Proceso de Software

Un proceso de software es una serie de actividades relacionadas que conduce a la elaboración de un producto de software. Estas actividades pueden incluir el desarrollo de software desde cero en un lenguaje de programación estándar como Java o C. Sin embargo, las aplicaciones de negocios no se desarrollan precisamente de esta forma. El nuevo software empresarial con frecuencia ahora se desarrolla extendiendo y modificando los sistemas existentes, o configurando e integrando el software comercial o componentes del sistema.

Existen muchos diferentes procesos de software, pero todos deben incluir cuatro actividades que son fundamentales para la ingeniería de software:

- **Especificación del software:** Tienen que definirse tanto la funcionalidad del software como las restricciones de su operación.
- **Diseño e implementación del software:** Debe desarrollarse el software para cumplir con las especificaciones.
- **Validación del software:** Hay que validar el software para asegurarse de que cumple lo que el cliente quiere.
- **Evolución del software:** El software tiene que evolucionar para satisfacer las necesidades cambiantes del cliente.

En cierta forma, tales actividades forman parte de todos los procesos de software. Por supuesto, en la práctica éstas son actividades complejas en sí mismas e incluyen subactividades tales como la validación de requerimientos, el diseño arquitectónico, la prueba de unidad, etcétera. También existen actividades de soporte al proceso, como la documentación y el manejo de la configuración del software.

Los procesos de software pueden mejorarse con la estandarización de los procesos, donde se reduce la diversidad en los procesos de software en una organización. Esto conduce a mejorar la comunicación, a reducir el tiempo de capacitación, y a que el soporte de los procesos automatizados sea más económico. La estandarización también representa un primer paso importante tanto en la introducción de nuevos métodos y técnicas de ingeniería de software, como en sus buenas prácticas. Herramientas del proceso de Software:

Las herramientas de desarrollo del software (llamadas en ocasiones herramientas de Ingeniería de Software Asistido por Computadora o CASE, por las siglas de Computer- Aided Software Engineering) son programas usados para apoyar las actividades

del proceso de la ingeniería de software. En consecuencia, estas herramientas incluyen editores de diseño, diccionarios de datos, compiladores, depuradores (debuggers), herramientas de construcción de sistema, etcétera.

Las herramientas de software ofrecen apoyo de proceso al automatizar algunas actividades del proceso y brindar información sobre el software que se desarrolla.

Cambios en el Software:

El cambio es inevitable en todos los grandes proyectos de software. Los requerimientos del sistema varían conforme la empresa procura que el sistema responda a presiones externas y se modifican las prioridades administrativas. A medida que se ponen a disposición nuevas tecnologías, surgen nuevas posibilidades de diseño e implementación. Por ende, cualquiera que sea el modelo del proceso de software utilizado, es esencial que ajuste los cambios al software a desarrollar.

Dos formas de enfrentar el cambio y los requerimientos cambiantes del sistema podrían ser:

1. Prototipo de sistema, donde rápidamente se desarrolla una versión del sistema o una parte del mismo, para comprobar los requerimientos del cliente y la factibilidad de algunas decisiones de diseño. Esto apoya el hecho de evitar el cambio, al permitir que los usuarios experimenten con el sistema antes de entregarlo y así refinar sus requerimientos. Como resultado, es probable que se reduzca el número de propuestas de cambio de requerimientos posterior a la entrega.

2. Entrega incremental, donde los incrementos del sistema se entregan al cliente para su comentario y experimentación. Esto apoya tanto al hecho de evitar el cambio como a tolerar el cambio. Por un lado, evita el compromiso prematuro con los requerimientos para todo el sistema y, por otro, permite la incorporación de cambios en incrementos mayores a costos relativamente bajos.

Entrega de Prototipo:

Un prototipo es una versión inicial de un sistema de software que se usa para demostrar conceptos, tratar opciones de diseño y encontrar más sobre el problema y sus posibles soluciones. El rápido desarrollo iterativo del prototipo es esencial, de modo que se controlen los costos, y los interesados en el sistema experimenten por anticipado con el prototipo durante el proceso de software.

Un prototipo de software se usa en un proceso de desarrollo de software para contribuir a anticipar los cambios que se requieran:

En el proceso de ingeniería de requerimientos, un prototipo ayuda con la selección y validación de requerimientos del sistema.

En el proceso de diseño de sistemas, un prototipo sirve para buscar soluciones específicas de software y apoyar el diseño de interfaces del usuario.

Los prototipos del sistema permiten a los usuarios ver qué tan bien el sistema apoya su trabajo. Pueden obtener nuevas ideas para requerimientos y descubrir áreas de fortalezas y debilidades en el software.

Mientras se elabora el sistema para la realización de experimentos de diseño, un prototipo del mismo sirve para comprobar la factibilidad de un diseño propuesto. Asimismo, la creación de prototipos es una parte esencial del proceso de diseño de interfaz del usuario. Debido a la dinámica natural de las interfaces de usuario, las descripciones textuales y los diagramas no son suficientemente buenos para expresar los requerimientos de la interfaz del usuario. Por lo tanto, la creación rápida de prototipos con la participación del usuario final es la única forma sensible para desarrollar interfaces de

usuario gráficas para sistemas de software.

