**Software**

1) instrucciones (programas de cómputo) que cuando se ejecutan proporcionan las características, función y desempeño buscados.

2) estructuras de datos que permiten que los programas manipulen en forma adecuada la información.

3) información descriptiva tanto en papel como en formas virtuales que describen la operación y uso de los programas

**Características del Software**

Su desarrollo o modificación se hace con intelecto, no se manufactura. No se desgasta físicamente, aunque sí se deteriora funcionalmente.

Gran parte se desarrolla para un uso individualizado.

Requiere de muchos recursos en su ingeniería, pero poco en su reproducción.

Su distribución es amplia debido a internet y redes.

Transforma datos en información útil.

Controla funciones de sistemas basados en computadoras.

**Ingeniería de software**

es la aplicación de principios de ingeniería al desarrollo, operación y mantenimiento del software mediante un proceso, métodos y herramientas. El objetivo es obtener software confiable, eficiente y que satisfaga las necesidades del cliente de modo económico. Incluye actividades estructurales, acciones de ingeniería, productos del trabajo, y revisiones para el control de calidad.

**Modelo de Ingeniería de Software**

**Especificación**: establecer los requerimientos y restricciones del sistema.

**Diseño**: modelar la solución.

**Construcción:** desarrollar el sistema.

**Prueba:** verificar y validar que el sistema cumpla con las especificaciones requeridas.

**Instalación**: entregar el sistema al usuario y asegurar su funcionamiento.

**Mantenimiento**: ¿reparar fallos en el sistema cundo sea descubiertos?

**Proceso prescriptivo**

**Modelo de Cascada.** Separa en distintas fases la especificación y el desarrollo, comunicación -> planeación -> modelado -> construcción -> despliegue

**Desarrollo Evolutivo.** La especificación y el desarrollo están intercalados.

**Ventajas:** Para sistemas interactivos pequeños o medianos.

Para partes de sistemas grandes (por ejemplo, la interfaz de usuario).

Para sistemas de corta vida.

**Desventajas:** Poca visibilidad en el proceso.

Los sistemas están pobremente especificados.

Se requieren habilidades especiales.

**Modelo espiral.** El sistema es ensamblado a partir de componentes existentes.

**Ventajas:** Centra su atención en la reutilización de componentes y eliminación de errores en información descubierta en fases iniciales.

Los objetivos de calidad son primordiales.

Integra desarrollo con mantenimiento.

Provee un marco de desarrollo de hardware/software.

**Desventajas:** El desarrollo contractual especifica el modelo del proceso y los resultados a entregar por adelantado.

Requiere de experiencia en la identificación de riesgos.

Requiere refinamiento para uso generalizado.

**Modelo incremental:** Produce incrementos de software mediante iteraciones de las actividades estructurales. Cada incremento es más completo

**Prototipado.** Un modelo sirve de prototipo para la construcción del sistema final.

**Transformación Formal.** Un modelo matemático del sistema se transforma formalmente en la implementación.

**Características que hacen diferente al software de otras cosas fabricadas**

Se desarrolla con intelecto, no se manufactura.

No se desgasta físicamente.

Gran parte se desarrolla para uso individual.

Requiere recursos en ingeniería más que en reproducción. Distribución amplia debido a redes.

Transforma datos en información. Controla funciones de sistemas.

**Proceso del Software**

Un proceso del software es una estructura para las actividades, acciones y tareas necesarias para construir software de alta calidad a través de la identificación de actividades estructurales y sombrilla.

**Actividades Proceso de Software**

**Comunicación**: tiene importancia crítica

comunicarse y colaborar con el cliente (y con otros participantes).

**Planeación**: define el trabajo al describir las tareas técnicas por realizar, los riesgos probables, los recursos que se requieren, los productos del trabajo que se obtendrán y una programación de las actividades., modelado, construcción y despliegue.

**Construcción**. combina la generación de código (ya sea manual o automatizada) y las pruebas que se requieren para descubrir errores en éste.

**Despliegue.** El software (como entidad completa o como un incremento parcialmente terminado) se entrega al consumidor que lo evalúa y que le da retroalimentación, misma que se basa en dicha evaluación.

**Actividades sombrilla:** Son aquellas que ocurren a lo largo del proceso, como seguimiento y control, medición, administración de riesgos, aseguramiento de calidad, entre otras.

**Actividades sombrilla más comunes**

Seguimiento y control

Administración de riesgos

Aseguramiento de calidad

Administración de configuración

Revisiones técnicas

**Modelo de Proceso Concurrente**

Representa elementos iterativos y concurrentes de cualquier modelo. Define estados para cada actividad, acción o tarea y eventos que disparan transiciones entre estados.

