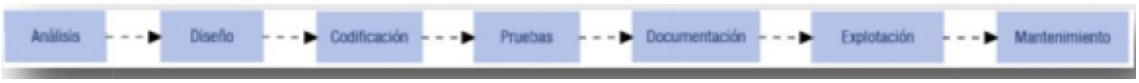
INTRODUCCIÓN.

EL desarrollo de SW pasa por una serie de fases o etapas conocidas como “ciclo de vida del software”, que van desde la necesidad de crear un software hasta que se finaliza el programa y se deja listo para ser explotado.

CICLO DE VIDA DEL SOFTWARE

El estándar ISO/IEC 12207-1 define el ciclo de vida como: *Un marco de referencia que contiene los procesos, las actividades y las tareas involucradas en el desarrollo, la explotación y el mantenimiento de un producto de software abarcando la vida del sistema desde la definición de requisitos hasta la finalización de su uso.*



2.1 Análisis

La fase de análisis define los requisitos del SW que hay que desarrollar.

La etapa comienza con una entrevista al cliente, que establecerá que quiere o que necesita, lo cual nos dará una buena idea global del problema a abordar, pero no necesariamente del todo acertada. Aunque el cliente crea que sabe lo que el software tiene que hacer, es necesaria una buena habilidad y experiencia para reconocer requisitos incompletos, ambiguos, contradictorios o incluso necesarios.

La comunicación bilateral con el cliente es esencial en esta etapa, siendo necesario un contraste y un consenso por ambas partes para llegar a definir los verdaderos requisitos del software.

Para facilitar esta comunicación se utilizan varias técnicas:

Entrevistas. La técnica más tradicional que consiste en hablar con el cliente.

Desarrollo conjunto de aplicaciones (JAD). Es un tipo de entrevista muy estructurada y aplicable a grupos de personas, en las que mediante dinámica de grupos cada persona juega un rol concreto.

Planificación conjunta de requisitos (JRP). Un subconjunto de JAD dirigido a la alta dirección. Los productos resultantes son los requisitos de alto nivel o estratégicos.

Brainstorming. Reuniones en grupo cuyo objetivo es generar ideas desde diferentes puntos de vista para la resolución de un problema. Su utilización más adecuada suele ser al principio del proyecto.

Prototipos. Versiones iniciales del sistema usados para clarificar algunos puntos, demostrar conceptos, etc.

Casos de uso. Se basa en escenarios que describen cómo se usa el software en determinas situaciones.

Como resultado de esta comunicación se especifican los requisitos funcionales y no funcionales del programa:

Requisitos funcionales: Las funciones que debe hacer la aplicación, la respuesta que dará a las entradas y cómo se comportará ante situaciones inesperadas.

Requisitos no funcionales: tiempos de respuesta del programa, legislación aplicable, tratamiento de peticiones simultáneas, etc.

El documento que se genera recogiendo todos estos requisitos se conoce como ERS (Especificación de Requisitos del Software).

2.2 Diseño

Ahora que tenemos claro lo que hay que hacer, la siguiente fase es como hacerlo. El sistema le dividiremos en partes y se establece las relaciones entre ellas, decidiendo lo que hace cada parte.

En esta etapa se pretende determinar el funcionamiento de una forma global y general, sin entrar en detalles. Uno de los objetivos principales es establecer las consideraciones de los recursos del sistema, tanto físicos como lógicos.

El proceso implica tomar decisiones y documentar:

Decisiones: diseño para los datos, entidades y relaciones, lenguaje de programación, etc.

Documentación: Documento de Diseño de SW (SDD). Describe la estructura global del sistema, especifica que debe hacer cada módulo y como se van a relacionar.

2.2.1 Notaciones gráficas para el diseño

Para representar el diseño se emplean herramientas gráficas como las que veremos a continuación.

Pseudocódigo

Su definición es: *En ciencias de la computación, y análisis numérico, el pseudocódigo (o falso lenguaje) es una descripción de alto nivel compacta e informal del principio operativo de un programa informático u otro algoritmo*.