La siguiente etapa del proceso consiste en decidir qué poner y, algo quizá más importante, qué dejar fuera del sistema de prototipo. Para reducir los costos de creación de prototipos y acelerar las fechas de entrega, es posible dejar cierta funcionalidad fuera del prototipo y, también, decidir hacer más flexible los requerimientos no funcionales, como el tiempo de respuesta y la utilización de memoria. El manejo y la gestión de errores pueden ignorarse, a menos que el objetivo del prototipo sea establecer una interfaz de usuario. Además, es posible reducir los estándares de fiabilidad y calidad del programa.

La etapa final del proceso es la evaluación del prototipo. Hay que tomar provisiones durante esta etapa para la capacitación del usuario y usar los objetivos del prototipo para derivar un plan de evaluación. Los usuarios requieren tiempo para sentirse cómodos con un nuevo sistema e integrarse a un patrón normal de uso. Una vez que utilizan el sistema de manera normal, descubren errores y omisiones en los requerimientos.

Los prototipos no tienen que ser ejecutables para ser útiles. Los modelos en papel de la interfaz de usuario del sistema pueden ser efectivos para ayudar a los usuarios a refinar un diseño de interfaz y trabajar a través de escenarios de uso. Su desarrollo es muy económico y suelen construirse en pocos días.

El prototipado es una técnica que apunta a conseguir los requerimientos del sistema.

Debe aplicarse en etapas tempranas del ciclo de vida. Sirve para validar requerimientos. Corta proceso de propagación de errores. Requiere intervención activa de los usuarios. Carácter cíclico de la retroalimentación. Permite detectar posibles riesgos del proceso.

Sus objetivos:

Reacciones iniciales de los usuarios Nuevas necesidades. Requerimientos incompletos, ambiguos o malentendidos. Preferencias de los usuarios. Actitudes de los usuarios Innovaciones. Nuevas capacidades. Identificar prioridades de los requerimientos Redirigir el plan según prioridades.

Tipos/Clasificación de Prototipos:

Parchado:

- Tiene parches
- Operacional
- Funcionalidad recortada. Ineficiente en sus funciones
- Usuarios interactúan con ellos
- Revisa la E/S
- No tiene lógica interna
- Código escrito sin calidad

No operacional:

- No corre en hardware
- Pueden ser listados, visiones de E/S

- Maqueta
- Esquemas, gráficos, bosquejos

Primero de una serie:

- Proyecto piloto
- Primero para variar instalaciones
- Útil para varias sucursales

De características seleccionadas:

- Operacional
- Incluye solo algunas operaciones
- Armado de estructuras de menús y submenús
- Construcción de módulos

Evolutivo:

- Los procesos de especificación, diseño e implementación se entrelazan
- Se hace en el lenguaje definitivo
- Utiliza las normas de calidad definidas para el proceso de producción
- El sistema se va a liberar luego de una serie de incrementos
- Las interfaces se realizan con un desarrollador interactivo
    - Generalmente no existe una SRS formal o es difícil establecer un contrato a partir de ella
    - En sistemas pequeños es más adecuado liberar funcionalidad rápidamente
    - En sistemas grandes requiere mayor coordinación si hubiera varios contratistas y donde los productos se construyen con las SRS

Incremental:

- Hay una SRS de todo el sistema
- Se define el incremento

- Se va construyendo por módulos que van apilando la funcionalidad

Desechable:

- No se usa para validar el diseño
- Objetivo es clarificar requerimientos
- El diseño puede diferir del producto final
- La aplicación de estándares de calidad es relajado o nulo
- El lenguaje puede ser diferente al del producto final
- Solo se desarrolla la funcionalidad crítica
- No se valoran los aspectos de requerimientos no funcionales
- Asegurarse que el usuario no lo adopte como real
- No hay documentación
- Se presentan estructuras

Entrega Incremental:

La entrega incremental es un enfoque al desarrollo de software donde algunos de los incrementos diseñados se entregan al cliente y se implementan para usarse en un entorno operacional. En un proceso de entrega incremental, los clientes identifican, en un bosquejo, los servicios que proporciona el sistema. Identifican cuáles servicios son más importantes y cuáles son menos significativos para ellos. Entonces, se define un número de incrementos de entrega, y cada incremento proporciona un subconjunto de la funcionalidad del sistema. La asignación de servicios por incrementos depende de la prioridad del servicio, donde los servicios de más alta prioridad se implementan y entregan primero.

