NOMBRE DEL PROFESOR:

Juan José Ojeda Trujillo

NOMBRE DEL ALMUNO:

Marlong Uriel ramos dominguez

MATERIA:

Ingeniería en software

CARRERA:

Ingeniería en sistemas computacionales

CUATRIMESTRE:

8

TEMA DE LA ACTIVIDAD:

Ensayo de la unidad I y II

27/02/24

Unidad I

FUNDAMENTOS 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.

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 UNIVERSIDAD DEL SURESTE 12 de programación, asociada a las bibliotecas y a la documentación. El Software, cuando se ha hecho para cubrir requisitos específicos se llama producto software.

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 eficiente en máquinas reales de forma económica.

- ## 1.2. CARACTERISITICAS Y APLICACIONES DEL SOFTWARE

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.

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.

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).

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.

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

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 UNIVERSIDAD DEL SURESTE 24 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.

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

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.

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 UNIVERSIDAD DEL SURESTE 28 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 UNIVERSIDAD DEL SURESTE 29 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".

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?

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?

2.3 IDENTIFICACIÓN, ANÁLISIS, NEGOCIACIÓN Métodos generales de entrevistas.

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

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

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.

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.

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

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.

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.

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.

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.

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 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. UNIVERSIDAD DEL SURESTE 56 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.

FUENTES DE INFORMACION

- https://plataformaeducativauds.com.mx/assets/docs/libro/ISC/c4bbf47ef7521fe5a5df5241378572b2-LC-ISC804%20INGENIERIA%20EN%20SOFTWARE.pdf