El pseudocódigo utiliza texto descriptivo para realizar el diseño de un algoritmo. A primera vista parecerá un lenguaje de programación, ya que mezcla frases en lenguaje natural con estructuras sintácticas que incluyen palabras clave que permiten construir las estructuras básicas de la programación estructurada, declarar datos, definir subprogramas y establecer características de modularidad.

Dependerá de quién lo escriba, ya que no hay un estándar definido.

Ejemplo 1: Programa que lee números y muestra la suma en pantalla

```
Inicio
        Leer numero1, numero2
        Suma=numero1+numero2
        Visualizar “la suma es = “ Suma
```

Fin

Diagramas de flujo

Un diagrama de flujo es una forma de representar gráficamente la secuencia de un proceso. Se utiliza en cualquier ciencia como en la programación, la economía y los procesos industriales.

Estos diagramas utilizan una serie de símbolos con significados especiales y son la representación gráfica de los pasos de un proceso. En computación, son modelos tecnológicos utilizados para comprender los rudimentos de la programación secuencial.

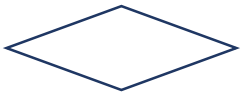
Los símbolos permiten crear la estructura gráfica que describe los pasos a seguir para obtener un resultado específico. Este diagrama facilita la escritura del programa en algún lenguaje de programación.

Los símbolos utilizados en los diagramas de flujo serían los siguientes:

Inicio y fin del diagrama de flujo.

Entrada de datos o lectura de datos.

Proceso.



Decisión.

Decisión múltiple.

Salida de información. Impresión de información.

Dirección.

Conexión dentro de una misma página.

Conexión entre páginas.

Las reglas para el diseño de un diagrama de flujo serán las siguientes:

Todo diagrama de flujo debe de tener un inicio y un fin.

Las líneas utilizadas para indicar la dirección de un flujo deben de ser rectas, horizontales o verticales, no inclinadas y tampoco deben de cruzarse.

Todas las líneas utilizadas para la dirección del flujo del diagrama deben de estar conectadas, esto es, no debe de haber "líneas sueltas".

El diagrama de flujo debe de ser diseñado de arriba hacia abajo y de izquierda a derecha.

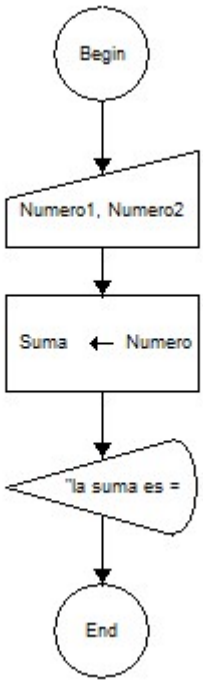
La notación utilizada debe ser independiente del lenguaje de programación para que se pueda traducir a cualquier lenguaje de programación.

Es conveniente poner comentarios que complemente la descripción del proceso.

Si el diagrama requiere de más de una página para su construcción, utilizar los conectores debidamente enumerados.

No debe de llegar más de una línea a un símbolo.

Ejemplo de diagrama de flujo: Suma de dos números.



Diagramas de cajas

Esta notación surgió del deseo de desarrollar una representación para el diseño procedimental que no permitiera la violación de construcciones estructuradas. Tienen las características siguientes:

El ámbito funcional está bien definido y es claramente visible.

La transferencia de control arbitraria es imposible.

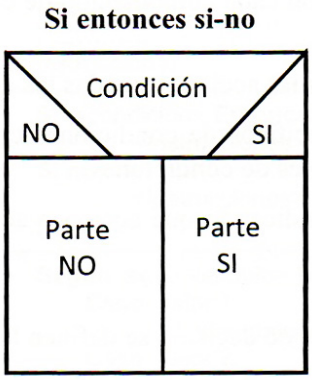
Es fácil determinar el ámbito de los datos locales y globales.

La recursividad es fácil de representar.