El Proceso Unificado Racional (RUP):

El Proceso Unificado Racional (RUP, por las siglas de Rational Unified Process) es un ejemplo de un modelo de proceso moderno que se derivó del trabajo sobre el UML y el proceso asociado de desarrollo de software unificado. Es un buen ejemplo de un modelo de proceso híbrido. Conjunta elementos de todos los modelos de proceso genéricos, ilustra la buena práctica en especificación y diseño, y apoya la creación de prototipos y entrega incremental.

RUP es una secuencia de pasos necesarios para el desarrollo y/o

mantenimiento de gran cantidad de sistemas, en diferentes áreas de aplicación diferentes organizaciones, diferentes medios de competencia y en proyectos de tamaños variables (desde el mas básico al mas complejo). Actualmente es propiedad de International Business Machines (IBM) y esta basado en un enfoque disciplinado de asignación de tareas y responsabilidades dentro de una organización de desarrollo con la finalidad de asegurar la

obtención de un software de alta calidad que satisfagan la necesidad de los usuarios finales dentro de un calendario y tiempo predecible.

Elementos de RUP

Disciplinas:

- Son los 'contenedores' empleados para organizar todas las actividades durante el ciclo de vida del sistema.

Artefactos:

- Son los elementos de entrada y salida de las actividades. Es un elemento que el proyecto produce y utiliza para componer el producto final.

Flujos de Trabajo:

- Constituye la secuencia de actividades que producen resultados visibles por medio de la integración de los roles y las actividades, artefactos y disciplinas.

Roles: - Son las personas o entes que están involucradas en cada proceso

El RUP reconoce que los modelos de proceso convencionales presentan una sola visión del proceso. En contraste, el RUP por lo general se describe desde tres perspectivas:

- Una perspectiva dinámica que muestra las fases del modelo a través del tiempo. - Una perspectiva estática que presenta las actividades del proceso que se establecen.

- Una perspectiva práctica que sugiere buenas prácticas a usar durante el proceso.

En una visión dinámica, el RUP es un modelo en fases que identifica cuatro fases discretas en el proceso de software. Sin embargo, a diferencia del modelo en cascada, donde las fases se igualan con actividades del proceso, las fases en el RUP

están más estrechamente vinculadas con la empresa que con las preocupaciones técnicas.

- **Concepción:**

- La meta de la fase de concepción es establecer un caso empresarial para el sistema. Deben identificarse todas las entidades externas (personas y sistemas) que interactuarán con el sistema y definirán dichas interacciones. Luego se usa esta información para valorar la aportación del sistema hacia la empresa. Si esta aportación es menor, entonces el proyecto puede cancelarse después de esta fase.

- **Elaboración:**

- Las metas de la fase de elaboración consisten en desarrollar la comprensión del problema de dominio, establecer un marco conceptual arquitectónico para el sistema, diseñar el plan del proyecto e identificar los riesgos clave del proyecto. Al completar esta fase, debe tenerse un modelo de requerimientos para el sistema, que podría ser una serie de casos de uso del UML, una descripción arquitectónica y un plan de desarrollo para el software.

- **Construcción:**

- La fase de construcción incluye diseño, programación y pruebas del sistema. Partes del sistema se desarrollan en paralelo y se integran durante esta fase. Al completar ésta, debe tenerse un sistema de software funcionando y la documentación relacionada y lista para entregarse al usuario.

- **Transición:**

- La fase final del RUP se interesa por el cambio del sistema desde la comunidad de desarrollo hacia la comunidad de usuarios, y por ponerlo a funcionar en un ambiente real. Esto es algo ignorado en la mayoría de los modelos de proceso de software aunque, en efecto, es una actividad costosa y en ocasiones problemática. En el complemento de esta fase se debe tener un sistema de software documentado que funcione correctamente en su entorno operacional.

La visión estática del RUP se enfoca en las actividades que tienen lugar durante el proceso de desarrollo. Se les llama flujos de trabajo en la descripción RUP. En el proceso se identifican seis flujos de trabajo de proceso centrales y tres flujos de trabajo de apoyo centrales. El RUP se diseñó en conjunto con el UML, de manera que la descripción

del flujo de trabajo se orienta sobre modelos UML asociados, como modelos de secuencia, modelos de objeto, etcétera.

El enfoque práctico del RUP describe las buenas prácticas de ingeniería de software que se recomiendan para su uso en el desarrollo de sistemas. Las seis mejores prácticas fundamentales que se recomiendan son:

- Desarrollo de software de manera iterativa: Incrementar el plan del sistema con base en las prioridades del cliente, y desarrollar oportunamente las características del sistema de mayor prioridad en el proceso de desarrollo.

- **Gestión de requerimientos:** Documentar de manera explícita los requerimientos del cliente y seguir la huella de los cambios a dichos requerimientos. Analizar el efecto de los cambios sobre el sistema antes de aceptarlos.

- **Usar arquitecturas basadas en componentes:** Estructurar la arquitectura del sistema en componentes.

- **Software modelado visualmente:** Usar modelos UML gráficos para elaborar representaciones de software estáticas y dinámicas.

- **Verificar la calidad del software:** Garantizar que el software cumpla con los estándares de calidad de la organización.

