓN Y OBJETIVOS DE LA INGENIERÍA DEL SOFTWARE

Ingeniería de Software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo operación (funcionamiento) y mantenimiento del software: es decir, la aplicación de ingeniería al software.

El concepto de ingeniería del software surgió en 1968, tras una conferencia en Garmisch (Alemania) que tuvo como objetivo resolver los problemas de la crisis del software. Este fue ocasionado por el avance desmesurado del hardware lo que hacía el software cada vez más completo y generalmente nunca se terminaba a tiempo.

Es muy importante ya que con ella se puede analizar, diseñar, programar y aplicar un software de manera correcta y organizada, cumpliendo con todas las especificaciones del cliente y el usuario final. Lo anterior es posible gracias a los objetivos que esta propone.

En la construcción y desarrollo de proyectos se aplican métodos y técnicas para resolver los problemas, la informática aporta herramientas y procedimientos sobre los que se apoya la ingeniería de software.

- Mejorar la calidad de los productos de software
- Aumentar la productividad y el trabajo de los ingenieros de software.
- Utilidad
- Facilitar el control en el proceso de desarrollo de software
- Suministrar a los desarrolladores las bases para construir software de alta calidad en una forma eficiente.

Definir una disciplina que garantice la producción y el mantenimiento de los productos software desarrollados en el plazo fijado y dentro del costo estimado.

Software es mucho más que un código de programa. Un programa es un código ejecutable, usado para propósitos computacionales. El Software se considera una colección de códigos ejecutables

de programación, asociada a las bibliotecas y a la documentación. El Software, cuando se ha hecho para cubrir requisitos específicos se llama producto software.

Ingeniería Por otro lado, trata de desarrollar productos, utilizando métodos y principios científicos bien definidos.

DEFINICIONES

El IEEE (Instituto de Ingeniería Eléctrica y Electrónica) define la Ingeniería de software como: (1) La aplicación de una aproximación sistemática, disciplinada y cuantificable, al desarrollo, las operaciones y al mantenimiento del software; Esto es básicamente la aplicación de la Ingeniería al software.

(2) El estudio de la aproximación, tal y como se ha mencionado anteriormente. Fritz Bauer, un informático teórico alemán, define Ingeniería de software como:

La ingeniería de Software es el establecimiento y uso de los principios de la Ingeniería de sonido con tal de obtener software fiable y eficiante en máquinas reales de forma económica.

1.2. CARACTERISITICAS Y APLICACIONES DEL SOFTWARE

Un producto software puede ser juzgado según lo que ofrece y la manera en que se puede usar. El software debe satisfacer en los siguientes aspectos:

- Operacional
- Transicional
- Mantenimiento Operacional

Esto nos cuenta lo bien que funciona el software en operaciones. Se puede medir en base a:

- Presupuesto
- Usabilidad
- Eficiencia

- Exactitud
- Funcionalidad
- Dependabilidad
- Seguridad informática
- Seguridad

Transicional

Este aspecto es importante cuando el software se mueve de una plataforma a la otra:

- Portabilidad
- Interoperabilidad
- Reutilización
- Adaptabilidad
- Mantenimiento

Estos aspectos resumen la capacidad que tiene el software para mantenerse en entornos contantemente cambiantes:

- Modularidad
- Sostenibilidad
- Escalabilidad

En resumen, La Ingeniería de Software es una rama de las ciencias de la computación, que usa conceptos de Ingeniería bien definidos requeridos para producir productos software eficientes, duraderos, escalable, y accesibles a tiempo.

La necesidad de la Ingeniería de software viene de la alta tasa de cambio en los requisitos y en el entorno en que trabaja el software.

Software de gran tamaño: Es más fácil construir una pared que construir una casa, de la misma manera, a medida que el software aumenta su tamaño, la ingeniería debe entrar para darle un proceso científico.

Escalabilidad: Si el proceso software no estuviera basado en conceptos científicos y de ingeniería, sería más fácil volver a crear nuevo software que escalar uno ya existente.

Costes: A medida que la industria del hardware ha mostrado sus capacidades y grandes fabricaciones, ha bajado el precio del hardware electrónico e informático. Pero el coste del software sigue siendo alto si el proceso no se ha adaptado a los nuevos avances.

Naturaleza dinámica: La naturaleza del software, creciente y adaptable, depende en gran medida del entorno en el que el consumidor trabaje. Si la naturaleza del software siempre cambia, se necesitará mejorar el ya existente. Aquí es donde la ingeniería de software juega un gran papel.

Gestión de calidad: Los mejores procesos de desarrollo de software producen productos mejores y de calidad.

1.3. EVOLUCIÓN HISTÓRICA DEL SOFTWARE

El proceso de desarrollo de un producto software usando principios y métodos de Ingeniería de software, se denomina Evolución del Software. Esto incluye el desarrollo inicial del software, mantenimiento y actualizaciones, hasta que el producto deseado finalmente es desarrollado, lo que satisface los requerimientos esperados.

La evolución empieza con un proceso de recogida de requisitos. Luego los desarrolladores crean un prototipo inicial del software y se muestra a los consumidores para tener un feedback en una etapa temprana del desarrollo del producto de software. Los consumidores sugieren cambios, los cuales irán mejorando con actualizaciones y tareas de mantenimiento de manera progresiva. Este proceso cambia el software original hasta llegar al producto deseado.

Incluso después de que el consumidor tenga el software en sus manos, el avance de la tecnología y los cambios de requisitos fuerzan al producto software a cambiar en acorde a estos. Volver a cerrar software desde cero, e ir cumpliendo uno por uno los requisitos no son viable. La única solución viable y económica es actualizar el software ya existente para que se adecue satisfactoriamente con los requisitos más recientes.

1.4 LEYES DE EVOLUCIÓN DEL SOFTWARE

Lehman formuló leyes para la evolución del software. Dividió el software en 3 categorías distintas:

'S-type' ('static-type', tipo estático): Es un tipo de software, que funciona estrictamente según se ha definido especificaciones y soluciones. La solución y el método mediante el cual se consigue, se deben entender de inmediato antes de empezar a codificar. El software 's- type' está menos sujeto a cambios, de ahí que sea el más simple de todos. Por ejemplo, el programa de calculadora, para computación matemática.
'P-type' ('practical-type', tipo práctico): Este es un software con una colección de procedimientos. Esto se define exactamente por lo que pueden hacer los procedimientos. En este software, las especificaciones se pueden describir, pero la solución no es obvia al instante. Por ejemplo, software de juegos.
'E-type' ('embedded-type', tipo embebido o empotrado): Este software funciona estrechamente como requisito del entorno del mundo real. Este software tiene un alto grado de evolución ya que hay varios cambios en las leyes, impuestos, etc. en las situaciones del mundo real. Por ejemplo, el software de comercio en línea.

EVOLUCIÓN DEL SOFTWARE 'E-TYPE'

Lehman dictó 8 leyes de evolución del software 'E-Type':

Cambio continuo: Los sistemas de software 'E-type' deben adaptarse de forma progresiva a los cambios del mundo real, de no ser así se volverá progresivamente menos útil.

Complejidad creciente: A medida que el sistema software 'E-type' evoluciona, sus complejidades tienden a incrementar a menos que se trabaje en ello con el fin de mantenerlas o reducirlas.

Conservación de la familiaridad: La familiaridad con el software o con el conocimiento sobre cómo y por qué fue desarrollado de una manera en concreto, etc. debe ser retenido a cualquier coste, con tal de poder implementar cambios en el sistema.

Crecimiento continuado: Para que un sistema 'E-type' intente resolver problemas de negocios, su magnitud para implementar cambios crece en acorde con los cambios de estilo de vida del negocio.
Decremento de la calidad: Los sistemas software 'E-type' reducen su calidad a menos que se mantengan de forma rigurosa o se adapten a los cambios operativos del entorno.

Sistemas de retroalimentación: Los sistemas software 'E-type' son sistemas de retroalimentación multiloop y multinivel, deben ser considerados como tal con el fin de ser modificados o mejorados con éxito.

Autorregulación: Los procesos de evolución del sistema 'E-type', se regulan a sí mismos con la distribución del producto y las medidas del proceso de una manera casi normal.

Estabilidad organizacional: La tasa media de actividad efectiva global en un sistema evolutivo de 'E-type', no varía en toda la vida del producto.

1.5 PARADIGMAS DE SOFTWARE

Los paradigmas de Software son métodos y pasos, que se llevan a cabo mientras el software se diseña. Hay muchos métodos que se han propuesto y que funcionan hoy en día, pero necesitamos ver donde se ubican estos paradigmas en el marco de la Ingeniería de software. Estos se pueden combinar en varias categorías, en las que cada uno de ellos contiene a la otra:

El paradigma de programación es una parte del paradigma de diseño de Software y más adelante también se considera parte del paradigma de desarrollo de Software.

PARADIGMA DEL DESARROLLO SOFTWARE

Este paradigma es conocido como paradigma de ingeniería de software, en el que todos los conceptos de ingeniería pertenecientes al desarrollo de software son implementados. Incluye

varias investigaciones y recogida de requisitos lo que ayuda a la construcción del producto software. Consiste de:

- Recogida de requisitos
- Diseño de Software
- Programación

PARADIGMA DE DISEÑO DE SOFTWARE

Este paradigma forma parte del desarrollo software e incluye:

- Diseño
- Mantenimiento
- Programación

PARADIGMA DE PROGRAMACIÓN

Este paradigma se relaciona de estrechamente a aspectos de programación en el desarrollo de software. Esto incluye:

- Codificación
- Pruebas
- Integración

1.6 PERSPECTIVA GENERAL DE LA INGENIERIA DEL SOFTWARE

Inicialmente la programación de las computadoras era un arte que no disponía de métodos sistemáticos en los que poder basarse para la realización de productos software. Se realizaban sin ninguna planificación.

Posteriormente, desde mediados de los 60 hasta finales de los 70 se caracterizó por el establecimiento del software como un producto que se desarrollaba para una distribución general. En esta época nació lo que se conoce como el mantenimiento del software que se da cuando cambian los requisitos de los usuarios y se hace necesaria la modificación del software. El esfuerzo requerido para este mantenimiento era en la mayoría de los casos tan elevado que se hacía imposible su mantenimiento.

A continuación, surge una etapa que se caracteriza por la aparición de una serie de técnicas como la Programación Estructurada y las Metodologías de Diseño que solucionan los problemas anteriores. A finales de esta etapa aparecen las herramientas CASE, aunque como podemos imaginar eran muy rudimentarias.

Comentemos brevemente algunas de las técnicas que surgieron a partir de entonces:

LA PROGRAMACIÓN ESTRUCTURADA

Los fundamentos de lo que se dio en llamar programación estructurada se consolidaron a principio de los años 70 con el trabajo realizado por Dijkstra, que proponía una serie de construcciones lógicas para realizar cualquier programa; la secuencia, la condición y la repetición.

Los programas cada vez eran más grandes y complejos por lo que se necesitaba algo que facilitara la legibilidad del código, las pruebas y el mantenimiento. Mediante estas construcciones lógicas se logró limitar el diseño procedimental del software y se pudo comprobar a través de métricas de software que el uso de estas construcciones reducía la complejidad de los programas.

LAS METODOLOGÍAS DE DISEÑO

El empleo de técnicas para el diseño de programas, surge como corriente a mediados de los

70 dando lugar a las metodologías de diseño. Cada una de ellas introdujo heurísticas y notaciones propias, así como una visión que caracterizaba a la calidad del diseño. Sin embargo todas ellas introducían características comunes:

- Mecanismos de traducción de la representación de la información a una representación del diseño.
- Notación para representar los componentes funcionales e interface.
- Refinamiento y partición
- Criterios de valoración de la calidad.

Surgieron dos tipos de metodologías de diseño a destacar:

- Diseño funcional descendente. El sistema se observa en términos de las funciones que suministra.
- Diseño orientado a objetos. El sistema se observa como una sociedad de objetos, donde cada elemento del sistema (objeto), encapsula datos y operaciones.

1.7 PROCESOS, MÉTODOS Y HERRAMIENTAS

MÉTODO

Un método de ingeniería del software es un enfoque estructurado para el desarrollo de software cuyo propósito es facilitar la producción de software de alta calidad de una forma costeable. Métodos como Análisis Estructurado (DeMarco, 1978) y JSD (Jack- son, 1983) fueron los primeros desarrolladores en los años 70. Estos métodos intentaron identificar los componentes funcionales básicos de un sistema de tal forma que los métodos orientados a funciones aún se utilizan ampliamente. En los años 80 y 90, estos métodos orientados a funciones fueron complementados por métodos orientados a objetos, como los propuestos por Booh (1994) y Rumbaugh et al., 1991). Estos diferentes enfoques se han integrado a un solo enfoque unificado basado en UML (Lenguaje de Modelado Unificado).

Todos los métodos basados en la idea de modelos gráficos de desarrollo de un sistema y en el uso de estos modelos como un sistema de especificación o de diseño. Los métodos incluyen una variedad de componentes diferentes que son:

DESCRIPCIONES DEL MODELO DEL SISTEMA

Descripción de los modelos del Sistema que se desarrollará y la notación utilizada para definir los modelos.

- Reglas
- Restricciones que siempre aplican a los modelos de sistemas.
- Recomendaciones

Heurística que caracteriza una buena práctica de diseño en este método. Seguir estas recomendaciones debe dar como resultado un modelo del sistema bien organizado.

Guías en el proceso

Descripciones de las actividades que deben seguirse para desarrollar los modelos del sistema y la organización de estas actividades.

Técnicas

Técnicas de caja Negra de AQ maneja aplicaciones, o partes de ellas, que ya estan construidas. Estas técnicas verifican si el software cumplo o no con sus requerimientos.

Técnicas de caja blanca (o caja de vidrio) de QA se aplica a los componentes que forman la unidad que se está probando. Encender el televisor, cambiar los canales y luego observar su calidad equivaldría a la técnica de la caja negra. Probar el aparato de manera que funcione cada circuito es una técnica de caja blanca, por que involucra los componentes que conforman el televisor. Los procesos AQ que no son uno de estos dos extremos suelen llamarse técnicas de caja gris. Verificar la mayor parte de los componentes de su televisor es una técnica de caja gris. En el sentido estricto, la diferencia entre las técnicas de caja blanca y caja gris no siempre está bien definida.

Aunque muchas veces se piensa en las cajas negras y blancas en el contexto de las pruebas, estos conceptos se aplican a varias actividades de aseguramiento de la calidad. Las técnicas de caja blanca requieren que el ingeniero piense en la estructura, forma y proposito del artefacto que examina. Esto incluye usar métodos formales e inspección.

1.8 MODELO CLÁSICO O LINEAL, MODELO EN CASCADA

El ciclo de vida del desarrollo Software (SDLC en sus siglas inglesas), es una secuencia estructurada y bien definida de las etapas en Ingeniería de software para desarrollar el producto software deseado.

Actividades del SDLC

El SDLC aporta una serie de pasos a seguir con la finalidad de diseñar y desarrollar un producto software de manera eficiente. El borrador del SDLC incluye los pasos que veremos a continuación:

Comunicación

Este es el primer paso donde l usuario inicia la petición de un producto software determinado. Contacta al proveedor de servicios e intenta negociar las condiciones. Presenta su solicitud al proveedor de servicios aportando la organización por escrito.

Recolección de solicitudes
A partir de este paso y en adelante el equipo de desarrollo software trabaja para tirar adelante el proyecto. El equipo se reúne con varios depositarios de dominio del problema, e intentan conseguir la máxima cantidad de información posible sobre lo que requieren. Los requisitos se contemplan y agrupan en requisitos del usuario, requisitos funcionales y requisitos del sistema. La recolección de todos los requisitos se lleva a cabo como se especifica a continuación:

- Estudiando el software y el sistema actual o obsoleto,
- Entrevistando a usuarios y a desarrolladores de Software,
- Consultando la base de datos o
- Recogiendo respuestas a través de cuestionarios.

Estudio de viabilidad

Después de la recolección de requisitos, el equipo idea un plan para procesar el software. En esta fase, el equipo analiza si el software puede hacerse para cubrir todos los requisitos del usuario y si hay alguna posibilidad de que el software ya no sea necesario. Se investiga si el proyecto es viable a nivel financiero, práctico, y a nivel tecnológico para que la organización acepte la oferta. Hay varios algoritmos disponibles, los cuales ayudan a los desarrolladores a concluir si el proyecto software es factible o no.