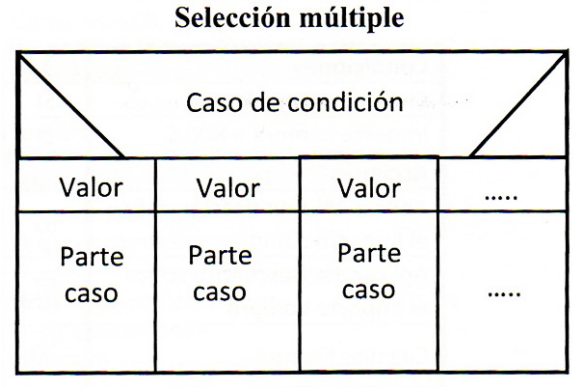
Las reglas en el diseño de diagramas de cajas son:

Para representar una secuencia se conectan varias cajas seguidas.

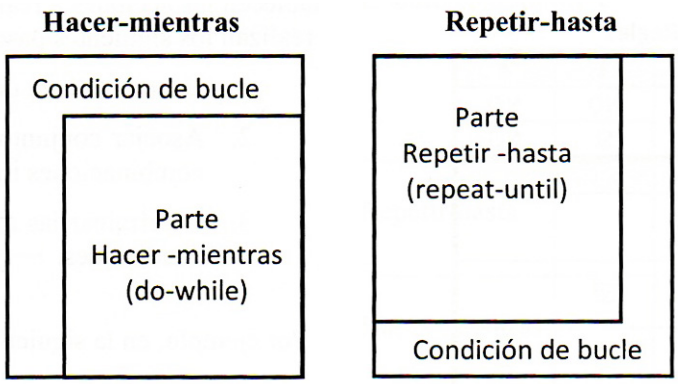
Para una condición se representa una caja para la Parte SI y otra para la Parte No, justo encima se ha de especificar la condición a la que hacen referencia.



En la selección múltiple se representará el caso de condición y se definen tantas columnas como valores se vayan a comprobar en la dicha condición, en las cuáles se indica la acción a realizar.



En los bucles, el proceso a repetir se encierra en una caja que está dentro de otra donde se indica la condición de bucle, ya sea en la parte superior (*do-while*) o inferior (*repeat-until*).



Tablas de decisión

En muchas aplicaciones de software puede ser necesario que un módulo evalúe una compleja combinación de condiciones, y de acuerdo con ellas, seleccione la opción adecuada.

Las tablas de decisión permiten representar en forma de tabla las condiciones y las acciones que se llevan a cabo combinando esas condiciones.

Condiciones	1	2	3	…	n
Condición 1	✓		✓		
Condición 2		✓	✓		
Condición 3	✓				
…					
Acciones					
Acción 1	✓		✓		
Acción 2	✓	✓			
Acción 3		✓			
…					

Se dividen en cuatro cuadrantes:

Cuadrante superior izquierdo: que contiene todas las listas de condiciones posibles.

Cuadrante inferior izquierdo: que contiene todas las acciones posibles basándose en la combinación de condiciones.

Cuadrante superior derecho: entrada de las condiciones.

Cuadrante inferior derecho: entrada de las acciones.

Cada columna de los cuadrantes de la derecha forma una regla de procesamiento, estableciendo las acciones a realizar con cada combinación de condiciones.

Para construir una tabla se realizan los siguientes pasos:

Hacer una lista con todas las acciones y todas las condiciones.

Asociar conjuntos de específicos de condiciones con acciones específicas, eliminando combinaciones imposibles de condiciones.

Determinar las reglas indicando qué acción o acciones ocurren para un conjunto de condiciones.

Un ejemplo de tabla de decisiones

Condiciones	1	2	3	4
Cliente registrado		✓		✓
Importe compra > 800€			✓	✓
Acciones				
Aplicar 1% bonificación sobre importe de compra		✓		✓
Aplicar 3% descuento sobre importe de compra			✓	✓
Calcular factura	✓	✓	✓	✓

2.3 Codificación

La fase más obvia en el proceso de desarrollo de software es sin duda la codificación. Una vez definido el software que hay que crear habrá que programarlo, crear su código fuente.

Gracias a las etapas anteriores, el programador contará con un análisis completo del sistema que hay que codificar y con una especificación de la estructura básica que se necesitará, por lo que en un principio solo habría que traducir el cuaderno de carga en el lenguaje deseado para culminar la etapa de codificación, pero esto no es siempre así, las dificultades son recurrentes mientras se modifica.