- **Controlar los cambios al software:** Gestionar los cambios al software con un sistema de administración del cambio, así como con procedimientos y herramientas de administración de la configuración.

Las innovaciones más importantes en el RUP son la separación de fases y flujos de trabajo, y el reconocimiento de que el despliegue del software en un entorno del usuario forma parte del proceso. Las fases son dinámicas y tienen metas. Los flujos de trabajo son estáticos y son actividades técnicas que no se asocian con una sola fase, sino que pueden usarse a lo largo del desarrollo para lograr las metas de cada fase.

Se dice que el RUP es un proceso iterativo porque es conveniente separar el trabajo en partes más pequeñas o mini proyectos. Cada mini proyecto es un iteración. Las iteraciones hacen referencia a los flujos de trabajo. Las iteraciones deben estar controladas, esto es, deben seleccionarse y ejecutarse de una forma planificada.

Disciplinas del RUP:

- Modelado de negocios - Requerimientos - Análisis y diseño - Implementación - Pruebas - Transición - Configuración y administración del cambio - Administrador de proyectos - Ambiente

Modelado de negocios:

- Entender los problemas que la organización desea solucionar e identificar mejoras potenciales.

- Medir el impacto del cambio organizacional. - Asegurar que clientes, usuarios finales, desarrolladores y los otros participantes tengan un entendimiento compartido del problema.

- Derivar los requerimientos del sistema de software, necesarios para dar soporte a los objetivos de la organización.

- Entender como el sistema a ser desarrollado entra dentro de la organización.

Requerimientos:

- Establecer y mantener un acuerdo con los clientes y los otros interesados. - Proveer a los desarrolladores del sistema de un mejor entendimiento de los requerimientos del sistema.

- Definir los límites (o delimitar) del sistema. - Proveer una base para la planeación de los contenidos técnicos de las iteraciones.

- Proveer una base para la estimación de costo y tiempo necesarios para desarrollar el sistema.

- Definir una interfaz de usuario para el sistema, enfocada en las necesidades y objetivos del usuario.

Análisis y diseño:

- Transformar los requerimientos a diseños del sistema. - Desarrollar una arquitectura robusta para el sistema. - Adaptar el diseño para hacerlo corresponder el ambiente de implementación y ajustarla para un desempeño esperado.

Implementación:

- Definir la organización del código, en términos de la implementación de los subsistemas organizados en capas.

- Implementar el diseño de elementos en términos de los elementos (archivos fuente, binarios, ejecutables y otros).

- Probar los componentes desarrollados como unidades. - Integrar los resultados individuales en un sistema ejecutable.

Pruebas:

- Actúa como un proveedor de servicios a las otras disciplinas en muchos aspectos. Se enfoca principalmente en la evaluación y aseguramiento de la calidad del producto, desarrollado a través de las siguientes prácticas:

- Encontrar fallas de calidad en el software y documentarlas. - Recomendar sobre la calidad percibida en el software. - Validar y probar las suposiciones hechas durante el diseño y la especificación de requerimientos de forma concreta.

- Validar que el software trabaja como fue diseñado. - Validar que los requerimientos son implementados apropiadamente.

Transición:

- Esta disciplina describe las actividades asociadas con el aseguramiento de la entrega y disponibilidad del producto de software hacia el usuario final.

Configuración y administración del cambio:

- Consiste en controlar los cambios y mantener la integridad de los productos que incluye el proyecto. Incluye:

- Identificar los elementos configurables. - Restringir los cambios en los elementos configurables. - Auditar los cambios hechos a estos elementos. - Definir y mantener las configuraciones de estos elementos.

Administración de proyectos:

- El propósito de la Administración de Proyectos es: - Proveer un marco de trabajo para administrar los proyectos intensivos de software.

- Proveer guías prácticas para la planeación, soporte, ejecución y monitoreo de proyectos. - Proveer un marco de trabajo para la administración del

riesgo.

Ambiente:

- Se enfoca en las actividades necesarias para configurar el proceso al proyecto. Describe las actividades requeridas para desarrollar las líneas guías de apoyo al proyecto. El propósito de las actividades de ambiente es proveer a las organizaciones de desarrollo de software del ambiente necesario (herramientas y procesos) que den soporte al equipo de desarrollo.

Requerimientos:

- Los requerimientos para un sistema son descripciones de lo que el sistema debe hacer: el servicio que ofrece y las restricciones en su operación. Tales requerimientos reflejan las necesidades de los clientes por un sistema que atienda cierto propósito, como sería controlar un dispositivo, colocar un pedido o buscar información. Al proceso de descubrir, analizar, documentar y verificar estos servicios y restricciones se le llama ingeniería de requerimientos (IR).