Análisis del sistema

En este paso los desarrolladores trazan su plan e intentan crear el mejor y más conveniente modelo de software para el proyecto. El análisis del sistema incluye el entendimiento de las limitaciones del producto Software; el aprendizaje de los problemas relacionados con el sistema; los cambios que se requieren en sistemas ya existentes con antelación, identificando y dirigiendo el impacto del proyecto a la organización y al personal, etc. El equipo del proyecto analiza las posibilidades del proyecto y planifica la temporalización y los recursos correspondientes.
Diseño de Software

El siguiente paso es diseñar el producto software con la ayuda de toda la información recogida sobre requisitos y análisis. Los inputs (aportaciones) de los usuarios y los resultados de la recogida de información hecha en la fase anterior serán las aportaciones base de la fase actual. El output (o resultado) de esta etapa toma la forma de 2 diseños; El diseño lógico y el diseño físico. Los ingenieros crean meta-data (Metadatos), Diagramas dilógicos, diagramas de flujo de datos, y en algunos casos pseudocódigos.

Codificación

Esta fase también se puede denominar 'fase de programación'. La implementación del diseño de software empieza con el lenguaje de programación más conveniente, y desarrollando programas ejecutables y sin errores de manera eficiente.

Pruebas

Se estima que el 50% de todos los procesos de desarrollo de software deberían ser evaluados. Los errores pueden arruinar el software tanto a nivel crítico y hasta el punto de ser eliminado. Las pruebas de Software se hacen mientras se codifica y suelen hacerlo los desarrolladores y otros expertos evaluadores a varios niveles. Esto incluye evaluación de módulos, evaluación del programa, evaluación del producto, evaluación interna y finalmente evaluación con el consumidor final. Encontrar errores y su remedio a tiempo es la llave para conseguir un software fiable.

Integración

El Software puede necesitar estar integrado con las bibliotecas, Bases de datos o con otro u otros programas. Esta fase del SDLC se focaliza en la integración del software con las entidades del mundo exterior.

Implementación

Aquí se instala el software en máquinas de clientes. A veces, el software necesita instalar configuraciones para el consumidor final con posterioridad. El Software se evalúa por su adaptabilidad y su portabilidad, en cuanto a las cuestiones relacionadas con la integración y conceptos asociados, se resuelven durante la implementación.

Mantenimiento y Funcionamiento

Esta fase confirma el funcionamiento del software en términos de más eficiencia y menos errores. Si se requiere, los usuarios se forman, o se les presta documentación sobre como operar y como mantenerlo en funcionamiento. El software se mantiene de forma temprana actualizando el código en acorde a los cambios que tienen lugar en entornos del usuario o tecnológicos. Esta fase puede que tenga que encarar retos originados por virus ocultos o problemas no identificados del mundo real.

Disposición

Con el paso del tiempo, puede que el software falle en su ejecución. Puede que se vuelva totalmente obsoleto o que necesite actualizaciones. De ahí surge una necesidad urgente de eliminar una parte importante del sistema. Esta fase incluye archivar datos y componentes software requerido, cierre del sistema, planificación de la actividad de disposición y terminación de sistema en el momento final del sistema.

1.9 Paradigma de desarrollo de Software

El Paradigma de desarrollo de Software ayuda al desarrollador a escoger una estrategia para desarrollar el software. El paradigma de desarrollo software tiene su propio set de herramientas, métodos y procedimientos, los cuales son expresados de forma clara, y define el ciclo de vida del

desarrollo del software. Algunos paradigmas de desarrollo de software o modelos de proceso se definen a continuación:

Modelo de cascada

El modelo de cascada es el modelo de paradigma más simple en desarrollo de software. Sigue un modelo en que las fases del SDLC funcionarán una detrás de la otra de forma lineal. Lo que significa que solamente cuando la primera fase se termina se puede empezar con la segunda, y así progresivamente.

Este modelo asume que todo se lleva a cabo y tiene lugar tal y como se había planeado en la fase anterior, y no es necesario pensar en asuntos pasados que podrían surgir en la siguiente fase. Este modelo no funcionará correctamente si se dejan asuntos de lado en la fase previa. La naturaleza secuencial del modelo no permite volver atrás y deshacer o volver a hacer acciones.

Este modelo es recomendable cuando el desarrollador ya ha diseñado y desarrollado softwares similares con anterioridad, y por eso está al tanto de todos sus dominios.

Modelo repetitivo

Este modelo guía el proceso de desarrollo de software en repeticiones. Proyecta el proceso de desarrollo de forma cíclica repitiendo cada paso después de cada ciclo en el proceso de SDLC.

El software primero se desarrolla en menor escala y se siguen y tienen en consideración todos los pasos. Entonces, por cada repetición, más módulos y características son diseñados, codificados, evaluados y añadidos al software. Cada ciclo produce un software completo, con más características y capacidad que los previos.

Después de cada repetición, el equipo directivo puede concentrarse en la gestión de riesgos y prepararse para la siguiente repetición. Como el ciclo incluye pequeñas porciones de la totalidad del proceso software, es más fácil gestionar el proceso de desarrollo, pero a la vez se consumen más recursos.

Modelo en espiral

El modelo en espiral es una combinación de ambos modelos, el repetitivo y uno del modelo SDLC. Se puede ver como si se combina un modelo de SDLC combinado con un proceso cíclico (modelo repetitivo).

Este modelo considera el riesgo, factor que otros modelos olvidan o no prestan atención en el proceso. El modelo empieza determinando los objetivos y las limitaciones del software al inicio de cada repetición. En la siguiente etapa se crean los modelos de prototipo del software. Esto incluye el análisis de riesgos. Luego un modelo estándar de SDLC se usa para construir el software. En la cuarta etapa es donde se prepara el plan de la siguiente repetición.

Modelo V

El mayor inconveniente del modelo de cascada es que solo se pasa a la siguiente fase cuando se completa la anterior, por tanto, no es posible volver atrás si se encuentra algún error en las etapas posteriores. El Modelo V aporta opciones de evaluación del software en cada etapa de manera inversa.

En cada etapa, se crea la planificación de las pruebas y los casos de pruebas para verificar y validar el producto según los requisitos de la etapa. Por ejemplo, en la etapa de recogida de requisitos, el equipo de evaluadores prepara las pruebas de caso correspondientes a los requisitos. Más tarde, cuando el producto se desarrolla y está preparado para ser evaluado, las pruebas de caso en esta etapa verifican el software y su validez según sus requisitos.

Esto hace que tanto la verificación como la validación vayan en paralelo. Este modelo también se conoce como modelo de validación y verificación.

Modelo Big Bang

Este modelo es el modelo con la forma más simple. Requiere poca planificación, mucha programación y también muchos fondos. Este modelo se conceptualiza alrededor de la teoría

de creación del universo 'Big Bang'. Tal como cuentan los científicos, después del big bang muchas galaxias, planetas y estrellas evolucionaron. De la misma manera, si reunimos muchos fondos y programación, quizá podemos conseguir el mejor producto de software.

Para este modelo, se requiere poca planificación. No sigue ningún proceso concreto, y a veces el cliente no está seguro de las futuras necesidades y requisitos. Por tanto, la entrada o input respecto a los requisitos es arbitraria.

Este modelo no es recomendable para grandes proyectos de software, pero es bueno para aprender y experimentar.

1.10 CONSTRUCCIÓN DE PROTOTIPOS

El modelo de prototipos permite que todo el sistema, o algunos de sus partes, se construyan rápidamente para comprender con facilidad y aclarar ciertos aspectos en los que se aseguren que el desarrollador, el usuario, el cliente estén de acuerdo en lo que se necesita así como también la solución que se propone para dicha necesidad y de esta forma minimizar el riesgo y la incertidumbre en el desarrollo, este modelo se encarga del desarrollo de diseños para que estos sean analizados y prescindir de ellos a medida que se adhieran nuevas especificaciones, es ideal para medir el alcance del producto, pero no se asegura su uso real.

Este modelo principalmente se lo aplica cuando un cliente define un conjunto de objetivos generales para el software a desarrollarse sin delimitar detalladamente los requisitos de entrada procesamiento y salida, es decir cuando el responsable no está seguro de la eficacia de un algoritmo, de la adaptabilidad del sistema o de la forma en que interactúa el hombre y la máquina. Este modelo se encarga principalmente de ayudar al ingeniero de sistemas y al cliente a entender de mejor manera cuál será el resultado de la construcción cuando los requisitos estén satisfechos.

El paradigma de construcción de prototipos tiene tres pasos:

- Escuchar al cliente. Recolección de requisitos. Se encuentran y definen los objetivos globales, se identifican los requisitos conocidos y las áreas donde es obligatorio más definición.
- Construir y revisar la maqueta (prototipo).

- El cliente prueba la maqueta (prototipo) y lo utiliza para refinar los requisitos del software.

Este modelo es útil cuando:

- El cliente no identifica los requisitos detallados.
- El responsable del desarrollo no está seguro de la eficiencia de un algoritmo, sistema operativo o de la interface hombre-máquina.

1.11 Ciclo de Vida de un Sistema basado en Prototipo.

Una maqueta o prototipo de pantallas muestra la interfaz de la aplicación, su cara externa, pero dicha interfaz está fija, estática, no procesa datos. El prototipo no tiene desarrollada una lógica interna, sólo muestra las pantallas por las que irá pasando la futura aplicación.

Por su parte, el prototipo funcional evolutivo desarrolla un comportamiento que satisface los requisitos y necesidades que se han entendido claramente. Realiza, por tanto, un un proceso real de datos, para contrastarlo con el usuario. Se va modificando y desarrollando sobre la marcha, según las apreciaciones del cliente. Esto ralentiza el proceso de desarrollo y disminuye la fiabilidad, puesto que el software está constantemente variando, pero, a la larga, genera un producto más seguro, en cuanto a la satisfacción de las necesidades del cliente.

Cuando un prototipo se desarrolla con el sólo propósito de precisar mejor las necesidades del cliente y después no se va a aprovechar ni total ni parcialmente en la implementación del sistema final se habla de un prototipo desechable.

Para que la construcción de prototipos sea posible se debe contar con la participación activa del cliente.

Ciclo de vida del prototipo
Ventajas del Modelo de Prototipo.

Este modelo es útil cuando el cliente conoce los objetivos generales para el software, pero no identifica los requisitos detallados de entrada, procesamiento o salida. También ofrece un mejor enfoque cuando el responsable del desarrollo del software está inseguro de la eficacia de un

algoritmo, de la adaptabilidad de un sistema operativo o de la forma que debería tomar la interacción humano-máquina.

Desventajas del Modelo de Prototipo.

Su principal desventaja es que una vez que el cliente ha dado su aprobación final al prototipo y cree que está a punto de recibir el proyecto final, se encuentra con que es necesario reescribir buena parte del prototipo para hacerlo funcional, porque lo más seguro es que el desarrollador haya hecho compromisos de implementación para hacer que el prototipo funcione rápidamente. Es posible que el prototipo sea muy lento, muy grande, no muy amigable en su uso, o incluso, que esté escrito en un lenguaje de programación inadecuado.

El cliente ve funcionando lo que para él es la primera versión del prototipo que ha sido construido con "plastilina y alambres", y puede desilusionarse al decirle que el sistema aún no ha sido construido. El desarrollador puede ampliar el prototipo para construir el sistema final sin tener en cuenta los compromisos de calidad y de mantenimiento que tiene con el cliente.

1.12 MODELOS EVOLUTIVOS

Los modelos evolutivos son iterativos. Se caracterizan por la forma en que permiten a los ingenieros del software desarrollar versiones cada vez más completas del software.

El software evoluciona con el tiempo. Los requisitos del usuario y del producto suelen cambiar conforme se desarrolla el mismo. Las fechas de mercado y la competencia hacen que no sea posible esperar a poner en el mercado un producto absolutamente completo, por lo que se aconsejable introducir una versión funcional limitada de alguna forma para aliviar las presiones competitivas.

MODELO ESPIRAL

Es un ciclo de vida de software definido por Barry Boehm en 1988, utilizado mayormente en la ingeniería de software. Fue descrito por Boehm de la siguiente manera: ¨El modelo de desarrollo en espiral es un generador de modelo de proceso guiado por el riesgo que se emplea para conducir sistemas intensivos de ingeniería de software concurrente y a la vez con muchos

usuarios¨. Las actividades que conforman este modelo forman una espiral, en la que cada bucle o interacciona representa un conjunto de actividades. Se tiene en cuenta fuertemente el riesgo que aparece a la hora de desarrollar software.

- Planificación: determinación de objetivos, alternativas y restricciones.
- Análisis de riesgo: análisis de alternativas e identificación/resolución de riesgos.
- Ingeniería: desarrollo del producto del "siguiente nivel",

Durante la primera vuelta alrededor de la espiral se definen los objetivos, las alternativas y las restricciones, y se analizan e identifican los riesgos. Si el análisis de riesgo indica que hay una incertidumbre en los requisitos, se puede usar la creación de prototipos en el cuadrante de ingeniería para dar asistencia tanto al encargado de desarrollo como al cliente.

El cliente evalúa el trabajo de ingeniería (cuadrante de evaluación de cliente) y sugiere modificaciones. Sobre la base de los comentarios del cliente se produce la siguiente fase de planificación y de análisis de riesgo. En cada bucle alrededor de la espiral, la culminación del análisis de riesgo resulta en una decisión de "seguir o no seguir".

Con cada iteración alrededor de la espiral (comenzando en el centro y siguiendo hacia el exterior), se construyen sucesivas versiones del software, cada vez más completa y, al final, al propio sistema operacional.

El paradigma del modelo en espiral para la ingeniería de software es actualmente el enfoque más realista para el desarrollo de software y de sistemas a gran escala. Utiliza un enfoque evolutivo para la ingeniería de software, permitiendo al desarrollador y al cliente entender y reaccionar a los riesgos en cada nivel evolutivo. Utiliza la creación de prototipos como un mecanismo de reducción de riesgo, pero, lo que es más importante permite a quien lo desarrolla aplicar el enfoque de creación de prototipos en cualquier etapa de la evolución de prototipos.

Características:

Tiene y está conformado en un enfoque cíclico para el crecimiento del grado de definición e implementación de un sistema, mientras que disminuye su grado de riesgo.

Utiliza un conjunto de puntos de fijación para asegurar el compromiso que asume el usuario con las soluciones de sistema que sean factibles y totalmente satisfactorias.

El análisis del riesgo se hace de forma explícita y clara. Une los mejores elementos de los restantes modelos.

1. Reduce riesgos del proyecto
2. Incorpora objetivos de calidad
3. Integra el desarrollo con el mantenimiento, etc.

Además, es posible tener en cuenta mejoras y nuevos requerimientos sin romper con la metodología, ya que este ciclo de vida no es rígido ni estático.

Desventajas:

1. Genera mucho tiempo en el desarrollo del sistema
2. Modelo costoso
3. Requiere experiencia en la identificación de riesgos

Básicamente consiste en una serie de ciclos que comienzan desde el centro que se repiten en forma de espiral, El Espiral puede verse como un modelo evolutivo que conjuga la naturaleza iterativa del modelo MCP con los aspectos controlados y sistemáticos del Modelo Cascada, este modelo no es tan usado por lo que no se tiene clara la medida de eficiencia de este modelo en un sistema de información, pero este modelo podemos ver que apto para el desarrollo de sistemas operativos complejos, este modelo evolutivo tiene aspectos buenos ya que todo lo que hace este modelo es controlar y sistematizar las actividades.

CONCURRENTE

La gran mayoría de los modelos de desarrollo de software están ligados al tiempo, cuanto más se dure sería mejor o peor para el modelo. Este modelo de desarrollo de software concurrente está ligado y dirigido primordialmente por las necesidades del usuario, las decisiones que se tomen acabo y las tareas que realicen dichos usuarios.

También define una serie de acontecimientos que se disparan a los estados de cada uno de las actividades realizadas.

Fue creado por Davis Sitaram, se puede representar en forma de esquema de una serie de actividades, técnicas tareas y estados asociados a ellas como ya lo habíamos mencionado. Tiene la capacidad de describir las múltiples actividades del software que están ocurriendo simultáneamente.