Por supuesto que cuanto más exhaustivo haya sido el análisis y el diseño, la tarea será más sencilla, pero nunca está exento de necesitar un re-análisis o un rediseño al encontrar un problema al programar el software.

Existen muy diversas metodologías en el desarrollo o codificación de software, cuyo objetivo es presentar un conjunto de técnicas tradicionales y modernas de modelado de sistemas que permitan desarrollar software de calidad. Para tal fin describen, fundamentalmente, herramientas de Análisis y Diseño Orientado a Objetos (UML), sus diagramas, especificación, y criterios de aplicación de las mismas.

Normas de codificación

En cualquier proyecto en el que trabaja un grupo de personas debe haber unas normas de codificación y estilo, claras y homogéneas. Estas normas facilitan las tareas de corrección y

mantenimiento de los programas, sobre todo cuando se realizan por personas que no los han desarrollado.

Por ejemplos, estos dos programas son el mismo, pero uno de ellos resulta más fácil de leer:

```
public class Ejemplo {
    public static void main (String[] args) {
        int suma = 0;
        int contador = 0;
        while(contador < 10) {
            contador++;
            suma = suma + contador;
        }
        System.out println("Suma => " + suma);
    }
}
```

```
public class Ejemplo {
public static

void main (String[] args) {

    int suma = 0; int contador = 0;
            while(contador < 10)
{
                    contador++;
        suma = suma + contador; }
    System.out println("Suma => "
  + suma);
    }
}
```

A continuación, se describen una serie de normas de escritura de código fuente en Java que facilita la lectura de los programas haciendo que sea más sencillo mantenerlos y encontrar errores, incluso por personas que no han desarrollado el código.

ORGANIZACIÓN DE LOS FICHEROS

Ficheros de código fuente: fichero.java

Ficheros compilados en java: fichero.class.

Ficheros java para ejecutar: fichero.jar

Cada fichero debe contener una sola clase pública y debe ser la primera. Las clases privadas e interfaces asociados con esa clase pública se pueden poner en el mismo fichero después de la clase pública. Las secciones en las que se divide el fichero son:

Comentarios. Todos los ficheros fuente deben comenzar con un comentario que muestra el nombre de la clase, información de la versión, la fecha, y el aviso de derechos de autor.

Sentencias del tipo package e import. Van después de los comentarios, la sentencia *package* va delante de *import*. Ejemplo:

```
/*
* Nombre de clase
*
* Información de la versión
*
* Fecha
*
* Aviso de Copyright
*/
package paquete.ejemplo;
import java.io.*;
```

Declaraciones de clases e interfaces. Consta de las siguientes partes:

Comentario de documentación (/** ... */) acerca de la clase o interface.

Sentencia *class* o *interface*.

Comentario de la implementación (/* ... */) de la clase o interface.

Variables estáticas, en este orden: públicas, protegidas y luego privadas.

Variables de instancia, en este orden: públicas, protegidas y luego privadas.

Constructores.

Métodos. Se agrupan por su funcionalidad, no por su alcance.

IDENTACIÓN

Como norma general se usarán cuatro espacios.

La longitud de las líneas de código no debe superar 80 caracteres.

La longitud de las líneas de comentarios no debe superar 70 caracteres.

Cuando una expresión no cabe en una sola línea: romper después de una coma, romper antes de un operador, alinear la nueva línea al principio de la anterior.

COMENTARIOS

Los comentarios deben contener solo la información que es relevante para la lectura y la comprensión del programa. Existen dos tipos de comentarios: de documentación y de implementación.

Los comentarios de documentación están destinados a describir la especificación del código. Se utilizan para describir las clases Java, los interfaces, los constructores, los métodos y los campos. Debe aparecer justo antes de la declaración. Existe una herramienta llamada Javadoc que genera páginas HTML partiendo de este tipo de comentarios. Tienen el siguiente formato:

```
/**
* La clase Ejemplo proporciona
*/
public class Ejemplo { ...
```

Los comentarios de implementación son para comentar algo acerca de la aplicación particular. Pueden ser de tres tipos:

Comentarios de bloque:

```
/*
* Esto es un comentario de bloque
*/
```

Comentarios de línea:

```
/* Esto es un comentario de línea * /
```

Comentario corto:

```
// Esto es un comentario corto
```

DECLARACIONES

Se recomienda declarar una variable por línea.