- Para entender los requerimientos se necesita, en parte, conocer los procesos de dominio y ambiente externo, es decir, los factores externos que participan en los procesos. Dichos procesos de dominio pueden expresarse en casos de uso, descripciones narrativas de los procesos del dominio en un formato estructurado de prosa. Los casos de uso no son propiamente un elemento de análisis orientado a objetos; se limitan a describir procesos y pueden ser igualmente eficaces en un proyecto de tecnología no orientada a objetos. No obstante, constituyen un paso preliminar muy útil ya que describen las especificaciones de un sistema. Resumidamente, los casos de usos son historias o casos de utilización de un sistema; no son exactamente los requerimientos ni las especificaciones funcionales, sino ejemplifican e incluyen tácitamente los requerimientos en las historias que narran.

Los requerimientos del usuario y los requerimientos del sistema se definen del siguiente modo:

Los requerimientos del usuario son enunciados, en un lenguaje natural junto con diagramas, acerca de qué servicios esperan los usuarios del sistema, y de las restricciones con las cuales éste debe operar.

Los requerimientos del sistema son descripciones más detalladas de las

funciones, los servicios y las restricciones operacionales del sistema de software. El documento de requerimientos del sistema (llamado en ocasiones especificación funcional) tiene que definir con exactitud lo que se implementará. Puede formar parte del contrato entre el comprador del sistema y los desarrolladores del software.

A menudo, los requerimientos del sistema de software se clasifican como requerimientos funcionales o requerimientos no funcionales:

Requerimientos funcionales:

- Son enunciados acerca de servicios que el sistema debe proveer, de cómo debería reaccionar el sistema a entradas particulares y de cómo debería comportarse el sistema en situaciones específicas. En algunos casos, los requerimientos funcionales también explican lo que no debe hacer el sistema.

Requerimientos no funcionales:

- Son limitaciones sobre servicios o funciones que ofrece el sistema. Incluyen restricciones tanto temporales y del proceso de desarrollo, como impuestas por los estándares. Los requerimientos no funcionales se suelen aplicar al sistema como un todo, más que a características o a servicios individuales del sistema.

En realidad, la distinción entre los diferentes tipos de requerimientos no es tan clara como sugieren estas definiciones sencillas. Un requerimiento de un usuario interesado por la seguridad, como el enunciado que limita el acceso a usuarios autorizados, parecería un requerimiento no funcional. Sin embargo, cuando se desarrolla con más detalle, este requerimiento puede generar otros requerimientos que son evidentemente funcionales, como la necesidad de incluir facilidades de autenticación en el sistema.

Proceso de Diseño:

El diseño de software es un proceso iterativo por medio del cual se traducen los requerimientos en un “plano” para construir el software. Al principio, el plano ilustra una visión totalizadora del software. Es decir, el diseño se representa en un nivel alto de abstracción, en el que se rastrea directamente el objetivo específico del sistema y los requerimientos más detallados de datos, funcionamiento y comportamiento. A medida que tienen lugar las iteraciones del diseño, las mejoras posteriores conducen a niveles menores de abstracción. Éstos también pueden rastrearse hasta los requerimientos, pero la conexión es más sutil.

A través del proceso de diseño se evalúa la calidad de éste, debiendo cumplir

con tres características que funcionan como guía para evaluar un buen diseño:

- Debe implementar todos los requerimientos explícitos contenidos en el modelo de requerimientos y dar cabida a todos los requerimientos implícitos que desean los participantes.

- Debe ser una guía legible y comprensible para quienes generan el código y para los que lo prueban y dan el apoyo posterior.

- Debe proporcionar el panorama completo del software, y abordar los dominios de los datos, las funciones y el comportamiento desde el punto de vista de la implementación.

Análisis y Diseño Orientado a Objetos:

La esencia del análisis y el diseño orientados a objetos consiste en situar el dominio de un problema y su solución dentro de la perspectiva de los objetos.

Durante el análisis orientado a objetos se procura ante todo identificar y describir los objetos -o conceptos- dentro del dominio del problema. En una biblioteca, los objetos a identificar posibles podrían ser Libro, Lector, Bibliotecario, Socio, etc. Para descomponer el dominio del problema hay que identificar los conceptos, los atributos y las asociaciones del dominio que se juzgan importantes. El resultado puede expresarse en un modelo conceptual, el cual se muestra gráficamente en un grupo de diagramas que describen los conceptos (objetos).

Durante el diseño orientado a objetos, se procura definir los objetos lógicos del

software que finalmente serán implementados en un lenguaje de programación orientado a objetos. Los objetos tienen atributos y métodos. Así, en el sistema de la biblioteca, un objeto Libro podría tener atributos titulo, autor, estaPrestado, etc; y métodos imprimir, prestarLibro, editarInformación, etc. Un paso esencial de esta fase es la asignación de responsabilidades entre los objetos y mostrar cómo interactúan a través de mensajes, expresados en diagramas de interacción. Éstos presentan el flujo de mensajes entre las instancias y la invocación de métodos, reflejando decisiones en cuanto a la asignación de responsabilidades entre los objetos (operaciones).