**Modelos de Proceso Especializados**

Desarrollo basado en componentes

Modelo de métodos formales

Desarrollo orientado a aspectos

**Proceso ágil:** Sigue principios ágiles y valores. Enfatiza la comunicación, planeación rápida, modelado, construcción e implementación en iteraciones cortas. combina filosofía y lineamientos. La filosofía enfatiza individuos e interacciones, software funcional, colaboración con cliente, respuesta al cambio.

**¿Qué caracteriza a los primeros modelos ágiles como XP (Programación Extrema)?**

Valores XP: comunicación, simplicidad, feedback, coraje Rol papel y tareas en equipos autoorganizados

Ciclos cortos, integración continua, pruebas automáticas

**¿Cuáles son otros modelos ágiles además de XP?** Proceso Unificado Ágil, Cristal, Dinámico, Esbelto, Ágil Scrum, DAS, Adaptativo de Software Dinámico

**Desarrollo Adaptativo de Software**

Itera ciclos cortos de planificación, diseño, codificación y pruebas. Usa métricas ágiles para adaptarse.

**¿Qué caracteriza al método Scrum?**

Grupos autoorganizados

Sprints o iteraciones cortas

Reuniones diarias cortas

Tablero Kanban visual

Papel del Scrum Master

**Patrones del proceso:** describen problemas asociados con el proceso, el contexto donde surgen y sugieren soluciones. Pueden ser de fase, etapa o tarea.

**Valor de emplear patrones de proceso: P**roporcionan contexto, problema y solución sobre eventos concurrentes del proceso. Sirven de bloques para modelar el proceso y reutilizar soluciones probadas a problemas recurrentes.

**Software Heredado y Nuevo:** El heredado ya existe y cumple una función importante del negocio. El nuevo responde a necesidades nuevas.

**Características del software heredado**

Antiguo, importante para el negocio, sujeto a cambios, adaptaciones a nuevos ambientes, reingeniería.

**¿Por qué es importante considerar al software como un producto dual (producto y vehículo)?**

El software es único porque actúa como un producto que distribuye información, pero también funciona como un vehículo para distribuir otros productos. Esto implica que el software tiene características distintas al hardware en cuanto a su desarrollo e implementación. Desarrollar software requiere un enfoque diferente al de fabricar hardware, dado que el software no se desgasta físicamente, aunque es susceptible a defectos con el tiempo. Su distribución también difiere, pudiendo entregarse a través de redes. Reconocer la naturaleza dual del software es clave para aplicar un enfoque de ingeniería apropiado.

**¿Por qué el software heredado plantea retos especiales?** El software heredado es crítico para el negocio, pero a menudo se caracteriza por diseños no extensibles, código confuso, documentación deficiente o inexistente, casos y resultados de prueba no registrados. Si evoluciona para cumplir nuevos requerimientos o migrar a plataformas modernas, requiere esfuerzos de reingeniería que pueden ser complejos. Además, generalmente tiene más años que el personal que lo mantiene, por lo que es más difícil comprender su funcionalidad y estructura.

**Herencia**: se utiliza para describir la relación entre la clase padre y la clase hija. La subclase hereda todas las funciones de la clase principal y la clase principal tiene todos los atributos, métodos y subclases.

**Polimorfismo** se refiere a que una característica de una clase puede tomar varias formas.

Representa la posibilidad de desencadenar operaciones distintas en respuesta a un mismo mensaje.

**Encapsulamiento**: Se protegen los datos de accesos indebidos.

El acoplamiento entre las clases se disminuye.

Favorece la modularidad y el mantenimiento.

Los atributos de una clase no deberían ser manipulables directamente por el resto de objetos.

**Interfaz**: Las interfaces permiten definir contratos que las clases deben cumplir, asegurando que ciertos métodos estén presentes y promoviendo una abstracción adecuada.

**Desarrollo Basado en Componentes:** Busca desarrollar componentes de software reutilizables, que puedan incorporarse en múltiples programas. Los componentes encapsulan datos y procesamiento en una interfaz conocida. El desarrollo se basa en la ingeniería de dominios, calificación, adaptación y combinación de componentes existentes. Define un proceso evolutivo.

**Actividades Estructurales del Proceso Unificado**

Incepción, elaboración, construcción, transición y producción. Coinciden con las actividades generales de comunicación, modelado, construcción y despliegue respectivamente.

**Modelo de Prototipos:** Permite que a través de prototipos iterativos se obtenga retroalimentación del cliente para mejorar requerimientos. Se construye con diseño simplificado para su evaluación, no para su uso. Finalmente se desecha y se construye una versión real siguiendo la retroalimentación.