Inicializar las variables locales donde están declaradas y colocarlas al comienzo del bloque. Ejemplo:

```
void miMetodo () {
    int varl = 0;        //comienza el bloque de miMetodo
    int var2 = 10 ;
    if (varl = var2) {
        int suma = 0;    //comienza el bloque if
        ....
    } else {
        var2 = varl;
        ....
```

```
        }
        ....
    }
```

En las clases e interfaces:

No se ponen espacios en blanco entre el nombre del método y el paréntesis '('.

La llave de apertura "{" se coloca en la misma línea que el nombre del método o clase.

La llave de cierre"}" aparece en una línea aparte y en la misma columna que el inicio del bloque, excepto cuando el bloque está vacío.

Los métodos se separan por una línea en blanco.

```
    class Ejemplo extends Object {
        int var1;
        int var2;
        Ejemplo(int i, int j) {
            var1 = i;
            var2 = j ;
        }
        int metodoVacio() {}
        ....
    }
```

SENTENCIAS

Cada línea debe contener una sentencia.

Si hay un bloque de sentencias, éste debe ser sangrado con respecto a la sentencia que lo genera y debe estar entre llaves, aunque solo tenga una sentencia.

Sentencias *if-else*, *if efse-if else*. Definen bloques a los que se aplican las normas anteriores. Todos los bloques tienen el mismo nivel de sangrado.

Bucles. Definen un bloque, que sigue las normas anteriores. Si el bucle está vacío no se abren ni se cierran llaves.

Las sentencias *return* no deben usar paréntesis.

SEPARACIONES

Mejoran la legibilidad del código. Se utilizan:

Dos líneas en blanco: entre las definiciones de clases e interfaces.

Una línea en blanco: entre los métodos, la definición de las variables locales de un método y la primera instrucción, antes de un comentario, entre secciones lógicas dentro de un método para mejorar la legibilidad.

Un carácter en blanco: entre una palabra y un paréntesis, después de una coma, los operadores binarios menos el punto, las expresiones del *for*, y entre un *cast* y la variable.

NOMBRES

Los nombres de las variables, métodos, clases, etc., hacen que los programas sean más fáciles de leer ya que pueden darnos información acerca de su función. Las normas para asignar nombres son las siguientes:

Paquetes: el nombre se escribe en minúscula, se pueden utilizar puntos para reflejar algún tipo de organización jerárquica. Ejemplo: `java.io`

Clases e interfaces: los nombres deben ser sustantivos. Se deben utilizar nombres descriptivos. Si el nombre está formado por varias palabras la primera letra de cada palabra debe estar en mayúscula: Ejemplo: `HiloServidor`

Métodos: se deben usar verbos en infinitivo. Si está formado por varias palabras el verbo debe estar en minúscula y la siguiente palabra debe empezar en mayúscula. Ejemplo: `ejecutar(), asignarDestino()`

Variables: deben ser cortas (pueden ser de una letra) y significativas. Si son varias palabras la primera debe estar en minúscula. Ejemplos: `i, j, sumaTotal`

Constantes: el nombre debe ser descriptivo. Se escriben en mayúsculas y si son varias palabras van unidas por un carácter de subrayado. Ejemplo: `MAX_VALOR`

2.4 Pruebas.

Coincidiendo con los diversos hitos durante el desarrollo de un producto software se suelen realizar controles. Durante estos controles se evalúa la calidad de la entrega y se busca detectar fallos o errores en ellas. El objetivo es detectar los fallos cuanto antes.

Las pruebas son un método para validar y verificar el software.

La validación se realiza al finalizar por completo el desarrollo, para determinar si satisface los requisitos especificados.

Trata de responder a la pregunta: ¿Estamos construyendo el producto correcto?

La verificación se realiza al final de cada fase y se comprueba el cumplimiento de los requisitos de esa fase.

Se trata de responder a la pregunta ¿Estamos construyendo el producto correctamente?