Características:

- Se expresa de manera esquematizada y organizada.
- Cada actividad lleva procesos concurrentes.
- Se aplica a la mayoría de tipos de desarrollo de software
- Es un módulo aplicable para el cliente soñador.
- Está dirigido básicamente y esencialmente a las necesidades del usuario.
- Es aplicable al cliente servidor.

Ventajas:

- Es excelente para proyectos en los que se conforman grupos de trabajo.
- Proporciona una imagen exacta del estado actual de un proyecto
- No restringe el proyecto a una secuencia de sucesos.

Desventajas:

- Si no se dan las condiciones específicas no se puede aplicar.
- Si no existe grupo de trabajo no se puede trabajar en este método.
- Todas las actividades de red existen simultáneamente con otras.
- Los sucesos generados dentro de una actividad, o en algún otro lado de la red de actividad, inician las transiciones entre los estados de otra actividad.

El modelo de proceso concurrente se puede sentar en forma de esquema como una serie de actividades técnicas importantes, tareas y estados asociados a ellas. Por ejemplo, la actividad de ingeniería definida para el modelo en espiral, se lleva acabo invocando las tareas siguientes: modelado de construcción de prototipos y/o análisis, especificación de requisitos y diseño.

Este modelo define una serie de acontecimientos que disparan transiciones de estado a estado para cada una de las actividades del proyecto, utilizando para ellos el paradigma de desarrollo de aplicaciones cliente/servidor.

INCREMENTAL

Es un modelo basado en varios ciclos en Cascada retroalimentados aplicados consecutivamente. Combina los elementos del MLS con la filosofía con la filosofía interactiva de construcción de prototipos.

Fue propuesto por Mills en 1980. En una visión genérica, el proceso se divide en 4 partes: Análisis, Diseño, Código y Prueba. Se usa el principio de trabajo en cadena o "Pipeline", utilizado en muchas otras formas de programación.

Características:

- Se evitan proyectos largos y se entrega "algo de valor" a los usuarios con cierta frecuencia.
- El usuario se involucre más.
- Difícil de evaluar el costo total.
- Difícil de aplicar a los sistemas transaccionales que tienden a ser integrados y a operar como un todo.
- Requiere gestores experimentados.
- Los errores en los requisitos se detectan tarde.
- El resultado puede ser muy positivo.

Ventajas:

- Con un paradigma incremental se reduce el tiempo de desarrollo inicial, ya que se implementa la funcionalidad parcial.
- También provee un impacto ventajoso frente al cliente, que es la entrega temprana de partes operativas del Software.
- El modelo proporciona todas las ventajas del modelo en cascada realimentado, reduciendo sus desventajas sólo al ámbito de cada incremento.
- Permite entregar al cliente un producto más rápido en comparación del modelo de cascada.

- Resulta más sencillo acomodar cambios al acotar el tamaño de los incrementos.
- Por su versatilidad requiere de una planeación cuidadosa tanto a nivel administrativo como técnico.

Desventajas:

- El modelo Incremental no es recomendable para casos de sistemas de tiempo real, de alto nivel de seguridad, de procesamiento distribuido, y/o de alto índice de riesgos. Requiere de mucha planeación, tanto administrativa como técnica.
- Requiere de metas claras para conocer el estado de los proyectos.

El modelo incremental es una unión de las mejores funcionalidades del modelo de cascada y del modelo de prototipos. A medida que se presenta un prototipo se produce un "incremento", que es una iteración del proceso anterior, pero aplicando las experiencias aprendidas del proceso anterior. A diferencia del modelo de prototipos, los prototipos de este modelo están orientados a ser operacionales en cada incremento y no ser solo una "previa" de cómo sería el sistema en su versión final.

UNIDAD II INGENIERÍA DE REQUISITOS

2.1 ANÁLISIS DE REQUERIMIENTOS

Los requerimientos permiten que los desarrolladores expliquen cómo han entendido lo que el cliente pretende del sistema. También, indican a los diseñadores qué funcionalidad y qué características va a tener el sistema resultante. Y, además, indican al equipo de pruebas qué demostraciones llevar a cabo para convencer al cliente de que el sistema que se le entrega es lo que solicitó. Las características de los requerimientos mencionados en el estándar IEEE830 los explica [Pfleeger, 2002] como sigue:

• Deben ser correctos: Tanto el cliente como el desarrollador deben revisarlos para asegurar que no tienen errores.

• Deben ser consistentes: Dos requerimientos son inconsistentes cuando es imposible satisfacerlos simultáneamente.

• Deben estar completos: El conjunto de requerimientos está completo si todos los estados posibles, cambios de estado, entradas, productos y restricciones están descritos en alguno de los requerimientos.

• Deben ser realistas: Todos los requerimientos deben ser revisados para asegurar que son posibles.

• ¿Cada requerimiento describe algo que es necesario para el cliente?: Los requerimientos deben ser revisados para conservar sólo aquellos que inciden directamente en la resolución del problema del cliente.

• Deben ser verificables: Se deben poder preparar pruebas que demuestren que se han cumplido los requerimientos.

• Deben ser rastreables: ¿Se puede rastrear cada función del sistema hasta el conjunto de requerimientos que la establece?

Requerimientos: Los requerimientos especifican qué es lo que el sistema debe hacer (sus funciones) y sus propiedades esenciales y deseables. La captura de los requerimientos tiene como objetivo principal la comprensión de lo que los clientes y los usuarios esperan que haga el sistema. Un requerimiento expresa el propósito del sistema sin considerar como se va a implantar. En otras palabras, los requerimientos identifican el qué del sistema, mientras que el diseño establece el cómo del sistema.

La captura y el análisis de los requerimientos del sistema es una de las fases más importantes para que el proyecto tenga éxito. Como regla de modo empírico, el costo de reparar un error se incrementa en un factor de diez de una fase de desarrollo a la siguiente, por lo tanto la preparación de una especificación adecuada de requerimientos reduce los costos y el riesgo general asociado con el desarrollo [Norris & Rigby, 1994].

Análisis de requerimientos: Es el conjunto de técnicas y procedimientos que nos permiten conocer los elementos necesarios para definir un proyecto de software. Es una tarea de ingeniería del software que permite especificar las características operacionales del software, indicar la interfaz del software con otros elementos del sistema y establecer las restricciones que debe cumplir el software.

2.2 Tipos de requerimientos.

Según el estándar internacional de Especificación de Requerimientos IEEE830, los documentos de definición y especificación de requerimientos deben contemplar los siguientes aspectos resumidos por [Pfleeger, 2002] como se indica a continuación:

Ambiente físico

- ¿Dónde está el equipo que el sistema necesita para funcionar?
- ¿Existe una localización o varias?
- ¿Hay restricciones ambientales como temperatura, humedad o interferencia magnética?

Interfaces

- ¿La entrada proviene de uno o más sistemas?
- ¿La salida va a uno o más sistemas?
- ¿Existe una manera preestablecida en que deben formatearse los datos?

Usuarios y factores humanos

- ¿Quién usará el sistema?
- ¿Habrá varios tipos de usuario?

- ¿Cuál es el nivel de habilidad de cada tipo de usuario?
- ¿Qué clase de entrenamiento requerirá cada tipo de usuario?
- ¿Cuán difícil le resultará al usuario hacer uso indebido del sistema?

Funcionalidad

- ¿Qué hará el sistema?
- ¿Cuándo lo hará?
- ¿Existen varios modos de operación?
- ¿Cómo y cuándo puede cambiarse o mejorarse un sistema?
- ¿Existen restricciones de la velocidad de ejecución, tiempo de respuesta o rendimiento?

Documentación

- ¿Cuánta documentación se requiere?
- ¿Debe estar en línea, en papel o en ambos?
- ¿A que audiencia está orientado cada tipo de información?

Datos

- ¿Cuál será el formato de los datos, tanto para la entrada como para la salida?
- ¿Cuán a menudo serán recibidos o enviados?
- ¿Cuán exactos deben ser?
- ¿Cuántos datos fluyen a través del sistema?
- ¿Debe retenerse algún dato por algún período de tiempo?

Recursos

- ¿Qué recursos materiales, personales o de otro tipo se requieren para construir, utilizar y mantener el sistema?
- ¿Qué habilidades deben tener los desarrolladores?
- ¿Cuánto espacio físico será ocupado por el sistema?
- ¿Cuáles son los requerimientos de energía, calefacción o acondicionamiento de aire?
- ¿Existe un cronograma prescrito para el desarrollo?
- ¿Existe un límite sobre la cantidad de dinero a gastar en el desarrollo o en hardware y software?

Seguridad

- ¿Debe controlarse el acceso al sistema o a la información?
- ¿Cómo se podrán aislar los datos de un usuario de los de otros?
- ¿Cómo podrán aislarse los programas de usuario de los otros programas y del sistema operativo?
- ¿Con qué frecuencia deben hacerse copias de respaldo?
- ¿Las copias de respaldo deben almacenarse en un lugar diferente?
- ¿Deben tomarse precauciones contra el fuego, el daño provocado por agua o el robo?
- ¿Cuáles son los requerimientos para la confiabilidad, disponibilidad, facilidad de mantenimiento, seguridad y demás atributos de calidad?
- ¿Cómo deben demostrarse las características del sistema a terceros?
- ¿El sistema debe detectar y aislar defectos?
- ¿Cuál es el promedio de tiempo prescrito entre fallas?
- ¿Existe un tiempo máximo permitido para la recuperación del sistema después de una falla?
- ¿El mantenimiento corregirá los errores, o incluirá también el mejoramiento del sistema?
- ¿Qué medidas de eficiencia se aplicarán al uso de recursos y al tiempo de respuesta?
- ¿Cuán fácil debe ser mover el sistema de una ubicación a otra o de un tipo de computadora a otro?

2.3 IDENTIFICACIÓN, ANÁLISIS, NEGOCIACIÓN

Métodos generales de entrevistas.

La entrevista es una forma de recoger información de otra persona a través de una comunicación interpersonal que se lleva a cabo por medio de una conversación estructurada, [Braude, 2003] distingue las siguientes fases:

- Preparación: El entrevistador debe documentarse e investigar la situación de la organización, analizando los documentos de la empresa disponible. Hay que intentar minimizar el número de entrevistados, hay que considerar las entrevistas de cortesía, analizar el perfil de los entrevistados, definir el objetivo y el contenido de la entrevista, planificar el lugar y la hora en la que se va a desarrollar la entrevista es conveniente realizarla en un lugar confortable.

Algunos proponen enviar previamente el entrevistado un cuestionario y un pequeño documento de introducción al proyecto de desarrollo.

- Realización: Hay tres fases:

1. Apertura: Presentarse e informar al entrevistado sobre la razón de la entrevista.
2. Desarrollo: Cumplir las reglas del protocolo, hay que llegar a un acuerdo sobre cómo se va a registrar la información obtenida.
3. Terminación: Se termina recapitulando la entrevista agradeciendo el esfuerzo y dejando abierta la posibilidad de volver a contactar para aclarar conceptos o bien citándole para otra entrevista.

- Análisis: Consiste en leer las notas, pasarlas en limpio, reorganizar la información, contrastarlas con otras entrevistas o fuentes de información, evaluar cómo ha ido la entrevista.

Las entrevistas con los involucrados con el sistema son parte de la mayoría de los procesos de la ingeniería de requerimientos. En estas entrevistas, el equipo de la ingeniería de requerimientos hace preguntas sobre el sistema que utilizan y sobre el sistema a desarrollar.

Los requerimientos provienen de las respuestas a estas preguntas. Las entrevistas pueden ser de dos tipos:

1. Entrevistas cerradas: donde los entrevistados responden a un conjunto predefinido de preguntas.

2. Entrevistas abiertas: donde no hay un programa predefinido. El equipo de la ingeniería de requerimientos examina una serie de cuestiones con los involucrados con el sistema y, por lo tanto, desarrolla una mejor comprensión de sus necesidades.

Las entrevistas sirven para obtener una comprensión general de lo que hacen los futuros usuarios del sistema, cómo podrían interactuar con el sistema y las dificultades a las que se enfrentan con los sistemas actuales. A la gente le gusta hablar de su trabajo y normalmente se alegran de verse implicados en las entrevistas. Sin embargo, no son de tanta utilidad para la comprensión de los requerimientos del dominio de la aplicación, tampoco son una técnica eficaz para obtener conocimiento sobre los requerimientos y restricciones organizacionales debido a que existen sutiles poderes e influencias entre los involucrados en el sistema.

En general, la mayoría de la gente es reacia a discutir cuestiones políticas y organizacionales que pueden influir en los requerimientos. Por otra parte, hay que destacar que, para la mayoría de las personas, la entrevista es un compromiso adicional sobre su cargada lista de trabajos pendientes. Algunos autores proponen mandar previamente un cuestionario que debe llenar el entrevistado y un pequeño documento de introducción al proyecto de desarrollo. El cuestionario permite que el entrevistado conozca los temas que se van a tratar y pueda conseguir con anticipación información que no tenga a disposición inmediata.

Los buenos entrevistadores poseen dos características importantes:

1. No tienen prejuicios, evitan ideas preconcebidas sobre los requerimientos y están dispuestos a escuchar a los entrevistados. Si el entrevistado propone requerimientos sorprendentes, están dispuestos a cambiar su opinión del sistema.
2. Ayudan al entrevistado a empezar las discusiones con una pregunta, una propuesta de requerimientos o sugiriendo trabajar juntos en un prototipo del sistema.

Preguntar al cliente por lo que quiere de manera general normalmente no proporciona información útil. Para la mayoría de la gente es mucho más fácil hablar de algo en particular que en términos generales.

Una manera de manejar las entrevistas:

Antes de la entrevista.

1. Enumerar y dar prioridad a los clientes que se entrevistarán.
2. Programar una entrevista con tiempos de inicio y terminación fijos.

En la entrevista

1. No ser pasivo, investigar y animar, persistir en entender deseos y explorar necesidades.

Examinar casos de uso, flujos de datos y/o diagramas de estado.

3. Tomar notas exhaustivas.
4. Programar una reunión de seguimiento.

Después de la entrevista.

1. Bosquejar la especificación de los requerimientos.

Enviar correos electrónicos a los clientes para obtener sus comentarios.

El análisis y especificación de requerimientos puede parecer una tarea relativamente sencilla, pero las apariencias engañan. Puesto que el contenido de comunicación es muy alto, abundan los cambios por mala interpretación o falta de información. El dilema con el que se enfrenta un ingeniero de software puede ser comprendido repitiendo la sentencia de un cliente anónimo: "Sé que crees que comprendes lo que piensas que he dicho, pero no estoy seguro de que entendiste lo que yo quise decir". En la tabla 1.1 [Pfleeger, 2002] ilustra el conflicto que encontró (Scharer, 1990) cuando los desarrolladores y los usuarios se limitan a ver el problema desde su particular punto de vista sin tomar en cuenta la situación del otro.

2.4 VALIDACIÓN Y GESTIÓN DE REQUISITOS

Es un proceso que consta de cuatro pasos:

1. Estudio de viabilidad
2. Recogida de requisitos
3. Requisitos del Software
4. Validación de los requisitos de Software

Estudio de viabilidad

Cuando el cliente se acerca a la organización para obtener el producto deseado desarrollado, expone una idea aproximada de las funciones que el software debe cumplir y qué características se esperan del software.

Refiriéndose a esta información, los analistas elaboran un estudio detallado sobre la viabilidad del sistema deseado y de sus funcionalidades, para proceder a desarrollarlo.

Este estudio de viabilidad se centra en el objetivo de la organización. El estudio analiza la materialización práctica del producto software respecto a su implementación, la contribución de proyecto a la organización, los límites de costes, y según los objetivos y valores de la organización. Explora aspectos técnicos del proyecto y del producto, como la utilidad, el mantenimiento, la productividad y la capacidad de integración.

El resultado o output de esta fase debe ser un informe del estudio de viabilidad, conteniendo comentarios adecuados y recomendaciones para la gestión sobre si se debe tirar adelante o no el proyecto.

Recogida de requisitos

Si el informe de viabilidad es positivo en relación a tomar el proyecto, la siguiente fase empieza con la recolección de requisitos por parte del consumidor. Analistas e Ingenieros se comunican con el cliente y los consumidores para conocer sus ideas sobre qué debe aportar el software y qué características quieren que incluya éste.

Requisitos del Software

El SRS es un documento creado por los analistas de sistema después de recoger los requisitos.