Los diagramas de interacción se realizan en la fase de diseño de un cliclo de desarrollo. No se pueden preparar si antes no se generan los siguientes artefactos:

- **Un modelo conceptual:** A partir de este modelo el diseñador podrá

definir las clases del software correspondientes a los conceptos. Los objetos de las clases participan en las interacciones que se describe gráficamente en los diagramas.

- **Contratos de la operación del sistema:** A partir de ellos el diseñador identifica las responsabilidades y las poscondiciones que han de llenar los diagramas de interacción.

Finalmente, durante la construcción o programación orientada a objetos, se implementan los componentes del diseño, como sería una clase Libro en C++, Java, etc.

UML: Diagramas para partes estáticas de un sistema:

1) **Clases:** muestran clases y objetos, junto con sus contenidos y relaciones, es decir el nombre de la clase, los atributos y las operaciones.

2) **Objetos:**

- Se muestran igual que como en una clase. Muestran un “snapshot” del sistema en un momento determinado.

- Un diagrama de objetos modela las instancias de los elementos contenidos en los diagramas de clases.

- Muestra un conjunto de objetos y sus relaciones en un momento concreto. Utiliza una notación muy similar al diagrama de clases.

3) **Componentes:**

- Describe elementos físicos de un sistema y sus relaciones. - Es un elemento de modelado de UML con el cual podemos representar comunicación y colaboración entre los distintos componentes de software de un sistema.

- Los diagramas de componentes son fundamentalmente diagramas de clases que se centran en los componentes de un sistema.

- Se utilizan para modelar la vista de implementación estática de un sistema. Es un modelo estático de UML

- Un componente es un bloque de construcción de software. Es un conjunto de clases que colaboran, con sus atributos, operaciones, interfaces que permiten la comunicación y la colaboración. Es una parte física reemplazable de un sistema que

empaqueta su implementación. Tiene un conjunto de interfaces a las que proporciona su realización.

- Las clases representan abstracciones lógicas, los componentes representan elementos físicos del mundo de los bits. Además, las clases pueden tener atributos y operaciones directamente accesibles. Los componentes tienen operaciones que sólo son accesibles a través de sus interfaces.

4) **Despliegue:** configuración de componentes hardware y distribución de procesos en los mismos. **Diagramas de comportamiento de UML:**

5) **Casos de uso:** ilustra los requerimientos funcionales de un sistema y como se comporta éste ante eventos. 6) **Secuencia:** muestra interacción entre objetos de una aplicación en el tiempo. Es como una película de nuestro sistema planteado en un escenario. 7) **Comunicación:** Muestra como interactúan los objetos en el tiempo, teniendo en cuenta también las relaciones entre ellos. Objetos en formato “red” y sus relaciones. 8) **Estados:** describe todos los estados posibles de un objeto y como cambia según eventos. 9) **Actividades:** Expansión del diagrama de estados, muestra un método, un caso de uso, un workflow, etc.

OCL:

OCL es un lenguaje formal usado para describir expresiones acerca de modelos UML. Las evaluación de expresiones OCL no afectan el estado del sistema en ejecución.

Cuando se diseña un sistema Orientado a Objetos, es necesario especificar un conjunto de restricciones semánticas:

- Utilizar lenguaje natural conduce a ambigüedades. - No a todos les resulta familiar utilizar métodos formales. - UML no cuenta con la expresividad suficiente para modelar muchas restricciones.

Debido a estas razones ha surgido OCL (Object Constraint Language), el cual permite describir restricciones sobre modelos UML, y definir y documentar modelos UML de manera mas precisa.

OCL es un leguaje declarativo, a partir del cual se establece que debe ser hecho por el sistema, y no como debe ser hecho. Se dice además que es un lenguaje

tipado, ya que todas las expresiones tienen un tipo y se requiere conformidad de tipo; y que es un lenguaje de expresión puro, ya que no modifica nada en el modelo: el estado nunca cambiará debido a una expresión OCL.

El vinculo entre una entidad en un diagrama UML y una expresión OCL se denomina definición de contexto de una expresión OCL. La definición de contexto de una expresión OCL especifica la entidad del modelo para la que se define la expresión OCL.

Un tipo contextual de una expresión OCL se corresponde con la entidad del modelo

para la cual se define la expresión OCL. El tipo contextual de una expresión OCL, se escribe luego de la palabra reservada context. Una expresión OCL se evalúa para un único objeto, el cual es siempre una instancia de tipo contextual (instancia contextual).

Una expresión con etiqueta invariante INV es una condición que deberá ser verdadera para todas las instancias del tipo específico, clasificador.

Ingeniería de sistemas:

Enfoque interdisciplinario cuyo objetivo es el estudio de los sistemas de información y a través de ellos comprender la realidad. Aborda la complejidad, modela y representa los mismos con herramientas propias. Aparece por los avances del conocimiento humano (tecnología). Integra teorías, conceptos, principios, técnicas, normativas y procedimientos de las ciencias para resolver problemas.