Durante el periodo de pruebas se toman un conjunto de entradas posibles o casos de prueba y se busca encontrar defectos, fallos o errores en el programa. Los defectos son procesos incorrectos que generan salidas erróneas. Los fallos son también la incapacidad para cumplir un requisito. Por ejemplo, la incapacidad de procesamiento de datos por segundo requerida.

Recomendaciones

Pueden hacerse una serie de recomendaciones para el proceso de pruebas:

Para cada caso de prueba debe definirse previamente cuál es el resultado esperado. Ese resultado se comparará con el realmente obtenido en la ejecución de la prueba.

El programador debe evitar probar sus propios programas. Es evidente que, si en la fase de diseño o programación no ha tenido algo en cuenta, muy probablemente volverá a pasarlo por alto de nuevo. Además, el programador, consciente o inconscientemente, deseará demostrar que su programa funciona, lo que no ayudará a encontrar los errores.

El resultado de cada prueba debe inspeccionarse a conciencia.

Al generar casos de prueba, hay que incluir tanto datos válidos como inválidos o inesperados.

Hay que probar que el software hace lo que debe hacer. Que el software no hace nada indebido. Es frecuente que se olvide realizarlo.

Hay que documentar los casos de prueba.

Asumir que siempre habrá algún defecto. EL 40% del desarrollo se emplea en pruebas y depuración.

No hay técnicas rutinarias y, al igual que en la programación, hay que recurrir casi siempre al ingenio para encontrar los fallos.

Técnicas de diseño de casos

Las técnicas de diseño de casos tendrán como objetivo conseguir una confianza aceptable en el producto.

Suelen elegirse para ello casos de prueba representativos, la elección aleatoria no suele dar buen resultado.

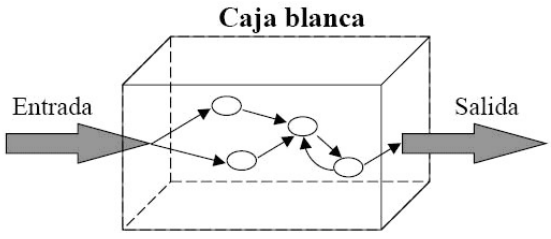
Existen dos enfoques:

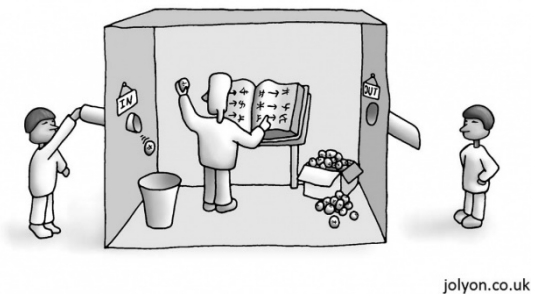
Estructural o de caja blanca, caja transparente o caja de cristal.

Enfoque funcional o de caja negra.

Ambos enfoques no son excluyentes.

CAJA BLANCA





El diseño de casos se basa en la elección de “caminos de ejecución” importantes que ofrezcan una seguridad aceptable de descubrir defectos. Se usan criterios de cobertura lógica para ello. Es decir, se trata de cubrir secciones del código fuente:

Cobertura de sentencias: intenta que se ejecuten todas las sentencias del programa al menos una vez.

Cobertura de decisiones: cada decisión debe resultar con un verdadero y un falso al menos una vez.

Cobertura de condiciones. Una decisión se compone de condiciones: if (a==3 && b==3) tiene dos condiciones. Se trata de que cada condición adopte sus dos posibles valores al menos una vez.

Criterios de decisión/condición: exige que se cumplan los dos puntos anteriores.

Cobertura de caminos: Se define "camino" como una secuencia encadenada de sentencias desde el inicio al final de la ejecución. La cobertura de caminos es el criterio más elevado y consiste en ejecutar cada posible camino al menos una vez.

CAJA NEGRA:

Utiliza las siguientes reglas para generar casos de prueba:

Cada caso debe cubrir el máximo número de valores distintos de los datos de entrada.

Debe dividirse el conjunto de datos de entrada en clases de equivalencia de modo que la prueba de un caso representativo de una clase permita suponer razonablemente que todos los datos de esa clase no testeados van a funcionar.