El SRS define cómo va a interactuar el software que quiere crearse con el hardware, las interfaces externas, la velocidad operativa, el tiempo de respuesta del sistema, la portabilidad del software en las diversas plataformas, el mantenimiento, la velocidad de reponerse después de estropearse, su seguridad, calidad, limitaciones, etc.

Los requisitos recibidos por parte del cliente se escriben en lenguaje natural. Es responsabilidad del analista de sistemas documentar sobre los requisitos en lenguaje tecnológico para que puedan ser útiles y comprendidos por el equipo de desarrollo de software.

El SRS debe venir con las siguientes características:

- Los requisitos del usuario se deben expresar en lenguaje natural.

- Los requisitos técnicos se deben expresar en lenguaje estructurado, el cual se usará dentro de la organización.
- La descripción del diseño se debe escribir en pseudocódigo.
- El formato de Forms y GUI impresiones de pantalla.
- Anotaciones condicionales y matemáticas para DFDs etc.

2.5 Validación de los requisitos de Software

Después del desarrollo de los requisitos, los que se mencionen en este documento serán validados. El usuario puede que pida soluciones ilegales y poco prácticas, y los expertos puede que interpreten los requisitos de forma incorrecta. Estos resultados se incrementan en coste si no se cortan de raíz. Los requisitos se pueden evaluar en contraste con las siguientes condiciones:

- Si pueden ser implementados de manera práctica.
- Si son válidos a nivel de funcionalidad y dominio del software
- Si hay alguna ambigüedad
- Si se han completado
- Si se pueden demostrar

Proceso de inducción de requisitos

El Proceso de inducción de requisitos se puede representar usando el siguiente diagrama:

- Recogida de requisitos - Los desarrolladores hablan con el cliente y los consumidores finales para conocer sus expectativas respecto al software.
- Organizar requisitos - Los desarrolladores priorizan y organizan los requisitos en orden de importancia, urgencia, y conveniencia.
- Negociación y debate - Si los requisitos son ambiguos o hay algún conflicto en los requisitos de varios accionistas, hay una negociación y un debate con ellos. Los requisitos entonces, se priorizan y se acuerdan de manera razonable.
- Los requisitos vienen de varios accionistas. Para eliminar cualquier ambigüedad o conflicto, se debate para encontrar claridad y corrección. Los requisitos surrealistas se acuerdan de forma razonable.

- Documentación - Los requisitos formales y no formales, funcionales y no funcionales, son documentados y se ponen disponibles para procesar en la siguiente fase.

2.6 MODELADO DEL ANÁLISIS, CASOS DE USO

Diagrama de Casos de Uso

Un caso de uso es una descripción de las acciones de un sistema desde el punto de vista del usuario. Es una herramienta valiosa dado que es una técnica de aciertos y errores para obtener los requerimientos del sistema, justamente desde el punto de vista del usuario.
Los diagramas de caso de uso modelan la funcionalidad del sistema usando actores y casos de uso. Los casos de uso son servicios o funciones provistas por el sistema para sus usuarios.

Símbolos de los casos de uso

Sistema: El rectángulo representa los límites del sistema que contiene los casos de uso. Los actores se ubican fuera de los límites del Sistema.
Caso de uso: Se representan con óvalos. La etiqueta en el óvalo indica la función del sistema.
Actor: Un diagrama de caso de uso contiene los símbolos del actor y del caso de uso, junto con líneas conectoras. Los actores son similares a las entidades externas; existen fuera del sistema. El término actor se refiere a un rol específico de un usuario del sistema.

Las relaciones entre un actor y un caso de uso, se dibujan con una línea simple. Para relaciones entre casos de uso, se utilizan flechas etiquetadas “incluir” o “extender.” Una relación “incluir” indica que un caso de uso es necesitado por otro para poder cumplir una tarea. Una relación “extender” indica opciones alternativas para un cierto caso de uso.

Relaciones de los casos de uso

Las relaciones activas se conocen como relaciones de comportamiento y se utilizan principalmente en los diagramas de casos de uso. Hay cuatro tipos básicos de relaciones de comportamiento: comunica, incluye, extiende y generaliza.

Documentación de los casos de uso

Existen dos formas principales de documentar un caso de uso:

- Un diagrama en UML
- Un documento detallado

Documentar casos de usos no es una tarea fácil que se pueda dominar de un día para otro, requiere de tiempo, disciplina y experiencia, sin embargo, podemos definir una serie de pasos identificables para escribir los casos de uso.

2.7 CONCEPTOS BÁSICOS DE LA ORIENTACIÓN A OBJETOS: **METODOLOGÍAS.**

La orientación a objetos se basa en los siguientes conceptos elementales, que facilitan el abstraer los diferentes:

- ☐ Clase
- ☐ Objeto
- ☐ Atributo
- ☐ Método
- ☐ Herencia
- ☐ Polimorfismo
- ☐ Encapsulamiento

Una clase es un conjunto de objetos que comparten una estructura y comportamiento comunes. En el mundo real, normalmente tenemos muchos objetos del mismo tipo. Por ejemplo, nuestro teléfono celular es sólo uno de los miles que hay en el mundo. Si hablamos en términos de la programación orientada a objetos, podemos decir que nuestro objeto celular es una instancia de una clase conocida como "celular". Los celulares tienen características (marca, modelo, sistema operativo, pantalla, teclado, etc.) y comportamientos (hacer y recibir llamadas, enviar mensajes multimedia, transmisión de datos, etc.). Podemos extrapolar este concepto a cualquier conjunto ya sean entidades tangibles o abstracciones, reales, imaginarias, etc.

Las clases nos permiten tener todas las características y comportamientos (las variables y métodos) en una sola entidad, algo que en los lenguajes estructurados esto era imposible. A esto se le conoce como encapsulamiento y lo abordaremos más adelante. Entonces ¿qué es un objeto? Entender que es un objeto es la clave para entender cualquier lenguaje o método orientado a objetos. Un objeto representa un ítem individual e identificable, o una entidad real o abstracta, con un papel definido en el dominio del problema. Un objeto no es un dato sino una instancia de una clase, contiene en su interior cierto número de componentes bien estructurados. Es posible intercambiar los términos objeto o instancia, los objetos son pues ejemplares de una clase cualquiera. La estructura y el comportamiento de objetos similares se definen en sus clases comunes. Un objeto dado se caracteriza por tener:

- ☐ Estado
- ☐ Comportamiento
- ☐ Identidad

El Comportamiento es como un objeto actúa y reacciona, en términos de sus cambios de estado y de los mensajes que intercambia. El Estado de un objeto representa los resultados acumulados de su comportamiento. Los Objetos de Software mantiene sus características (identidad) en una o más "variables" o "atributos" (su estado), e implementa su comportamiento con "métodos" mediante los que interactúa con otros objetos o altera sus propios atributos.

La Identidad es la propiedad de un objeto que lo lleva a distinguirse de otros. En programación la identidad de los objetos sirve para comparar si dos objetos son iguales o no. En muchos lenguajes de programación la identidad de un objeto esté determinada por la dirección de memoria de la computadora en la que se encuentra el objeto.

La API, Atributos y Métodos

Un Atributo es una variable, un contenedor de algún tipo de dato asociado a la clase. Los valores de los tributos pueden ser alterados por la ejecución de algún método.

Método: es un término utilizado en algunos lenguajes de programación para referirse a algún comportamiento de los objetos de una clase, un algoritmo asociado a una clase que se desencadena después de recibir un mensaje. Es lo que el objeto puede hacer.

Características de la Programación orientada a objetos (POO)

La Herencia es el mecanismo fundamental de relación entre clases en la orientación a objetos. Relaciona las clases de manera jerárquica; una clase padre o superclase sobre otras clases hijas o subclases. Es decir que una clase puede heredar sus variables y métodos a varias subclases. Por tanto, la subclase, aparte de los atributos y métodos propios, tiene incorporados los atributos y métodos heredados de la superclase. Los métodos heredados pueden ser polimorfos, de forma que las subclases pueden responder de forma diferente al mismo mensaje que se envía a su clase base. De esta manera se crea una jerarquía de herencia, una jerarquía de clases cada vez más especializada.

En la orientación a objetos, se consideran dos tipos de herencia, simple y múltiple. En el caso de la primera, una clase sólo puede derivar de una única superclase. Para el segundo tipo, una clase puede descender de varias superclases.

Algunos lenguajes orientados a objetos, como C++ permiten herencias múltiples, lo que significa que una clase puede heredar los atributos de otras dos superclases. Este método puede utilizarse para agrupar atributos y métodos desde varias clases dentro de una sola.

En otros lenguajes, como Java, sólo se dispone de herencia simple, para una mayor sencillez del lenguaje, si bien se compensa de cierta manera la inexistencia de herencia múltiple con un concepto denominado interface.

La herencia ofrece una ventaja importante, permite la reutilización del código. Una vez que una clase ha sido depurada y probada, el código fuente de dicha clase no necesita modificarse. Su funcionalidad se puede cambiar derivando una nueva clase que herede la funcionalidad de la clase base y le añada otros comportamientos. Reutilizando el código existente, el programador ahorra tiempo y dinero, ya que solamente tiene que verificar la nueva conducta que proporciona la clase derivada.
La palabra Polimorfismo proviene del griego y significa que posee varias formas diferentes. Así como la herencia está relacionada con las clases y su jerarquía, el polimorfismo se relaciona con los métodos. Es la capacidad que tienen los objetos de una clase de responder al mismo mensaje o evento en función de los parámetros utilizados durante su invocación.

El polimorfismo es una característica que merece un artículo propio, por ahora nos limitaremos a conocer lo tipos. Se puede clasificar el polimorfismo en dos grandes clases:

Polimorfismo dinámico (o polimorfismo paramétrico) es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo de datos sobre el que se trabaja. Así, puede ser utilizado a todo tipo de datos compatible.
Polimorfismo estático (o polimorfismo ad hoc) es aquél en el que los tipos a los que se aplica el polimorfismo deben ser explicitados y declarados uno por uno antes de poder ser utilizados.
Diferencias entre polimorfismo y sobrecarga. La sobrecarga se da siempre dentro de una sola clase, mientras que el polimorfismo se da entre clases distintas. Un método está sobrecargado si dentro de una clase existen dos o más declaraciones de dicho método con el mismo nombre, pero con parámetros distintos, por lo que no hay que confundirlo con polimorfismo.

La sobrecarga se resuelve en tiempo de compilación utilizando los nombres de los métodos y los tipos de sus parámetros; el polimorfismo se resuelve en tiempo de ejecución del programa, esto es, mientras se ejecuta, en función de que clase pertenece un objeto.

Encapsulamiento

Hay muchos datos que no tiene por qué conocerlo aquel que este usando una clase; ya que son inherentes al objeto y solo controlan su funcionamiento interno, es decir el usuario no necesita conocer la implementación. Al evitar que el usuario modifique los atributos directamente y forzándolo a utilizar funciones definidas para modificarlos (llamadas interfaces), se garantiza la integridad de los datos.

Esto es el Encapsulamiento, encapsulación u ocultación. Consiste en el ocultamiento del estado, es decir, de los datos miembro, de un objeto de manera que sólo se puede cambiar mediante las operaciones definidas para ese objeto. El aislamiento protege a los datos asociados a un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

El encapsulamiento es muy conveniente y nos permite colocar en funcionamiento nuestro objeto en cualquier tipo de sistema, de una manera modular y escalable.

La encapsulación da lugar a que las clases se dividan en dos partes:

1. Interface: captura la visión externa de una clase, abarcando la abstracción del comportamiento común a los ejemplos de esa clase.
2. Implementación: comprende la representación de la abstracción, así como los mecanismos que conducen al comportamiento deseado.

La encapsulación define los niveles de acceso para elementos de la clase. Estos niveles de acceso definen los derechos de acceso para los datos, permitiéndonos el acceso a datos a través de un método de esa clase en particular, desde una clase heredada o incluso desde cualquier otra clase. En forma general la visibilidad se aplica tanto a métodos como atributos. Cada lenguaje implementa la forma de aplicar el principio de ocultación.

Existen tres niveles de acceso:

☐ Privado: Son los elementos que solo pueden ser accedidos directamente por la clase que los define. El símbolo usado para su representación es el menos "-".
☐ Protegido: Los elementos protegidos son aquellos que pueden ser accedidos por las clases descendientes o clases que compartan el mismo espacio físico "paquete", "namespace", etc. El símbolo usado es el numeral "#".
☐ Público: Estos son los elementos en los cuales no hay restricción alguna y pueden ser accedidos por cualquier clase y objeto del modelo. El símbolo usado es el más "+".

2.9 EL LENGUAJE DE MODELADO UNIFICADO (UML)

El 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), esta asociación se

encarga de la definición y mantenimiento de estándares para aplicaciones de la industria de la computación. UML es un lenguaje gráfico que permite especificar, modelar, construir y documentar los elementos que forman un sistema software, principalmente orientado a objetos, sin embargo, UML no está diseñado exclusivamente para software orientado a objetos.

A continuación, se especifica cada una de las palabras del UML:

☐ Lenguaje: el UML es un lenguaje. Existen reglas sobre cómo deben agruparse los elementos del lenguaje y el significado de esta agrupación.
☐ Modelado: el UML es visual. Mediante su sintaxis se modelan distintos aspectos del mundo real, que permiten una mejor interpretación y entendimiento de éste.
☐ Unificado: unifica varias técnicas de modelado en una única.

La notación UML se deriva y unifica las tres metodologías de análisis y diseño Orientado a Objetos 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: ObjectModeling Technique).
☐ Aproximación de Ivar Jacobson (OOSE: Object- Oriented Software Engineering) mediante la metodología de casos de uso (use case).

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 ayuda a visualizar el diseño y a hacerlo más accesible para otros.

UML se compone de muchos elementos de esquematización que representan las diferentes partes de un sistema de software. Los elementos UML se utilizan para crear diagramas, que representan alguna parte o punto de vista del sistema, UML contiene 13 tipos diferentes de diagramas. Para comprenderlos de manera concreta, es útil clasificarlos por su jerarquía.

Los Diagramas de Estructura muestran cuales son los elementos que deben existir en el sistema modelado:

1. Diagrama de clases
2. Diagrama de componentes
3. Diagrama de objetos
4. Diagrama de estructura compuesta (UML)
5. Diagrama de despliegue
6. Diagrama de paquetes

Los Diagramas de Comportamiento muestran lo que debe suceder en el sistema modelado:

1. Diagrama de actividades
2. Diagrama de casos de uso
3. Diagrama de transición de estados

Los Diagramas de Interacción son un subtipo de diagramas de comportamiento, que están enfocados al flujo de control y de datos entre los elementos del sistema modelado:

1. Diagrama de secuencia
2. Diagrama de colaboración
3. Diagrama de tiempos (UML)
4. Diagrama de vista de interacción (UML)

En las tres secciones siguientes se define y ejemplifica un tipo de diagrama por cada uno de los tres grupos mencionados anteriormente; los Diagramas de Clases pertenecen al grupo de los Diagramas de Estructura, los Diagramas de Casos de Uso pertenecen al grupo de los Diagramas de Comportamiento y finalmente, los Diagramas de secuencia pertenecen al grupo de los Diagramas de Interacción.

Diagramas de clases

Los diagramas de clases representan las estructuras estáticas de un sistema, incluidas sus clases, atributos, operaciones y objetos. Un diagrama de clases puede mostrar datos computacionales u organizacionales en la forma de clases de implementación y clases lógicas, respectivamente. Puede haber superposición entre estos dos grupos.

1. Las clases se representan con una forma rectangular dividida en tercios. La sección superior muestra el nombre de la clase, mientras que la sección central contiene los atributos de la clase. La sección inferior muestra las operaciones de la clase (también conocidas como métodos).

2. Agrega formas de clases a tu diagrama de clases para modelar la relación entre esos objetos. Además, podría ser necesario que agregues subclases.

3. Usa líneas para representar asociación, traspaso, multiplicidad y otras relaciones entre clases y subclases. Tu estilo de notación preferido informará la notación de estas líneas.

Diagramas de implementación

Un diagrama de implementación modela la implementación física y la estructura de los componentes de hardware. Los diagramas de implementación muestran dónde y cómo operarán los componentes de un sistema en conjunto con los demás.

1. Al trazar un diagrama de implementación, usa la misma notación que usas para un diagrama de componentes.
2. Usa un cubo 3D para modelar un nodo (lo cual representa una máquina física o máquina virtual).
3. Etiqueta el nodo con el mismo estilo que se usa para los diagramas de secuencia. Agrega otros nodos según sea necesario, luego conéctalos con líneas.