Centrada en la comprensión de la realidad

Ingeniería de software:

Disciplina que crea métodos de resolución de problemas con el menor costo de recursos, optimizando, y con la capacidad de medir los procesos y construir productos software de calidad.

Además el conocimiento final es mayor que el inicial, esto significa que no sólo se aplicaron pasos o métodos preexistentes sino que se generó conocimiento.

Centrada en la construcción del software

Analista:

- Investiga cómo se toman las decisiones - Indaga los problemas del negocio - Define las necesidades de los usuarios - Comunicación con stakeholders - Captura el conocimiento del

dominio - Descubre circuitos de la información - Interpreta y aprende las costumbres de la organización (cultura) Separa los juicios y subjetividades

- Identifica las fuerzas dentro de la organización - Resuelve conflictos, intereses, mediador - Mitiga impacto de la implantación (stress social) - Elicita los requerimientos

Diseñador:

- Posee base de conocimiento tecnológico - Conoce tecnología disponible - Traza los planos (modelos) - Crea soluciones - Identifica las fortalezas y debilidades de las alternativas de solución - Establece la solución más adecuada a la problemática actual

Cohesión:

- Nivel de cohesión indica la ligazón entre elementos de un grupo o conjunto. Unidad de objetivo. Relacionamiento. Vinculación

- La medida que indica si una clase tiene una función bien definida dentro del sistema.

- El objetivo es enfocar de la forma más precisa posible el propósito de la clase. - Examinar una clase y/o componente, decidir si todo su contenido está directamente relacionado con el nombre de la clase y descrito por el mismo.

- Alta cohesión hace más fácil: Entender qué hace una clase o método, Usar nombres descriptivos, reutilizar clases o métodos, y extender.

Acoplamiento:

- El acoplamiento entre clases es una medida de la interconexión o dependencia entre esas clases.

- El acoplamiento fuerte significa que las clases relacionadas necesitan saber detalles internos unas de otras

- Los cambios se propagan por el sistema y el sistema es posiblemente más difícil de entender.

- Siempre intentar que nuestras clases tengan un acoplamiento bajo.

DISEÑO DE G.U.I.

Panorama del sistema

- Descripción escrita sobre objetivo y función del sistema completo - Descripción del negocio y alcance del sistema - Perfiles de usuarios - Evolución de lo expresado en el plan general del proyecto

Panorama de la aplicación

- Descripción escrita para cada aplicación contenida en el sistema - Flujos y circuitos de la información. Ejemplo: gestión de pedidos, control de stock de mercadería, etc.

- Define características de la aplicación - Describe la estructura de la aplicación en línea - Resumen de eventos del negocio - Recopilación de los documentos previos - Se utilizará en la ayuda en línea o documentación impresa

Productos del Diseño de G.U.I:

Diagrama de navegación de ventanas

- Muestra ventanas o páginas disponibles y rutas de navegación - Es un prototipo no operacional-Refinamiento - Definición de unidades de trabajo adecuadas - Grafo dirigido o digrafo - Notación del diagrama:

Rectángulo = ventana (nombre significativo) Flecha simple = invocación o derivación de ventana Flecha doble punta = invocación y retorno Limite (número o M) = cantidad máxima de instancias - Se puede incluir el botón de invocación o nominar la flecha - Unidad de trabajo (tamaño de la transacción):

- El diagrama de navegación de ventanas permite definir claramente los límites de la unidad de trabajo, establecer el lugar de cierre.

- Establecer ubicación del botón de Guardar o Confirmar. - En el caso contener varias ventanas utilizar una línea punteada para definir el alcance de la operación Guardar

Disposición de ventana

- Se refiere a la descripción de cada ventana, su contenido, las etiquetas, los aspectos estéticos y encuadre general.

- Cohesión de la información contenida. - Se puede maquetar o realizar un prototipo. **Descripción de ventana**

- Objetivo de la ventana - Descripción de los eventos que se manejan en ella - Punto de inicio - Etc...

Mini-especificación de la ventana

- Qué sucede cuando se acciona sobre la ventana

- Cuando se abre la ventana - Cuando se cierra la ventana - Al presionar botón - Etc...

Especificación de atributo

CODIFICACION EFECTIVA

- Forma en que los datos pueden ser capturados precisa y eficientemente mediante el empleo con conocimiento de varios códigos.

- Proceso de poner datos no ambiguos o problemáticos en unos cuantos dígitos o letras fácilmente capturables.

- Ayuda a que el analista de sistemas alcance el objetivo de eficiencia, debido a que los datos que son codificados requieren menos tiempo para su captura y reducen la cantidad de conceptos capturados.

- También puede ayudar en el reordenamiento adecuado de los datos en un punto posterior del proceso de transformación de datos.

- Los datos codificados pueden ahorrar espacio valioso de memoria y de almacenamiento. **Los objetivos de la codificación incluyen:**

- Hacer seguimiento de algo. - Clasificar información. - Ocultamiento de información. - Revelar información. - Solicitar una acción adecuada.