**Objetivo de hacer métricas del proceso y producto:**

Evaluar el estado actual del proceso para mejorarlo, identificando fortalezas y áreas de oportunidad. Proporcionan evidencia empírica del desempeño y efectividad para la mejora continua del proceso.

**Modelo de Métodos Formales:** Énfasis en formalización matemática de requerimientos, diseño y verificación. Permite especificar, desarrollar y verificar usando notación formal. Su objetivo es software libre de defectos mediante análisis riguroso.

**Desarrollo Orientado a Aspectos** Permite definir "aspectos" que expresan preocupaciones globales del cliente que afectan múltiples funciones e información del sistema. Son "mecanismos más allá de subrutinas y herencia" para localizar preocupaciones transversales al sistema.

**Patrón de Fase, Etapa y Tarea:** Un patrón de fase define la secuencia de actividades estructurales. Un patrón de etapa describe un problema asociado a una actividad estructural. Un patrón de tarea define un problema relacionado con una acción o tarea técnica de la ingeniería de software.

**Proceso Personal de Software** Énfasis en medición personal para la mejora continua, planeación y autodirección. Busca construir equipos con capacitación, planeación y control de su propio proceso. Centra la atención en métricas para medir producto y proceso.

**Proceso del Equipo de Software:** Busca generar equipos autodirigidos que planeen, midan su trabajo, establezcan metas y sean dueños de su proceso. Enfatiza la medición, planeación y autonomía como elementos centrales para el éxito del equipo y la organización.

**Principales técnicas de prueba:** prueba de unidad, integración, validación, regresión, confiabilidad, usabilidad, compatibilidad y rendimiento. Cada técnica requiere una planeación detallada incluyendo casos de prueba, datos, expectativas, registro de hallazgos y evolución del plan.

**Enfoque estratégico para la prueba de software:** Un enfoque estratégico define el alcance, enfoque, recursos y programación de las actividades de verificación y validación buscando satisfacer objetivos de calidad del proceso y producto.

**Estrategias de prueba de software:** Verificación, pruebas unitarias, de integración, de validación, alfa, beta, sistemas, recuperación, seguridad, rendimiento, de esfuerzo.

**Prueba de Unidad:** Comprueba un componente aislado del software, usualmente una función, procedimiento o módulo. Evalúa la corrección y completitud de cada unidad aislada.

**Prueba de Integración:** Prueba la interfaz y comunicación entre módulos unidos de software. Detecta errores de integración temprano.

**Prueba Orientada a Objetos:** Permite el aislamiento de componentes para prueba, herencia de casos de prueba, jerarquía de clases para organizar casos.

**Pruebas de Validación:** Representan condiciones reales de uso, empleo de equipos finales, validan características visibles al usuario, satisfacen requerimientos.

**Pruebas Alfa y Beta:** Pruebas alfa validan el software cuando está listo para entrega. Las beta prueban el sistema en condiciones reales de usuarios, antes de su lanzamiento comercial.

**Pruebas de Sistema:** Integradoras, validan interacción de componentes, simulan ambiente real de operación, consideran capacidades de carga y rendimiento.

**Pruebas de Recuperación:** Evalúan la habilidad de recuperarse de fallos, pérdidas de datos, desastres, simulan fallos y recuperaciones de sistemas y componentes.

**Pruebas de seguridad:** Validan la resistencia del software a amenazas, ataques, vulnerabilidades y fallos de seguridad. Prueban características de confidencialidad, integridad y disponibilidad.

**Pruebas de rendimiento:** Miden eficiencia en términos de velocidad y uso de recursos, y determinan cómo responde el sistema ante diferentes cargas de trabajo.

**Pruebas de Esfuerzo:** Solicitan cargas de trabajo extremas más allá de los requerimientos normales para identificar límites y fallas de diseño ante carga excesiva.

**Técnicas para pruebas de interfaces gráficas de usuario:** Scripts, métodos cualitativos e inspecciones visuales, automatización de las pruebas, validar aspecto y aspectos sensoriales, medir el rendimiento.

**Estrategias para pruebas de aplicaciones orientadas a objetos:** Pruebas unitarias, de integración, validación en el contexto OO, patrones de diseño de pruebas, métricas para OO Testing.

**Técnicas para Pruebas Webapps:** Prueba de contenido, interfaz, mecanismos, semántica, usabilidad, compatibilidad, componente, navegación, configuración, rendimiento, seguridad.

**Pruebas Formales:** Especificación formal, diseño formal, verificación formal, lenguajes de especificación formal, métodos abstractos, corrección matemática, demostración de teoremas.

**Lenguajes de Especificación Formal:** OCL, lógica de restricciones de objetos; Z, lenguaje de especificación; VDM, método Vienés de especificación.