2.10 MODELADO ESTRUCTURAL

CLASES

¿Qué es un diagrama de clases?

Un diagrama de clases muestra la existencia de clases y sus relaciones en el diseño lógico de un sistema. Un diagrama de clases puede representar todo o parte de la estructura de un sistema. Los diagramas de clase muestran la estructura de un modelo en particular, las entidades que deben existir, su estructura interna y las relaciones con otras clases. Los diagramas de clases no muestran información temporal.

Usos de un diagrama de clases

Cuando se modela la vista estática de un sistema, los diagramas de clases son utilizados en alguna de las siguientes formas:

- Capturar el vocabulario de un sistema.
- Representar las clases participantes en una colaboración.
- Modelar el esquema lógico de una base de datos.

RELACIONES

¿Qué es una Relación?

Una relación puede definirse como:

- Una relación estructural entre dos o más clases, que representa que una de ellas utiliza los servicios de las demás.
- Las relaciones son representadas en un diagrama de clases a través de una línea que las interconecta.
- Una clase puede tener relaciones reflexivas.
- Una asociación se puede adornar con indicadores de multiplicidad, un nombre y los roles que desempeñan las clases de los extremos.

De los diferentes tipos de relaciones a las que más se les da importancias son: las dependencias, las generalizaciones y asociaciones. También existe la relación de realización.

Dependencias

Es una relación de significado entre dos elementos, donde cualquier cambio a un elemento independiente, puede afectar el significado de otro elemento dependiente.

Las dependencias generalmente representan relaciones de uso que manifiestan que un cambio en la especificación de un elemento puede afectar a otro que la utiliza, pero no necesariamente a la inversa.

Las clases o paquetes utilizan en su mayoría en su contexto, para indicar que una clase utiliza a otra como argumento en la asignatura de una operación.

INTERFACES Y PAQUETES

Un interfaz es un elemento modelo que define un sistema de los comportamientos (un sistema de operaciones) ofrecidos por un elemento del modelo del clasificador (específicamente, una clase,

un subsistema o un componente). La relación entre las interfaces y los clasificadores (subsistemas) no es siempre de una a una. Los clasificadores múltiples pueden realizar una interfaz y un clasificador pueden realizar interfaces múltiples. La realización es una relación semántica entre dos clasificadores. Un clasificador sirve como el contrato que el otro clasificador acuerda realizar.

Las interfaces son una evolución natural de las clases públicas de un paquete a las abstracciones fuera del subsistema. Todas las clases dentro del subsistema son privadas y no accesibles del exterior.

Una interfaz es una especificación pura. Las interfaces proporcionan la "familia de comportamiento" que un clasificador, pone la interfaz en ejecución. Las interfaces tienen vidas separadas de los elementos que los realizan. Esta separación de la interfaz y de la implementación ejemplifica los conceptos de modularidad y la encapsulación, así como el polimorfismo.

Identificando Interfaces

Una vez que se identifiquen los subsistemas, sus interfaces necesitan ser identificados. Identifique las interfaces del candidato. Organice las responsabilidades del subsistema en los grupos de responsabilidades cohesivas, relacionadas.

Semejanzas entre las interfaces. Busque los nombres similares, las responsabilidades similares, y las operaciones similares. Extraiga las operaciones comunes en una nueva interfaz. Busque interfaces existentes también, reutilizándolas en lo posible.

Defina las dependencias de la interfaz. Agregue las relaciones de la dependencia de la interfaz a todas las clases e interfaces que aparezcan en las firmas de la operación de la interfaz.

Mapee las interfaces a subsistemas. Cree las asociaciones de la realización del subsistema a la(s) interfaz(ces) que realiza.

Defina el comportamiento especificado por las interfaces. Si las operaciones en la interfaz se deben invocar en una orden particular, defina una máquina del estado que ilustre los estados público visible.

Empaquete las interfaces. Las interfaces se pueden manejar y controlar independientemente de los subsistemas. Se dividieron las interfaces según sus responsabilidades.

Guía para las Interfaces

Nombre de la interfaz: El nombre de la interfaz para reflejar el papel que juega en el sistema. El nombre debe ser corto, de 1 a 2 palabras. No es necesario incluir la palabra "interfaz" en el nombre.

Descripción de la interfaz: La descripción debe corresponder con sus responsabilidades. Debe ser de varias oraciones. La descripción no debe exponer simplemente el nombre en forma modificada de la interfaz, sino que debe expresar el papel que la interfaz desempeña en el sistema.

Definición de la operación: Cada interfaz debe proporcionar un sistema único y bien definido de operaciones. Los nombres de la operación deben reflejar el resultado de la operación.

Descripción de la operación: Debe describir lo que hace la operación, incluyendo cualquier algoritmo dominante, y qué valor devuelve. Nombre los parámetros de la operación para indicar qué información se está pasando a la operación.

Documentación de la interfaz: El comportamiento definido por la interfaz se especifica como sistema de operaciones.

¿Qué es un paquete?

Un paquete se puede definir como:

Un mecanismo de propósito general para organizar elementos en grupos.

- Los modelos pueden contener cientos, o incluso, miles de elementos. Este número de elementos puede llegar a ser rápidamente inmanejable. En consecuencia, es crítico poder agrupar los elementos del modelo en colecciones lógicas que sean fáciles de mantener y fáciles de localizar dentro del modelo.
- Los paquetes son mecanismos que permiten agrupar elementos semánticamente relacionados.
- Un paquete contiene clases, pero no agrega ningún comportamiento adicional al ya definido en estas clases.
- En UML un paquete es representado a través de un folder.

2.11 Grupo de Diseño de Clases en Paquetes

Al identificar clases, éstas se deben agrupar en los paquetes para los propósitos de organización y configuración. El modelo de diseño se puede estructurar en unidades más pequeñas para hacerlo más comprensible. Agrupando los elementos modelo del diseño en los paquetes y los subsistemas, y mostrando cómo esas agrupaciones se relacionan una con otra, es más fácil entender la estructura total del modelo.

Es importante dividir el modelo de diseño por varias razones:

- Se pueden utilizar los paquetes y los subsistemas como orden, configuración o unidades de la entrega cuando termina un sistema.
- La asignación de los recursos y la capacidad de diversos equipos del desarrollo pueden requerir que el proyecto esté dividido entre diversos grupos en diversos sitios.
- Los subsistemas se pueden utilizar para estructurar el modelo del diseño de una manera que refleje los tipos del usuario.
- Los subsistemas aseguran de que los cambios de un tipo particular del usuario afecten solamente las partes del sistema que corresponden a ese tipo del usuario.

Los subsistemas se utilizan para representar los productos existentes y los servicios del sistema.

Recomendaciones de empaquetado: Clases

Cuando las clases del límite se distribuyen a los paquetes, hay dos diversas estrategias que pueden ser aplicadas:

- Es probable que la interfaz del sistema sea substituida o experimente cambios considerables, la interfaz debe ser separada del resto del modelo del diseño. Cuando se cambia la interfaz utilizadora, sólo se afectan estos paquetes.
- Si no se perciben modificaciones importantes en la interfaz, los cambios a los servicios del sistema deben ser la guía que se maneje en principio, debido a los cambios en la interfaz. Las clases frontera deben ser colocadas junto con la entidad y las clases de control con las cuales son funcionalmente relacionadas.

Es fácil observar que las clases frontera se ven afectadas si se cambia cierta clase de la entidad o del control. Las clases frontera obligatorias que no se relacionan funcionalmente con ninguna entidad o clases de control se deben colocar en paquetes separados con las clases frontera que pertenecen a la misma interfaz.

Si una clase frontera se relaciona con un servicio opcional, se debe agrupar en un subsistema separado con las clases que colaboran para proporcionar el servicio.

Dependencia de Paquetes: Visibilidad de Elementos de Paquete

La visibilidad se puede definir para los elementos del paquete de la misma manera que se define para las cualidades y las operaciones de la clase. Esta visibilidad permite que usted especifique cómo otros paquetes pueden tener acceso a los elementos que son poseídos por el paquete.
La visibilidad de un elemento del paquete puede ser expresada incluyendo un símbolo de la visibilidad como prefijo al nombre del elemento del paquete.

Hay tres tipos de visibilidad definidos en el UML:
Public: Las clases públicas se pueden alcanzar afuera del paquete que posee. Visibility symbol:

+ Protected: Las clases protegidas se pueden alcanzar solamente por el paquete que posee, y cualquier paquete que herede del paquete que posee. Visibility symbol: #
Private: Las clases privadas se pueden alcanzar solamente por las clases dentro del paquete que posee. Visibility symbol: -
Los elementos públicos de un paquete constituyen la interfaz del paquete. Todas las dependencias en un paquete deben ser dependencias en los elementos públicos del paquete. La visibilidad del paquete proporciona la ayuda para el principio de OO de la encapsulación.

Paquetes de Acoplamiento: Recomendaciones
Un acoplador de paquete puede ser bueno y malo. Es bueno cuando el acoplador representa la reutilización, y malo cuando el acoplador representa las dependencias que hacen el sistema más difícil de cambiar y desarrollarse.

Algunos principios generales a seguir:

• Los paquetes no deben cruzar los pares (es decir codependiente); por ejemplo, dos paquetes no deben depender uno de otro. En estos casos, los paquetes necesitan ser reorganizados para quitar las dependencias de crce.

• Los paquetes en capas más bajas no deben depender de los paquetes en capas superiores. Los paquetes deben solamente ser dependientes sobre los paquetes en la misma capa y en la capa

más baja siguiente. En estos casos, la funcionalidad necesita estar repartida. Una solución es indicar las dependencias en términos de interfaces, y organizar las interfaces en la capa más baja.

- Las dependencias no deben saltarse capas, a menos que el comportamiento dependiente sea común a través de todas las capas. Los paquetes no deben depender de subsistemas, solamente de otros paquetes o de interfaces.

UNIDAD III MODELADO DE COMPORTAMIENTO

3.1 Diagramas

Son aquellos que muestran las interacciones de un usuario con el sistema. Interacción es una cadena de mensajes enviados entre los objetos en respuesta a un evento generado por el usuario sobre la aplicación.

Los diagramas de interacción pueden ser Diagramas de Secuencia y Diagramas de Colaboración. Estos diagramas conforman la etapa del diseño de la aplicación, y se crean a partir de los diagramas de Casos de Uso y el Conceptual.

Los Diagramas de Secuencia representan una interacción entre objetos de manera secuencial en el tiempo. Muestra la participación de objetos en la interacción entre sus "líneas de vida" (desde que son instancias) y los mensajes que ellos organizadamente intercambian en el tiempo. El responsable o ACTOR es quien inicia el ciclo interactuando inicialmente con la interfaz de usuario: GUI; en seguida se inician todos los objetos que intervienen en el funcionamiento del aplicativo. En este diagrama se comienza a observar el comportamiento del sistema a partir de los eventos generados por los actores. Aquí se interactúa con instancias, no con clases.

Los Diagramas de Colaboración dan todas las especificaciones de los métodos. Éstos permiten describir una operación específica incluyendo sus argumentos y variables locales creadas durante su ejecución. Se muestran los objetos y mensajes que son necesarios para cumplir con un requerimiento o propósito, o con un conjunto de ellos. Se pueden elaborar para una operación o para un Caso de Uso, con el fin de describir el contexto en el cual su comportamiento ocurre.

3.2. INTERACCIÓN (SECUENCIA Y COLABORACIÓN)

¿Qué es un modelo?

Un modelo captura una vista de un sistema del mundo real. Es una abstracción de dicho sistema, considerando un cierto propósito. Así, el modelo describe completamente aquellos aspectos del sistema que son relevantes al propósito del modelo, y a un apropiado nivel de detalle.

¿Qué es un Diagrama?

Un diagrama es una representación gráfica de una colección de elementos de modelado, a menudo dibujada como un grafo con vértices conectados por arcos.

¿Qué es la conducta de un sistema?

Ningún sistema vive asilado; cada sistema interactúa con personas, sistemas o dispositivos, con el fin de alcanzar un objetivo. Estas interacciones generan resultados predecibles. Estos resultados conforman la conducta del sistema.
Los casos de uso son el mecanismo para capturar la conducta deseada de un sistema que está bajo desarrollo. Estas especificaciones no contienen detalles de cómo esta conducta es implantada.
UML define un modelo para comunicar la conducta de un sistema: el modelo de casos de uso.

¿Qué es el comportamiento del sistema?
No existen sistemas en aislamiento. Cada sistema interactúa con la gente o sistemas automatizados para el mismo propósito. Estas interacciones resultan en algún tipo de resultado predecible.
Este resultado predecible es el comportamiento del sistema.
Los casos de uso son los mecanismos para capturar el mecanismo deseado, esto es, bajo el desarrollo, pero no especifica cómo debe ser implementado el comportamiento.
UML especifica un modelo para comunicar el comportamiento del sistema, el modelo de caso de uso.

3.3 Diagramas de secuencia.

Los Diagramas de secuencia pertenecen al grupo de los Diagramas de Interacción, sirven para describir los aspectos dinámicos del sistema, mostrando el flujo de eventos entre objetos en el tiempo. Muestran el intercambio de mensajes (es decir la forma en que se invocan) en un momento dado. Ponen especial énfasis en el orden y el momento en que se envían los mensajes a los objetos.

Los objetos están representados por líneas intermitentes verticales, con el nombre del objeto en la parte más alta. El eje de tiempo también es vertical, incrementándose hacia abajo, de forma que

los mensajes son enviados de un objeto a otro en forma de flechas con los nombres de la operación y los parámetros.

Los Diagramas de Colaboración. Se utilizan para describir la comunicación entre objetos de un sistema y también pertenecen al grupo del grupo de los Diagramas de Interacción.

3.4 ESTADOS

Un objeto tiene estado

El estado de un objeto son las condiciones en las que se encuentra en un momento en el tiempo. Este estado normalmente cambia a través del tiempo y es representado por los valores de los atributos del objeto en un momento determinado.

¿Qué son los diagramas de transición de estados?

Un diagrama de transición de estados muestra una máquina de estados, especifica la secuencia de estados en los que un objeto puede estar, los eventos y condiciones que ocasionan cambios de estado, y las acciones que toman lugar cuando el objeto alcanza un estado determinado.

Un estado es representado a través de un rectángulo con esquinas redondeadas. A continuación, se muestra las diferencias de representación entre un estado y una actividad.

Estados especiales

El estado inicial es aquel que se alcanza cuando un objeto es creado.

- Es obligatorio.
- Solo un estado inicial es permitido por diagrama.
- El estado inicial es representado a través de un circulo sólido.

Un estado final indica el final del ciclo de vida de un objeto.

- Son opcionales.
- Puede existir más de un estado final por diagrama.
- Un estado final es representado por un “ojo de toro”.

3.5 ACTIVIDADES

¿Qué es un diagrama de actividades?

El flujo de trabajo de un caso de uso describe lo que el sistema debe realizar para proporcionar el resultado de valor que el sistema espera.

Consiste en una secuencia de actividades que en conjunto producen un resultado de valor a un actor. Este flujo de trabajo contiene un flujo básico y flujos alternos que representan las condiciones de error que se pueden generar.

Este flujo de trabajo puede ser descrito gráficamente a través de un diagrama de actividades.

¿Qué es una actividad?

Una actividad es la ejecución de un conjunto de pasos no atómicos que:

- Puede descomponerse.
- Puede ser interrumpida y toma cierto tiempo su ejecución.

Una actividad representa la ejecución de un paso dentro de un flujo de trabajo. Un diagrama de actividades puede incluir alguno de los siguientes elementos:

- Actividad, que representa la realización de un paso de un flujo de trabajo.
- Transiciones, que representan el paso de una actividad a otra.
- Decisiones, también conocidas como guardas. Estas condiciones de guarda controlan hacia dónde va el flujo una vez que una actividad concluye, es decir, para que una transición que tiene una guarda se pueda ejecutar, se debe evaluar a verdadero.
- Barras de sincronización, permiten modelar actividades concurrentes o en paralelo.

Un diagrama de actividades puede ser dividido en callejones utilizando líneas verticales. Cada callejón representa a un responsable de ejecutar las actividades que contiene.