Pasos:

Identificar las condiciones que se exigen para los datos de entrada, restricciones de formato o de valor de los mismos.

Tras lo anterior, separar en clases y por grupos de:

Datos válidos

Datos no válidos.

Veremos cómo identificar las clases de equivalencia y cómo crear esos casos de prueba.

Si se especifica un rango de valores de entrada, "el número debe estar entre 1 y 49", se creará una clase válida 1<= n <= 49 y otras dos clases no válidas <1 y >49.

Si se especifica un número de valores: “un piso puede tener hasta tres propietarios”, se creará una clase válida “propietarios de 1 a 3”. Y dos no válidas, “no hay propietarios” y “hay más de tres”.

Una situación “x debe ser...”, o booleana, provocará dos clases. EJ: “el CIF contiene una letra al final” genera dos clases “CIF con letra al final” y “CIF sin letra al final”.

Si hay valores de conjunto, “los colores de un semáforo son rojo amarillo y verde”, se genera una clase válida por cada valor y una no válida con cualquier otro distinto de los anteriores.

Si una vez creadas una clase se sospecha que el tratamiento va a ser diferente para determinados casos, hay que subdividir esa clase según esos casos.

Una vez determinadas las clases:

Se asigna un número único a cada clase.

Se crea un caso que cubra el mayor número posible de clases válidas y se repite el proceso hasta cubrir todas las clases válidas.

Se crea un caso para cada clase no válida. Se hace así porque un caso erróneo podría enmascarar otros casos erróneos y no ser estos últimos reparados en la depuración posterior.

Análisis de valores límite

Por experiencia se ha constatado que los casos que exploran condiciones límite de un programa producen mejores resultados para detectar defectos.

Las condiciones límite son aquellas que se encuentran arriba, bajo y en los márgenes de las clases de equivalencia.

En lugar de elegir un representante de cada clase, se elige más de uno, probando valores cercanos a los límites, los límites mismos y valores inválidos cercanos a los límites.

Los valores límite se aplican a valores de entrada y de salida. Se deben escribir casos de entrada que provoquen salidas límite.

Pruebas unitarias, de integración y de sistema

PRUEBAS UNITARIAS

Una unidad de prueba es uno o más módulos que cumplen que:

Son todos del mismo programa

Al menos uno de ellos no ha sido probado

El conjunto de módulos es objeto de una prueba

Las pruebas unitarias suelen diseñarse con casos de caja blanca. Las pruebas las hacen desarrolladores, pero distintos del programador del módulo.

PRUEBAS DE INTEGRACIÓN

Presuponen que diversos módulos han sido probados ya. Tratan de probar que esos módulos interactúan y funcionan de modo correcto conjuntamente. Está pues orientado a la prueba de los interfaces (flujos de datos) entre módulos.

Suele especificarse si la prueba será total o incremental y el orden de codificación y prueba de cada módulo: desde la raíz hacia las hojas o al revés. Se detalla también el coste de las pruebas.

Se trata de probar integración hardware y software para ver si cumple los requisitos, es decir:

Requisitos funcionales.

Rendimiento de interfaces hardware, software, de usuario y de operador.

Adecuación de la documentación al usuario.

Ejecución en condiciones límite y de sobrecarga.

Suelen probarse.

Casos de caja negra aplicadas a las especificaciones.

Casos necesarios para pruebas de rendimiento y capacidad: grandes volúmenes de datos y de transacciones. *Stress testing*.

Caja blanca en casuísticas de alto nivel, por ejemplo las especificadas en diagramas de flujo de datos de alto nivel.

PRUEBAS DE ACEPTACIÓN

El usuario realizará pruebas para evaluar la fiabilidad y facilidad de uso del software, esta prueba se llama prueba de aceptación y tiene como objetivo transmitir confianza al usuario en el producto y convencerle de que está listo para su uso. El usuario debe ser asesorado por un profesional, no debe probar él solo. En proyectos que reemplazan un producto antiguo, pueden establecerse periodos de funcionamiento en paralelo de ambos sistemas, para permitir comparar ambos y ver si son exactos en sus resultados.