Para el establecimiento de un sistema de codificación el analista debe:

- Mantener los códigos concisos. - Mantener los códigos estables. - Hacer códigos que sean únicos. - Permitir que los códigos sean ordenables. - Evitar códigos confusos. - Mantener los códigos uniformes. - Permitir la modificación de los códigos. - Hacer que los códigos sean significativos.

PATRONES GRASP:

Los patrones del diseño que se requieren para construir buenos diagramas de interacción pueden codificarse, explicarse y utilizarse en forma metódica. Esta manera de entender y usar los principios del diseño se funda en los patrones con que se asignan las responsabilidades. Éstas se relacionan con las obligaciones de un objeto respecto a su compartimiento y se asignan a los objetos durante el diseño orientado a objetos.

Los patrones GRASP (General Responsability Asignament Software Paterns) describen los principios fundamentales de la asignación de responsabilidades a objetos, expresados en forma de patrones.

- Patrón experto:

- Asignar una responsabilidad al experto en información, es decir, la clase que cuenta con la información necesaria para cumplir la responsabilidad.

- Se conserva el encapsulamiento, ya que los objetos se valen de su propia información para hacer lo que se les pide.

- **Patrón Creador:**

- Asignarle a una clase B la responsabilidad de crear una instancia de clase A en uno de los siguientes casos:

- B agrega los objetos A - B contiene los objetos A - B registra las instancias de los objetos A - B utiliza específicamente los objetos A - B tiene los datos de inicialización que serán transmitidos a A cuando este objeto sea creado (B es un Experto respecto a la creación de A)

- Un buen ejemplo seria una instancia Pago que al momento de ser creada necesitara inicializarse con el total Venta. Como Venta conoce el total es un buen candidato para ser el partero creador de Pago.

- Patrón Bajo Acoplamiento:

- El acoplamiento es una medida de la fuerza con que una clase está conectada a otras clases, con que las conoce y con que recurre a ellas. Una clase con bajo (o débil) acoplamiento no depende de muchas otras, mientras que una con un alto acoplamiento si lo hará.

- El Bajo Acoplamiento es un principio que se debe recordar durante las decisiones de diseño, siendo la meta principal a tener siempre presente. Es un patrón evaluativo que el diseñador debe aplicar al juzgar sus decisiones de diseño.

- Patrón Alta Cohesión:

- La cohesión es una medida de cuán relacionadas y enfocadas están las responsabilidades de una clase. Una alta cohesión caracteriza a las clases con responsabilidades estrechamente relacionadas que no realicen un trabajo enorme, y que comparten el esfuerzo si la tarea es grande.

- Como el patrón Bajo Acoplamiento, también Alta Cohesión es un principio que se debe tener presente en todas las decisiones de diseño, siendo un patrón evaluativo que el desarrollador aplica para valorarlas. Una clase con mucha cohesión es útil ya que resulta bastante fácil darle mantenimiento, entenderla y reutilizarla.

- Patrón Controlador:

- Asignar la responsabilidad del manejo de un mensaje de los eventos de un sistema a una clase que represente de alguna forma una representación global del sistema o parte de él.

- Un defecto frecuente al diseñar controladores consiste en asignarles demasiada responsabilidad. Normalmente un controlador debería delegar a otros objetos el trabajo que ha de realizarse mientras coordina la actividad.

- Delegar a un controlador la responsabilidad de la operación de un sistema entre las clases del dominio soporta la reutilización de la lógica para manejar los procesos afines del negocio en aplicaciones futuras.

- Controladores Saturados:

- Una clase controlador mal diseñada presentará baja cohesión, estando dispersa y teniendo demasiadas áreas de responsabilidad. Un controlador de

este tipo recibe el nombre de controlador saturado.

- **Polimorfismo:**

  - Cuando por el tipo varían las alternativas o comportamientos afines, las responsabilidades del comportamiento se asignarán a los tipos en que el comportamiento presenta variantes.

  - Facilita agregar las futuras extensiones que requieren las variaciones imprevistas.

- **Fabricación Pura:**

  - Cuando se debe asignar un conjunto altamente cohesivo de responsabilidades a una clase artificial que no representa nada en el dominio del problema.

  - Una Fabricación Pura es una cosa invitada para dar soporte a una alta cohesión, un bajo acoplamiento y reutilización.

- **Indirección:**

  - Se asigna la responsabilidad a un objeto intermedio para que medie entre otros componentes o servicios, y éstos no terminen directamente acoplados.

  - El intermediario crea una indirección entre el resto de los componentes o servicios.

- **No Hables con Extraños:**

  - Se asigna la responsabilidad a un objeto directo del cliente para que colabore con un objeto indirecto, de modo que el cliente no necesite saber nada del objeto indirecto.

  - Los objetos directos son “conocidos” del cliente, los objetos indirectos son “extraños”, y un cliente debería tener solo conocidos, no extraños.