Los diagramas de actividades muestran el flujo de control de procedimiento entre objetos de clases, junto con procesos organizacionales, como los flujos de trabajo de negocios. Estos diagramas se integran con formas especializadas que luego se conectan con flechas. La notación establecida para los diagramas de actividades es similar a la de los diagramas de estados.

1. Empieza tu diagrama de actividades con un círculo negro.
2. Conecta el círculo a la primera actividad, la cual se modela con un rectángulo redondeado.
3. Ahora, conecta cada actividad a otras actividades con líneas que muestren el flujo paso a paso de todo el proceso.
4. También puedes probar usar carriles para representar los objetos que realizan cada actividad.

3.6 MODELADO ARQUITECTÓNICO

Análisis arquitectónico en contexto

El análisis arquitectónico es cuando el proyecto (o el arquitecto) decide cómo hacer funcionar el análisis. Se centra sobre todo en la limitación del esfuerzo del análisis en términos de patrones y de idiomas arquitectónicos acordados.

El análisis arquitectónico es una configuración del análisis de caso de uso ya que nos concentramos en las capas superiores del sistema, haciendo una tentativa inicial en definir las piezas/partes del sistema y de sus relaciones y la organización de estas piezas/partes en capas bien definidas con dependencias explícitas.

En el análisis arquitectónico de casos de uso, nos basamos identificando clases del análisis de los requerimientos. Entonces, adentro incorporamos los elementos existentes del diseño, se refina la arquitectura inicial, y las capas más bajas de la arquitectura se definen, considerando el ambiente de la puesta en práctica y cualquier otro apremio de la puesta en práctica.

El análisis arquitectónico se hace generalmente una vez por proyecto, primero en la fase de la elaboración. El arquitecto del software o el equipo de la arquitectura realiza la actividad. Esta actividad puede ser saltada si el riesgo arquitectónico es bajo o no se cuenta con ningún arquitecto experimentado que tenga experiencia.

El propósito es:

1. Definir los patrones arquitectónicos, mecanismos dominantes y las convenciones del modelado para el sistema.
2. Definir la estrategia de la reutilización.
3. Proporcionar la entrada al proceso de planeamiento.

Artefactos de la Entrada:

1. Modelo de casos de uso
2. Especificaciones suplementarias
3. Glosario
4. Modelo de negocio
5. Documento de la Arquitectura del Software
6. Modelo de Diseño
7. Pautas de Diseño.

Artefactos que resultan:

1. Documento actualizado de la Arquitectura del Software.
2. Modelo actualizado de diseño.
3. Pautas actualizadas de diseño.
4. Realizaciones de casos de uso (apenas identificadas, no desarrolladas).

3.7 LAS 4+1 VISTAS

Un enfoque en la presentación de un sistema en UML es conocida como 4+1 vistas. Esta forma de documentar nuestros modelos divide lo que sabemos de él en cinco áreas:

Vista de Casos de Uso o Escenarios: que contiene requisitos desarrollados en las restantes vistas. Los escenarios describen secuencias de interacciones entre objetos, y entre procesos. Se utilizan para identificar y validar el diseño de arquitectura. También sirven como punto de partida para pruebas de un prototipo de arquitectura.

Vista Lógica: Muestra la estructura estática del sistema. (Diagramas de clases).

Vista Física: Muestra el despliegue de la aplicación en la red de computadoras. (Diagrama de despliegue).

Vista de Procesos: Muestra los hilos y procesos de ejecución, así como la comunicación entre estos. (Diagrama de actividades).

Vista de Desarrollo: Muestra la estructura en modelos del código del sistema. (Diagramas de componentes).

Estas vistas se presentan tradicionalmente en una figura de cuatro cajas con un ovalo central que representa al modelo de casos de uso.

El modelo "4+1" de Kruchten, es un modelo de vistas [1] diseñado por el profesor Philippe Kruchten y que encaja con el estándar "IEEE 1471-2000" (Recommended Practice for Architecture Description of Software-Intensive Systems [5]) que se utiliza para describir la arquitectura de un sistema software intensivo basado en el uso de múltiples puntos de vista.

Vale, si por ahora no te has enterado de nada y no estas en 3 o 4 de carrera de Ingeniería del Software (o derivados) no te preocupes es normal, y si estas en 3 o 4 de carrera y aun así no te has enterado de nada, ¡Ponte las pilas YA! Porque estas cosas te deberían (por lo menos) sonar.

Bueno, vamos por pasos. Antes de entrar a explicar mas en detalle el modelo de kruchten vamos a explicar e intentar dejar claro algunos conceptos como por ejemplo qué es un sistema software, qué es una vista y qué es un punto de vista.Lo primero es saber que es eso de "un sistema software", el cual lo definimos con la siguiente "ecuación" (made in jarroba.com)

Efectivamente, a grandes rasgos un sistema software es un software (mas o menos complejo) que "corre" en un determinado hardware (mas o menos complejo). Por ejemplo, todo el rollo de los "cajeros automáticos" es un sistema software ya que en un "hardware" que llamamos "cajero", se ejecuta algún tipo de programa (software) el cual nos permite realizar determinadas gestiones.

Otra cosa de la que habla este modelo de Kruchten es sobre los conceptos de vista y puntos de vista, pues bien una vista no es mas que una representación de todo el sistema software desde una determinada perspectiva, y un punto de vista se define como un conjunto de reglas (o normas) para realizar y entender las vistas.

Bien, sino te ha quedado muy claro que es esto de las vistas y los puntos de vista, vamos a explicarlo con una sencilla analogía del mundo de la arquitectura (de la arquitectura de las casas, edificios y esas cosas):

Si un arquitecto nos muestra un plano de una casa (como la de la siguiente imagen), nos esta mostrando una vista de la casa y como no tenemos ni idea de arquitectura, cuando nos explique o nos de un documento en el que explique que un determinado símbolo del plano representa a una puerta u otro símbolo representa una mesa, nos estará dado un punto de vista para que podamos entender el plano de la casa. Si mas tarde nos mostrase otro plano (o maqueta) de la casa, nos estaría dando otra vista de la casa y nos tendrá que explicar el nuevo punto de vista, es decir, que nos tendrá que explicar que significa cada símbolo u objeto de esa nueva vista.

Bueno pues vistos los conceptos de lo que son las vistas y los puntos de vista, y habiendo explicado que es un sistema software, uno ya se puede hacer a la idea de que va el modelo "4+1" vistas de Kruchten para la descripción de arquitecturas de sistemas software Pues sí, lo que propone Kruchten es que un sistema software se ha de documentar y mostrar (tal y como se propone en el estándar IEEE 1471-2000) con 4 vistas bien diferenciadas y estas 4 vistas se han de relacionar entre sí con una vista más, que es la denominada vista "+1". Estas 4 vista las denominó Kruchten como: *vista lógica, vista de procesos, vista de despliegue y vista física y la vista "+1"* que tiene la función de relacionar las 4 vistas citadas, la denominó vista de escenario.

Cada una de estas vistas ha de mostrar toda la arquitectura del sistema software que se esté documentando, pero cada una de ellas ha de documentarse de forma diferente y ha de mostrar aspectos diferentes del sistema software. A continuación, pasamos a explicar que información ha de haber en la documentación de cada una de estas vistas

Vista Lógica: En esta vista se representa la funcionalidad que el sistema proporcionara a los *usuarios finales*. Es decir, se ha de representar lo que el sistema debe hacer, y las funciones y servicios que ofrece. Para completar la documentación de esta vista se pueden incluir los diagramas de clases, de comunicación o de secuencia de UML

Vista de Despliegue: En esta vista se muestra el sistema desde la perspectiva de *un programador* y se ocupa de la gestión del software; o en otras palabras, se va a mostrar como esta dividido el sistema software en componentes y las dependencias que hay entre esos

componentes. Para completar la documentación de esta vista se pueden incluir los diagramas de componentes y de paquetes de UML

Vista de Procesos: En esta vista se muestran los procesos que hay en el sistema y la forma en la que se comunican estos procesos; es decir, se representa desde la perspectiva de un *integrador de sistemas*, el flujo de trabajo paso a paso de negocio y operacionales de los componentes que conforman el sistema. Para completar la documentación de esta vista se puede incluir el diagrama de actividad de UML.

Vista Física: En esta vista se muestra desde la perspectiva de *un ingeniero de sistemas* todos los componentes físicos del sistema así como las conexiones físicas entre esos componentes que conforman la solución (incluyendo los servicios). Para completar la documentación de esta vista se puede incluir el diagrama de despliegue de UML.

+1" Vista de Escenarios: Esta vista va a ser representada por los casos de uso software y va a tener la función de unir y relacionar las otras 4 vistas, esto quiere decir que desde un caso de uso podemos ver como se van ligando las otras 4 vistas, con lo que tendremos una trazabilidad de componentes, clases, equipos, paquetes, etc., para realizar cada caso de uso. Para completar la documentación de esta vista se pueden incluir el diagrama de casos de uso de UML.

3.8 CASOS DE USO

¿Qué es la conducta de un sistema?

Ningún sistema vive aislado, cada sistema interactúa con personas, sistemas o dispositivos, con el fin de alcanzar un objetivo. Estas interacciones generan resultados predecibles. Estos resultados conforman la conducta del sistema.

La conducta de un sistema es cómo actúa y reacciona a los estímulos de sus usuarios, es la actividad visible y verificable de un sistema y ésta es capturada en casos de uso.

Los casos de uso son el mecanismo para capturar la conducta deseada de un sistema que está bajo desarrollo. Estas especificaciones no contienen detalles de cómo esta conducta es implantada.

UML define un modelo para comunicar la conducta de un sistema: el modelo de casos de uso.

Un modelo de casos de uso describe los requerimientos funcionales de un sistema. Este modelo contiene las funciones deseadas y sirve como un contrato entre el cliente y los desarrolladores.

El cliente debe aprobar el modelo de casos de uso. Cuando esta aprobación se obtiene, tenemos la certeza de que sabemos lo que el sistema debe realizar. Este modelo también puede ser utilizado durante el desarrollo del sistema para realizar ajustes a la funcionalidad solicitada por el cliente.

Los participantes en el proyecto lo utilizan para entender mejor el sistema. Los diseñadores lo utilizan como la base para realizar su trabajo y obtener una visión general del sistema. El personal de pruebas los utiliza para planificar qué funcionalidad es la que debe ser validada.

Los documentadores los utilizan como base para escribir las guías de usuarios. El arquitecto los utiliza para identificar y validar la funcionalidad arquitectónicamente representativa. El líder de proyecto los utiliza para planificar las actividades del proyecto.

El rol más importante del modelo de casos de uso es comunicar el comportamiento del sistema a los clientes y usuarios finales. En consecuencia, debe ser fácil de entender.

Los actores son los usuarios y cualquier otro sistema que interactúa con el sistema que se está desarrollando.

Los actores ayudan a delimitar el sistema y nos dan una clara visión de lo que se supone que debe realizar. Los casos de uso son desarrollados con base en las necesidades de los actores, asegurando así que el sistema satisfaga todos los requerimientos.

Componentes principales del modelo de casos de uso

Un actor representa un conjunto coherente de roles que son desempeñados cuando interactúa con el sistema.

Típicamente un actor representa un rol que puede ser desempeñado por un humano, un dispositivo de hardware u otro sistema al momento de interactuar con el sistema en desarrollo.

Un caso de uso es una secuencia de acciones que un sistema realiza y que dan como resultado un elemento de valor a un actor en particular. Un caso de uso describe lo que el sistema hace, pero no especifica cómo lo hace.

¿Qué es un actor?

Un actor se puede definir como:

Un actor es cualquier elemento que intercambia información con el sistema y es externo a él. Un actor puede ser un usuario, un dispositivo de hardware u otro sistema. Un actor puede intercambiar información con el sistema activamente o ser un recipiente pasivo de información.

La diferencia entre un actor y un usuario particular de un sistema, es que el actor representa a todos los usuarios que interactúan a través de un mismo conjunto de operaciones.
En ocasiones un usuario puede desempeñar varios actores. Para entender totalmente el propósito de un sistema, debemos saber a quienes está dirigido.

¿Qué es un caso de uso?

Un caso de uso puede ser definido como:
Una secuencia de acciones que un sistema realiza y que originan un resultado de valor para un actor en particular.

- Un caso de uso modela un diálogo entre uno o más actores con el sistema.
- Un caso de uso describe las acciones que el sistema realiza para entregar un resultado de valor a un actor.

Casos de uso y actores

Es importante indicar cómo los actores se relacionan con los casos de uso. En consecuencia, una vez que un caso de uso es definido, se deben establecer los actores que interactúan con él. Para hacer esto, debemos definir una asociación entre ambos.

Los actores pueden ser conectados a casos de uso solamente con asociaciones. Una asociación entre un actor y un caso de uso indica que existe una comunicación entre ambos, donde ambos extremos son capaces de enviar y recibir mensajes.

3.9 COMPONENTES

Una parte del sistema modular, instalable y reemplazable que encapsula detalles de implantación y expone un conjunto de interfaces bien definidas.

Proporciona la realización física de un conjunto de interfaces.

Diagramas de componentes

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

Un diagrama de componentes muestra las organizaciones y dependencias lógicas entre componentes software, sean estos componentes de código fuente, binarios o ejecutables. Desde el punto de vista del diagrama de componentes se tienen en consideración los requisitos relacionados con la facilidad de desarrollo, la gestión del software, la reutilización, y las restricciones impuestas por los lenguajes de programación y las herramientas utilizadas en el desarrollo. Los elementos de modelado dentro de un diagrama de componentes serán componentes y paquetes.

Dado que los diagramas de componentes muestran los componentes software que constituyen una parte reusable, sus interfaces, y sus interrelaciones, en muchos aspectos se puede considerar que un diagrama de componentes es un diagrama de clases a gran escala. Cada componente en el

diagrama debe ser documentado con un diagrama de componentes más detallado, un diagrama de clases, o un diagrama de casos de uso.

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

Los diagramas de componentes muestran cómo se combinan los componentes para formar componentes más grandes o sistemas de software. Estos diagramas están diseñados para modelar las dependencias de cada componente en el sistema. Un componente es algo necesario para ejecutar una función de estereotipo. Un estereotipo de componente puede

constar de ejecutables, documentos, tablas de bases de datos, archivos o archivos de bibliotecas.

Representa un componente con una forma rectangular. Debe tener dos rectángulos pequeños en un lado o mostrar un icono con esa forma.

Agrega líneas entre formas de componentes para representar las relaciones pertinentes.

3.10. DESPLIEGUE

Los Diagramas de Despliegue muestran la disposición física de los distintos nodos que componen un sistema y el reparto de los componentes sobre dichos nodos.

Los estereotipos permiten precisar la naturaleza del equipo:

- Dispositivos
- Procesadores
- Memoria

Los nodos se interconectan mediante soportes bidireccionales que pueden a su vez estereotiparse.

Ejemplo de conexión entre nodos:
Cuando se trata de hardware y el software del sistema, se utiliza los diagramas de despliegue para razonar sobre la tipología de procesadores y dispositivos sobre los que re-ejecuta el software. Los diagramas de despliegue se utilizan para visualizar los aspectos estáticos de estos nodos físicos y sus relaciones y para especificar sus detalles para la construcción, como se muestra a continuación.
Un diagrama de despliegue es un diagrama que muestra la configuración de los nodos que participan en la ejecución y de los componentes que residen en ellos, gráficamente, un diagrama de despliegue es una colección de nodos y arcos.

Son los complementos de los diagramas de componentes que, unidos, proveen la vista de implementación del sistema. Describen la topología del sistema la estructura de los elementos de hardware y el software que ejecuta cada uno de ellos. Los diagramas de despliegue representan a los nodos y sus relaciones. Los nodos son conectados por asociaciones de comunicación tales como enlaces de red, conexiones TCP/IP.

Usos

1. Sistemas empotrados: Un sistema empotrado es una colección de hardware con una gran cantidad de software que interactúa con el mundo físico.

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

3. Sistemas completamente distribuidos: En el otro extremo se encuentra aquellos sistemas que son ampliamente o totalmente distribuidos y que normalmente incluyen varios niveles de servidores.

Ventajas

- Muestra un conjunto de nodos y sus relaciones.
- Se utilizan para describir la vista de despliegue estática de un sistema.

- Se relacionan con los diagramas de componentes, ya que un nodo normalmente incluye uno o más componentes.