**Modelado y Verificación de Cuarto Limpio:** Especificación funcional, diseño de cuarto limpio, pruebas de cuarto limpio, verificación de diseño, certificación formal, refinamiento de diseño.

**SCAMPI y CBA IPI:** Métodos formales de evaluación del proceso usando el CMMI como base; proporcionan diagnóstico y estrategia para la mejora.

**Características de las Métricas para Pruebas**

Métricas de cobertura, métricas de defectos, métricas de rendimiento, métricas para OO Testing, métricas de halstead aplicadas a pruebas, métricas para pruebas ágiles.

**Principios SOLID**

**Principio de Responsabilidad Única:** un objeto debe realizar una única cosa.

**Principio Open/Close**: Abierto a la extensión: Se puede agregar un nuevo comportamiento a futuro. El código que está cerrado a la extensión tiene un comportamiento fijo.

Cerrado a la modificación: No se realizan cambios en el código fuente o binario. La única forma de cambiar el comportamiento del código que está cerrado a la extensión es cambiar el código en sí.

**Principio Sustitución Liskov**: si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido.

**Principio Segregación Interfaz**: ninguna clase debería depender de métodos que no usa. Por tanto, cuando creemos interfaces que definan comportamientos, es importante estar seguros de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. En caso contrario, es mejor tener varias interfaces más pequeñas.

**Principio Inversión Dependencia**:

1. Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.
2. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

namespace StringManipulatorTests

{

public class StringManipulatorTests

{

//**Metodo**

private readonly StringManipulator.StringManipulator stringManipulator;

//**Constructor**

public StringManipulatorTests()

{

this.stringManipulator = new StringManipulator.StringManipulator();

}

//Test **voltear palabra simple**

**[Fact]**

public void ReverseString\_WhenCalled\_ReturnsReversedString()

{

// **Arrange**

string input = "Hello World";

// **Act**

string result = stringManipulator.ReverseString(input);

// **Assert**

Assert.Equal("dlroW olleH", result);

}

//**Test voltear palabra multiple**

[**Theory]**

[InlineData("Chau", "uahC")]

[InlineData("Hola", "aloH")]

public void Theory\_ReverseString\_WhenCalled\_ReturnsReversedString(string word, string drow)

{

// **Arrange**

string input = word;

// **Act**

string result = stringManipulator.ReverseString(input);

// **Assert**

Assert.Equal(drow, result);

}

//**Test contar palabra simple**

[Fact]

public void CountWords\_WhenCalled\_ReturnsNumberOfWords()

{

// Arrange

string input = "This is a sample sentence.";

// Act

int result = stringManipulator.CountWords(input);

// Assert

Assert.Equal(5, result);

}

//**Test contar palabra simple**

[Theory]

[InlineData("Ian es un hinchapelotas.",4)]

//[InlineData("Hola", "aloH")]

public void Theory\_CountWords\_WhenCalled\_ReturnsNumberOfWords(string sentence, int drow)

{

// Arrange

string input = sentence;

// Act

int result = stringManipulator.CountWords(input);

// Assert

Assert.Equal(drow,result);

}

//**Test contar espacios vacios simple**

[Fact]

public void CountWords\_WithEmptyString\_ReturnsZero()

{

// Arrange

string input = "";

// Act

int result = stringManipulator.CountWords(input);

// Assert

Assert.Equal(0, result);

}

//**Test contar palabra excepcion null simple**

[Fact]

public void CountWords\_WithNullInput\_ThrowsNullReferenceException()

{

// Arrange

string input = null;

// Act & Assert

Assert.Throws<NullReferenceException>(() => stringManipulator.CountWords(input));

}

//**Test contar palabra excepcion cero caracteres x linea simple**

[Fact]

public void CountWords\_WithSubZeroCharactersPerLine\_ThrowsArgumentException()

{

// Arrange

string input = "Hola";

int charsByLine = 0;

// Act & Assert

Assert.Throws<ArgumentException>(() => stringManipulator.CountLines(input, charsByLine));

}

//**Test contar palabra excepcion cero caracteres x linea simple**

[Fact]

public void CountWords\_WithSubZeroCharactersPerLine\_ThrowsArgumentException2()

{

// Arrange

string input = "Hola";

int charsByLine = 0;

// Act & Assert

try

{

stringManipulator.CountLines(input, charsByLine);

} catch (ArgumentException ex)

{

Assert.True(true);

return;

}

Assert.True(false);

}

[Theory]

[InlineData("Ian es un hinchapelotas",20)]

public void CountCharsWithoutSpace(string sentence, int drow)

{

// Arrange

string input = sentence;

// Act

int result = stringManipulator.CountCharsWithoutSpace(input);

// Assert

Assert.Equal(drow, result);

}

}

}