Desventajas

- La posible falla en la modelación de un hardware.
- Tales sistemas contienen a menudo varias versiones de componentes software, alguno de los cuales pueden incluso migrar de un nodo a otro. El diseño de tales sistemas requiere tomar decisiones que permitan un cambio continuo de la topología del sistema.

3.11 Diseño de alto nivel y diseño detallado

Durante el diseño de alto nivel se determina una solución arquitectónica, es decir, una descomposición del sistema en subsistemas. Los subsistemas son módulos que pueden descomponerse en sub-módulos. A su vez, cada sub-módulo se descompone en sub-módulos a este proceso se le conoce como modularización. Además, se lleva a cabo el ensamblaje, que es la definición de la comunicación y dependencia entre los módulos. También se toma en cuenta la posibilidad de aplicar una arquitectura ya existente reutilizando algún patrón de diseño conocido. Además, se considera en qué ambiente va a funcionar el sistema. El diseño de alto nivel (diseño arquitectónico) según la IEEE-1471 [Maier et. al, 2004] es "la organización fundamental de los componentes de un sistema, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución".

El diseño de alto nivel o diseño arquitectónico tiene como principal tarea especificar la organización del software, considerando los componentes que integrarán el sistema, las interfaces y comportamientos que caracterizan a estos componentes, así como la forma en que se establece la comunicación, interacción y colaboración entre estos componentes. La arquitectura del software resultante de este proceso de diseño a alto nivel deberá integrar todos los aspectos del software: lógicos, de proceso, de componentes, físicos, etc El diseño de alto nivel se ocupa de tres puntos esenciales: 1).- La definición de la arquitectura (módulos que componen el sistema), 2).- La definición de la estructura de los datos que van a usar los módulos, 3).- La definición de las interfaces con el usuario y entre los módulos, y 4).- La propuesta de una solución que cumpla con los requerimientos no funcionales, tales como fiabilidad, seguridad, eficiencia en tiempo/espacio, etc

El diseño detallado describe a detalle cada uno de los datos que comparten los módulos de la solución, sirve como guía para codificar de tal forma que se satisfagan los requerimientos, y también sirve como explicación de lo que debería estar codificado, lo que facilita las pruebas y el mantenimiento futuro.

Como se ilustra en la Figura 5-1, los módulos que componen al sistema y sus interfaces son producto del diseño de alto nivel. El producto del diseño de alto nivel es la entrada para comenzar el diseño detallado. Durante el diseño detallado e especifican los datos y la lógica de lo que se tiene que hacer y para lograrlo se utilizan diagramas de secuencia, de flujo, de estado, de clases, etc. Podemos decir ue el diseño detallado específica los requerimientos de cada módulo. El diseño detallado sirve como entrada para el proceso de codificación.

3.12 Diseño Estructurado

El diseño estructurado es útil para resolver problemas de naturaleza algorítmica, es decir, dada cierta entrada se produce cierta salida. La información fluye a través de las estructuras y de llamados a funciones. La modularización se lleva a cabo por medio de la descomposición jerárquica del problema (ver sección 5.5). El diseño estructurado se recomienda para sistemas pequeños.

Cuando la construcción del software es guiado por el diseño estructurado, entonces:

☐ El sistema software es una jerarquía de módulos, con un módulo principal (también llamado programa principal) con una función de controlador.

El módulo principal transfiere el control a los módulos inmediatamente subordinados (o subprogramas), de modo que éstos puedan ejecutar sus funciones. Una vez que el módulo subordinado haya completado su tarea, devolverá nuevamente el control al módulo controlador.

☐ La descomposición de un módulo en submódulos continúa hasta que se llegue a un punto en que el módulo resultante tenga solo una tarea específica que ejecutar (lectura, salida de resultados, procesamiento de datos o control de otros módulos).

Las estructuras de datos y las funciones se representan por medio de diagramas de entidad-relación, de flujo de datos y de transición de estados El DFD se apoya en el Diccionario de Datos para documentar más ampliamente sus elementos, el Diccionario de Datos es un documento aparte en el que se explica en qué consisten los términos que se usan un diagrama. Un DFD es independiente del tamaño y de la complejidad del sistema, porque se puede organizar por niveles

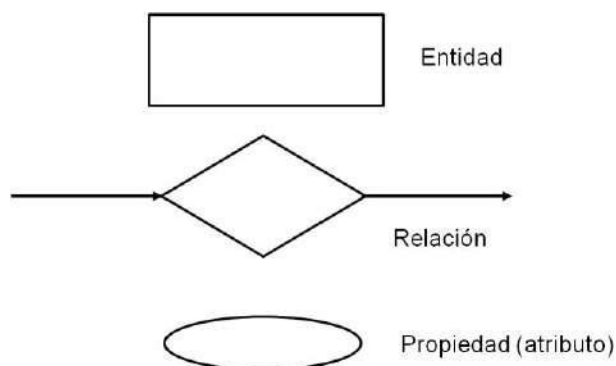
de abstracción, es decir, se comienza con un diagrama quedescribe el flujo de datos de manera general, y después se van haciendo diagramas adicionales para detallar los procesos que sean necesarios. las actividades que tienen que ver con los clientes, productos y proveedores de una empresa.

El Diagrama de Transición de Estados (DTE) se utiliza cuando el comportamiento del sistema puede ser distinto para un mismo estímulo. Esto se debe a que el sistema puede estar en diferentes estados y el comportamiento depende no solamente de estímulo que éste recibe, sino también del estado en el que se encuentre. En un DTE se indican todos los estados del sistema, aquellos eventos que estén relacionados con cambios de estado y los cambios de estado que se producen. Los componentes de un DTE son los estados (nodos) y las transiciones (flechas que representan cambios de estado). Además, se pueden agregar condiciones y las acciones asociadas a las transiciones.

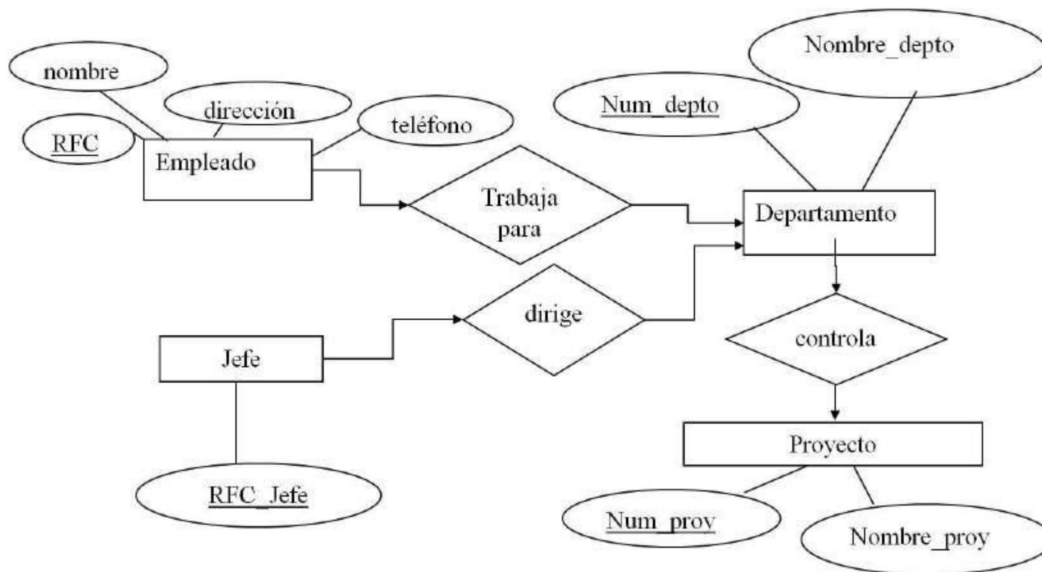
Un DTE relaciona eventos y estados. Cuando se recibe un evento, el estado siguiente depende del actual, así como del evento recibido. Un cambio de estado causado por un evento es lo que se conoce como transición.

3.13 Diagrama Entidad-Relación (DER)

Un diagrama Entidad-Relación (DER) modela el mundo real como un conjunto de objetos básicos llamados entidades y las relaciones existentes entre ellas. En la Figura 5-7 se ilustran los elementos de un DER.



El ejemplo de la Figura 5-8 modela con un DER las entidades y las relaciones que existen en una empresa en la que hay empleados, departamentos, jefes de departamento y proyectos que se llevan a cabo en los departamentos.



El Diseño Orientado a Objetos se basa en objetos que tienen un estado y un comportamiento tomando en cuenta que existen interacciones entre los objetos.

Este tipo de diseño es útil cuando el sistema se puede modelar de forma casi análoga a la realidad, porque así se simplifica el diseño de alto nivel. Con el Diseño Orientado a Objetos se promueve la reutilización, ya que las similitudes entre objetos se programan en forma abstracta y el programador concentra su esfuerzo en las diferencias concretas.

Bajo el principio del encapsulamiento, el software queda organizado y protegido, así un programador puede entender mejor el código de los otros y hay menor riesgo de que sus cambios afecten el trabajo de los demás. Por esto, el diseño orientado a objetos se usa en el desarrollo de software a gran escala, pues los equipos de programadores trabajan sobre objetos diferentes y posteriormente se integra el trabajo de todos haciendo uso de las interfaces entre los objetos.

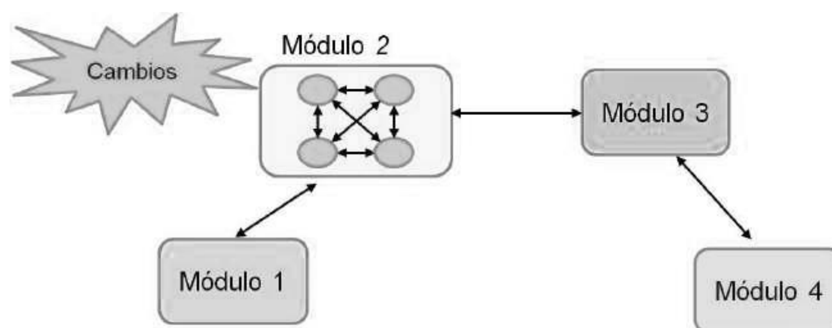
3.14 La descomposición (modularización) en el diseño

La descomposición según Braude (2003) consiste en desglosar un problema para que tenga las características de varios programas pequeños y sirve para afrontar la complejidad del problema. A la descomposición también se le llama modularización. En la programación estructurada los módulos son procedimientos y funciones que en conjunto proporcionan la funcionalidad del sistema, mientras que en la programación orientada a objetos los módulos son las clases que intervienen en la solución.

El principio del diseño modular consiste en descomponer el sistema en módulos. Cuando usamos el paradigma orientado a objetos, un módulo comúnmente consiste de un conjunto de clases e interfaces relacionadas. A nivel del diseño del sistema de software, se considera un módulo como un conjunto de clases e interfaces estrechamente relacionadas entre sí. A nivel del análisis, el módulo puede contener otros elementos tales como diagramas de casos de uso, etc

Hay dos aspectos a tomar en cuenta durante la modularización del software: la cohesión y el acoplamiento. La cohesión es el grado de relación que tienen entre sí las actividades que realiza un módulo. Es decir, el grado en el que un módulo se dedica a realizar un solo tipo de tareas. Y el acoplamiento es el grado en el que un módulo requiere comunicarse con otros módulos para realizar sus tareas. Para lograr una modularización efectiva se requiere que exista una alta cohesión en los módulos, es decir, cada módulo debe dedicarse, en la mayor medida posible, a realizar un mismo tipo de tareas. Además, se requiere que exista un bajo acoplamiento. Esto significa que los módulos deben hacer sus tareas de la forma más independiente que se pueda

.



3.15 Los patrones de arquitectura y diseño

Un patrón de arquitectura o diseño es una solución que ha probado ser útil para resolver cierto tipo de problemas. Consiste en una combinación de componentes que resuelve un problema común de diseño. Los patrones de arquitectura o diseño se usan para aprovechar la experiencia de los diseñadores expertos y se logre un buen diseño más rápido que si se empieza desde cero. Un patrón de arquitectura o diseño es una solución a un problema en un contexto. El patrón es el resultado de la experiencia en un dominio específico. Un sistema bien estructurado utiliza patrones para aprovechar la experiencia de los expertos [Gamma et. al, 2003].

En ingeniería de software un patrón se refiere a la forma en la que un determinado problema de análisis, diseño, arquitectura, implementación, etc. Fue solucionado, y cómo reutilizar la esencia de esta solución para resolver nuevos problemas. Un patrón encierra una experiencia, un conocimiento en la solución de problemas previos

Un patrón:

 Maneja un problema de diseño (diseño arquitectónico o diseño de componentes) recurrente que aparece en situaciones de diseño específicas, presentando una solución a éste.

 Documenta experiencia de diseño existente que ha demostrado funcionar bien.

 Identifica y especifica abstracciones que están por encima del nivel de simples clases o instancias, o de componentes.

Además, un patrón proporciona un vocabulario y comprensión comunes para principios de diseño, es un medio para documentar arquitecturas de software, ayuda a construir software complejo a gran escala, y ayuda a manejar la complejidad del software.

En ingeniería de software un patrón es presentado como un esquema (frame) o descripción que consta de tres partes (ver Figura 5-17):

 Contexto: la situación donde surge el problema.

 Problema: descripción del problema que aparece repetidamente en el contexto dado.

 Solución: muestra una solución ya probada para el problema.

Existe una gran variedad de patrones de arquitectura y diseño, los cuales se estudian en cursos más avanzados. En este curso explicaremos tres: ModeloVista-Controlador (patrón arquitectónico), fachada y singleton (patrones de diseño).

UNIDAD IV MODELO DE IMPLEMENTACIÓN

4.1 Modelos de implementación

El Modelo de Implementación es comprendido por un conjunto de componentes y subsistemas que constituyen la composición física de la implementación del sistema. Entre los componentes podemos encontrar datos, archivos, ejecutables, código fuente y los directorios. Fundamentalmente, se describe la relación que existe desde los paquetes y clases del modelo de diseño a subsistemas y componentes físicos.

Un diagrama de implementación muestra:

- Las dependencias entre las partes de código del sistema (diagramas de componentes).
- La estructura del sistema en ejecución (diagrama de despliegue).

4.2DIAGRAMAS DE COMPONENTES

Un componente es una parte física de un sistema (modulo, base de datos, programa ejecutable, etc.). Se puede decir que un componente es la materialización de una o más clases, porque una abstracción con atributos y métodos pueden ser implementados en los componentes.

Respecto a los componentes…

- Es implementado por una o más clases/objetos del sistema.
- Es una unidad autónoma que provee una o más interfaces.
- Las interfaces representan un contrato de servicios que el componente ofrece.

Los componentes pueden ser:

- Archivos
- Código fuente + Cabeceras

- Librerías compartidas (DLLs)
- Ejecutables
- Paquetes

Muestra como el sistema está dividido en componentes y las dependencias entre ellos.

- Proveen una vista arquitectónica de alto nivel del sistema.
- Ayuda a los desarrolladores a visualizar el camino de la implementación.
- Permite tomar decisiones respecto a las tareas de implementación y los Skills requeridos.

En un DC, un componente se representa con un rectángulo en el que se escribe su nombre y en él se muestran dos pequeños rectángulos al lado izquierdo. O también los siguientes:

Representación simple de un Componente

4.3 ELEMENTOS DEL DIAGRAMA DE COMPONENTES

Normalmente los diagramas de Componentes contienen:

- Componentes
- Interfaces
- Relaciones de dependencia, generalización, asociación y realización
- Paquetes o subsistemas

Los componentes se pueden agrupar en paquetes, así como los objetos en clases, además puede haber entre ellos relaciones de dependencia como:

Generalización

- Asociación
- Agregación
- Realización

Estereotipos de componentes

UML define cinco estereotipos estándar que se aplican en los componentes

- Executable, componente que se puede ejecutar
- Library, biblioteca de objetos estática o dinámica
- Table, Componentes que representa una tabla de base de datos
- File, componente que representa un documento que contiene código fuente o datos.
- Document, Comp. Que representa un documento.

¿Por qué utilizar un Diagrama de Componentes?

- Nos permite ver el modelado de un sistema o subsistema
- Permite especificar un componente con interfaces bien definidas.

4.4 DIAGRAMAS DE DESPLIEGUE

El Diagrama de Despliegue es un diagrama que se utiliza para modelar el hardware utilizado en las implementaciones de sistemas y las relaciones entre sus componentes.

- Permiten modelar la disposición física o topología de un sistema.
- Muestra las conexiones físicas entre el hardware y las relaciones entre componentes. Usos que se les da a los diagramas de despliegue son para modelar:
- Sistemas cliente-servidor

- Sistemas completamente distribuidos

El elemento principal del diagrama son los NODOS. Los nodos representan un recurso físico:

- Computadoras
- Sensores
- Impresoras
- Servidores
- Dispositivos externos

Los nodos pueden ser interconectados mediante líneas para describir una estructura de red. Un nodo es un objeto físico en tiempo de ejecución que representa un recurso computacional, generalmente con memoria y capacidad de procesamiento.

Estereotipo de nodo

- Estereotipo, son cosas u objetos q se repiten sin variación.
- El estereotipo de un nodo es la manera de poder verificar que tipo de nodo es el que se está observando.

Artefactos

Un artefacto es un producto del proceso de desarrollo de software, que puede incluir los modelos del proceso (modelos de Casos de Uso, modelos de Diseño, etc.), archivos fuente, ejecutables, documentos de diseño, reportes de prueba, prototipos, manuales de usuario etc.

- Donde un artefacto es un conjunto de componentes. Ejemplo Grafico

Un artefacto se denota por un rectángulo mostrando el nombre del artefacto, el estereotipo «artifact» y un icono de documento, como a continuación.

4.5 MODELOS DE PRUEBA

Objetivos de las pruebas

- Encontrar defectos en el software
- Una prueba tiene éxito si descubre un defecto
- Una prueba fracasa si hay defectos, pero no los descubre
- Pruebas de Verificación
- Pruebas de Validación

El proceso de pruebas del software tiene dos objetivos:

- Demostrar al desarrollador y al cliente que el software satisface sus requerimientos.
- Descubrir defectos en el software: que su comportamiento es incorrecto, no o no cumple su especificación.

PRUEBAS DE "CAJA BLANCA"

- Pruebas en que se conoce el código a probar
- Caja blanca (clear box: caja clara o transparente)
- Se procura ejercitar cada elemento del código
- Algunas clases de pruebas
- Pruebas de cubrimiento
- Pruebas de condiciones
- Pruebas de bucles

PRUEBAS DE "CAJA NEGRA"

- Pruebas en que se conoce sólo la interfaz
- Caja negra (black box: caja opaca)
- Se procura ejercitar cada elemento de la interfaz

- Algunas clases de pruebas
- Cubrimiento. Invocar todas las funciones (100%)
- Pruebas de valores límite

ESTRATEGIAS DE PRUEBA DEL SOFTWARE

- Pruebas de unidades
- Pruebas de integración
- Pruebas de regresión
- Pruebas de validación

PRUEBAS DE UNIDADES:

- Se concentra en el esfuerzo de verificación de la unidad más pequeña del diseño del software: el componente o módulo del software.
- Las pruebas de unidad se concentran en la lógica del procesamiento interno.
- Este tipo de prueba se puede aplicar en paralelo a varios componentes.

PRUEBAS DE INTEGRACIÓN:

- La prueba de integración es una técnica sistemática para construir la arquitectura del software, mientras, al mismo tiempo, se aplican las pruebas para descubrir errores asociados con la interfaz.
- El objetivo es tomar componentes a los que se aplicó una prueba de unidad y construir una estructura de programa que determine el diseño.

PRUEBAS DE REGRESIÓN:

- La prueba de integración es una técnica sistemática para construir la arquitectura del software, mientras, al mismo tiempo, se aplican las pruebas para descubrir errores asociados con la interfaz.

- El objetivo es tomar componentes a los que se aplicó una prueba de unidad y construir una estructura de programa que determine el diseño.

PRUEBAS DE VALIDACIÓN:

- Las pruebas de validación empiezan tras la culminación de la prueba de integración, cuando se han ejercitado los componentes individuales. Se ha terminado de ensamblar el software como paquete y se han descubierto y corregido los errores de interfaz.

- La prueba se concentra en las acciones visibles para el usuario y en la salida del sistema que éste puede reconocer.

4.6 IMPLEMENTACIÓN EN JAVA DE LOS DIAGRAMAS DE CLASE

Cuando se va a construir un sistema software es necesario conocer un lenguaje de programación, pero con eso no basta. Si se quiere que el sistema sea robusto y sustentable, es necesario que el problema sea analizado y la solución sea cuidadosamente diseñada. Se debe seguir un proceso robusto, que incluya las actividades principales. Si se sigue un proceso de desarrollo que se ocupa de plantear cómo se realiza el análisis y el diseño, y cómo se relacionan los productos de ambos, entonces la construcción de sistemas software va a poder planificar y repetible. Y la probabilidad de obtener un sistema de mejor calidad al final del proceso aumenta considerablemente, especialmente cuando se trata de un equipo de desarrollo formado por varias personas.

La notación que se usa para los distintos modelos, tal y como se ha dicho anteriormente, es la proporcionada por UML, que se ha convertido en el estándar de facto en cuanto a notación orientada a objetos. El uso de UML permite integrar con mayor facilidad en el equipo de desarrollo a nuevos miembros y compartir con otros equipos la documentación, pues es de esperar que cualquier desarrollador versado en orientación a objetos conozca y use UML (o se esté planteando su uso).

Se va a abarcar todo el ciclo de vida, empezando por los requisitos y acabando en el sistema funcionando, proporcionando así una visión completa y coherente de la producción de sistemas software. El enfoque que toma es el de un ciclo de vida iterativo incremental, el cual permite una gran flexibilidad a la hora de adaptarlo a un proyecto y a un equipo de desarrollo específicos. El ciclo de vida está dirigido por casos de uso, es decir, por la funcionalidad que ofrece el sistema a los futuros usuarios del mismo. Así no se pierde de vista la motivación principal que debería estar en cualquier proceso de construcción de software: el resolver una necesidad del usuario/cliente.

Podemos interpretar que una clase es el plano que describe como es un objeto de la clase, por tanto, podemos entender que a partir de la clase podemos fabricar objetos. A ese objeto construido se le denomina instancia, y al proceso de construir un objeto se le llama instanciación.

Cuando se construye un objeto es necesario dar un valor inicial a sus atributos, es por ello que existe un método especial en cada clase, llamado constructor, que es ejecutado de forma automática cada vez que es instanciada una variable. Generalmente el constructor se llama igual que la clase y no devuelve ningún valor. Análogamente, destructor es un método perteneciente a una clase que es ejecutado de forma automática cuando un objeto es destruido. Java no soporta los destructores. Es posible que exista más de un constructor en una clase, diferenciados sólo en los parámetros que recibe, pero en la instanciación sólo será utilizado uno de los constructores.

4.7 Constructores y destructores declaración, uso y aplicaciones

Para crear un objeto se necesita reservar suficiente espacio en memoria e inicializar los valores de los campos que representan el estado del objeto. Este trabajo es realizado por un tipo especial de método denominado constructor.

Constructor

Un método constructor de una clase es un método especial que: tiene el mismo nombre que la clase y no tiene tipo de retorno. La sintaxis para la declaración de un método constructor es:

```
[atributos] [modificadores] <identificador> ( [parámetros] ) [inicializador]
{
// Cuerpo del constructor.
}
```

Dónde: atributos (opcional) es información declarativa adicional, modificadores (opcional) se restringen a extern y a los modificadores de acceso, identificador es el nombre del método constructor (igual al nombre de la clase), parámetros (opcional) es la lista de parámetros pasados al constructor, inicializador (opcional). Con el inicializador, el constructor invoca previamente a otro constructor.

El inicializador puede ser uno de los siguientes: base([listaDeParámetros])

this([listaDeParámetros]).

Cuerpo del constructor es el bloque de programa que contiene las instrucciones para inicializar la instancia de clase (objeto).

Destructor

La sintaxis para declarar un destructor es: [Atributos] ~ <Identificador> ( )

{

// Cuerpo del destructor.

}

Una clase solamente puede tener un destructor. Los destructores no pueden heredarse o sobrecargarse, los destructores no pueden invocarse, sino que son invocados automáticamente, un destructor no acepta modificadores ni parámetros, por ejemplo, la siguiente es una declaración de un destructor para la clase Figura:

{

// Instrucciones para limpiar.

}

La destrucción por defecto: Recogida de basura

El intérprete de Java posee un sistema de recogida de basura, que por lo general permite que no nos preocupemos de liberar la memoria asignada explícitamente.

El recolector de basura será el encargado de liberar una zona de memoria dinámica que había sido reservada mediante el operador new, cuando el objeto ya no va a ser utilizado más durante el programa (por ejemplo, sale del ámbito de utilización, o no es referenciado nuevamente).

El sistema de recogida de basura se ejecuta periódicamente, buscando objetos que ya no estén referenciados.

La destrucción personalizada: finalizó

A veces una clase mantiene un recurso que no es de Java como un descriptor de archivo o un tipo de letra del sistema de ventanas. En este caso sería acertado el utilizar la finalización explícita, para asegurar que dicho recurso se libera. Esto se hace mediante la destrucción personalizada, un sistema similar a los destructores de C++.

Para especificar una destrucción personalizada se añade un método a la clase con el nombre finalizó.

4.8 REALIZACIÓN DE LOS CASOS DE USO Y DIAGRAMAS DE INTERACCIÓN

- La vista de casos de uso captura la funcionalidad de un sistema, de un subsistema, o de una clase, tal como se muestra a un usuario exterior
- Reparte la funcionalidad del sistema en transacciones significativas para los usuarios ideales de un sistema
- Los usuarios del sistema se denominan actores y las particiones funcionales se conocen con el nombre de casos de uso
- La técnica que se utiliza para modelar esta vista es el diagrama de casos de uso

CARACTERÍSTICAS

Los casos de uso son una técnica para la especificación de requisitos funcionales propuesta inicialmente por Ivar Jacobson [Jacobson, 1987], [Jacobson et al. 1992] e incorporada a UML Modela la funcionalidad del sistema tal como la perciben los agentes externos, denominados actores, que interactúan con el sistema desde un punto de vista particular.

Sus componentes principales son:

- Sujeto: sistema que se modela
- Casos de uso: unidades funcionales completas

• Actores: entidades externas que interactúan con el sistema

El sujeto se muestra como una caja negra que proporciona los casos de uso.

El modelo de casos de uso se representa mediante los diagramas de casos de uso.

Los casos de uso pueden tener asociaciones y dependencias con otros clasificadores. Relación entre actores y casos de uso:

Asociación

Relaciones entre casos de uso;

• Generalización: Un caso de uso también se puede especializar en uno o más casos de uso hijos.

• Inclusión: Un caso de uso puede incorporar el comportamiento de otros casos de uso como fragmentos de su propio comportamiento.

• Extensión: Un caso de uso también se puede definir como una extensión incremental de un caso de uso base.

Relación entre un caso de uso y una colaboración

• Realización

4.9 ARQUITECTURA LÓGICA CON PATRONES: SEPARACIÓN MODELO- VISTA.

Modelo-vista-controlador (MVC) es un patrón de arquitectura de software, que separa los datos y la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario.

Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

DESCRIPCIÓN DEL PATRÓN

De manera genérica, los componentes de MVC se podrían definir como sigue:

- El Modelo: Es la representación de la información con la cual el sistema opera, por lo tanto, gestiona todos los accesos a dicha información, tantas consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio). Envía a la 'vista' aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'.

- El Controlador: Responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una base de datos). También puede enviar comandos a su 'vista' asociada si se solicita un cambio en la forma en que se presenta el 'modelo' (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos), por tanto se podría decir que el 'controlador' hace de intermediario entre la 'vista' y el 'modelo' (véase Middleware).

- La Vista: Presenta el 'modelo' (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario), por tanto requiere de dicho 'modelo' la información que debe representar como salida.

INTERACCIÓN DE LOS COMPONENTES

Aunque se pueden encontrar diferentes implementaciones de MVC, el flujo de control que se sigue generalmente es el siguiente:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.).

2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler) o callback.

3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la

compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.

4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista. Sin embargo, se podría utilizar el patrón Observador para proveer cierta indirección entre el modelo y la vista, permitiendo al modelo notificar a los interesados de cualquier cambio. Un objeto vista puede registrarse con el modelo y esperar a los cambios, pero aun así el modelo en sí mismo sigue sin saber nada de la vista. Este uso del patrón Observador no es posible en las aplicaciones Web puesto que las clases de la vista están desconectadas del modelo y del controlador. En general el controlador no pasa objetos de dominio (el modelo) a la vista aunque puede dar la orden a la vista para que se actualice. Nota: En algunas implementaciones la vista no tiene acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista. Por ejemplo, en el MVC usado por Apple en su framework Cocoa. Suele citarse como Modelo-Interface-Control, una variación del MVC más puro

5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente...

.10 MVC Y BASES DE DATOS

Muchos sistemas informáticos utilizan un Sistema de Gestión de Base de Datos el cual gestiona los datos que debe utilizar la aplicación; en líneas generales del MVC dicha gestión corresponde al modelo. La unión entre capa de presentación y capa de negocio conocido en el paradigma de la Programación por capas representaría la integración entre la Vista y su correspondiente Controlador de eventos y acceso a datos, MVC no pretende discriminar entre capa de negocio y capa de presentación pero si pretende separar la capa visual gráfica de su correspondiente programación y acceso a datos, algo que mejora el desarrollo y mantenimiento de la Vista y el Controlador en paralelo, ya que ambos cumplen ciclos de vida muy distintos entre sí.

4.11 USO EN APLICACIONES WEB

Aunque originalmente MVC fue desarrollado para aplicaciones de escritorio, ha sido ampliamente adaptado como arquitectura para diseñar e implementar aplicaciones weben los principales lenguajes de programación. Se han desarrollado multitud de frameworks, comerciales y no comerciales, que implementan este patrón (ver apartado siguiente "Frameworks MVC"); estos frameworks se diferencian básicamente en la interpretación de como las funciones MVC se dividen entre cliente y servidor.

Los primeros frameworks MVC para desarrollo web planteaban un enfoque de cliente ligero en el que casi todas las funciones, tanto de la vista, el modelo y el controlador recaían en el servidor. En este enfoque, el cliente manda la petición de cualquier hiperenlace o formulario al controlador y después recibe de la vista una página completa y actualizada (u otro documento); tanto el modelo como el controlador (y buena parte de la vista) están completamente alojados en el servidor. Como las tecnologías web han madurado, ahora existen frameworks como JavaScriptMVC, Backbone o jQuery que permiten que ciertos componentes MVC se ejecuten parcial o totalmente en el cliente.

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.).

2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler) o callback.

3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.

4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista. Sin embargo, se podría utilizar el patrón Observador para proveer cierta indirección entre el modelo y la vista, permitiendo al modelo notificar a los interesados de cualquier cambio. Un objeto vista puede registrarse con el modelo y esperar a los cambios, pero aun así el modelo en sí mismo sigue sin

saber nada de la vista. El controlador no pasa objetos de dominio (el modelo) a la vista aunque puede dar la orden a la vista para que se actualice. Nota: En algunas implementaciones la vista no tiene acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista.

5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

BIBLIOGRAFÍA BÁSICA Y COMPLEMENTARIA:

- TutorialsPoint; aprender la ingeniería de software, principios, procesos y procedimiento; Enlace:https://www.tutorialspoint.com/es/software_engineering/software_engineering_overview.htm

- Ingeniería de software; Reseña del Modelo de Prototipo y Herramientas Case realizado por Breton J. García G. y Rojas I.; mayo 2011; Enlace: http://gestionrrhhusm.blogspot.com/2011/05/modelo-de-prototipo.html

- Modelos evolutivos; Mayo del 2013, Enlace: http://cuartomodelo.blogspot.com/

- Evolución y Perspectivas de la Ingeniería del Software, Enlace: https://www.cimat.mx/~clemola/Ponencias/utng.ppt

- Gómez Fuentes María del Carmen; Análisis de requerimientos; Universidad Autónoma Metropolitana; Departamento de Matemáticas Aplicadas y sistemas; Primera edición 2011.

- Kendall, K y Kendall, J. 2011. Análisis y diseño de sistemas. 8 ed. México. Pearson Educación. p 600

- http://es.scribd.com/doc/7884665/Arquitectura-de-Software-II-Diagrama-de- Componentes-y-Despliegue

- http://www.slideshare.net/techmi/curso-uml-25-diagramas-de-implementacion