Los criterios de aceptación estarán firmados previamente por el usuario.

En ocasiones se liberan versiones beta para prueba por usuarios de confianza distintos de los usuarios finales.

2.5 Documentación

Todas las etapas del desarrollo deben quedar perfectamente documentadas. En esta etapa será necesario reunir todos los documentos generados y clasificarlos según el nivel técnico de sus descripciones.

Los documentos relacionados con un proyecto de software:

Deben actuar como un medio de comunicación entre los miembros del equipo de desarrollo.

Deben ser un repositorio de información del sisten1a para ser utilizado por el personal de mantenimiento.

Deben proporcionar información para ayudar a planificar la gestión del presupuesto y programar el proceso del desarrollo del software.

Algunos de los documentos deben indicar a los usuarios cómo utilizar y administrar el sistema.

Por lo general se puede decir que la documentación presentada se divide en dos clases:

La documentación del proceso. Estos documentos registran el proceso de desarrollo y mantenimiento. Son documentos en los que se indican planes, estimaciones y horarios que se utilizan para predecir y controlar el proceso de software, que informan sobre cómo usar los recursos durante el proceso de desarrollo, sobre normas de cómo se ha de implementar el proceso.

La documentación del producto. Esta documentación describe el producto que está siendo desarrollado. Define dos tipos de documentación: la documentación del sistema que describe el producto desde un punto de vista técnico, orientado al desarrollo y mantenimiento del mismo; y la documentación del usuario que ofrece una descripción del producto orientada a los usuarios que utilizarán el sistema.

La documentación del proceso se utiliza para gestionar todo el proceso de desarrollo del software. La documentación del producto se utiliza después de que el sistema está en funcionamiento, aunque también es esencial para la gestión del desarrollo del sistema.

2.5.1 Documentación del usuario

Los usuarios de un sistema no son todos iguales. Hay que distinguir entre los usuarios finales y los usuarios administradores del sistema.

Los usuarios finales utilizan el software para realizar alguna tarea, quieren saber cómo el software les ayuda a realizar su tarea, no están interesados en el equipo informático o en los detalles del programa. Los administradores del sistema son responsables de la gestión de los programas informáticos utilizados por los usuarios finales, pueden actuar como un gestor de red si el sistema implica una red de estaciones de trabajo o como un técnico que arregla problemas de software de los usuarios finales y que sirve de enlace entre los usuarios y el proveedor de software.

Para atender a los distintos tipos de usuarios con diferentes niveles de experiencia, se definen una serie de documentos que deben ser entregados con el sistema de software (aunque se pueden entregar en el mismo documento):

Evaluadores del sistema:

Descripción funcional del sistema.

Descripción de los servicios ofrecidos.

Administrador del sistema:

Documentación de instalación del sistema.

Guía de administración del sistema.

Usuarios noveles:

Manual introductorio.

Usuarios experimentados:

Manual de referencia.

Tarjeta de referencia rápida.

2.5.2 Documentación del sistema

La documentación del sistema incluye los documentos que describen el sistema, desde la especificación de requisitos hasta las pruebas de aceptación. Los documentos que describen el diseño, la implementación y las pruebas del sistema son esenciales para entender y mantener el software.

En los grandes sistemas de software la documentación debe incluir:

Fundamentos del sistema.

El análisis y especificación de requisitos.

Diseño.

Implementación.

Plan de pruebas del sistema.

Plan de pruebas de aceptación

Los diccionarios de datos.

2.5 Explotación

Una vez que se han realizado todas las pruebas y documentado todas las etapas, se pasa a la explotación del sistema. En esta etapa se lleva a cabo la instalación y puesta en marcha del producto software en el entorno de trabajo del cliente.

En esta etapa se llevan a cabo las siguientes tareas:

Se define la estrategia para la implementación del proceso. Se desarrolla un plan donde se establecen las normas para la realización de las actividades y tareas de este proceso. Se definen los procedimientos para recibir, registrar, solucionar, hacer un seguimiento de los problemas y para probar el producto software en el entorno de trabajo.

Pruebas de operación. Para cada *release* del producto software, se llevarán a cabo pruebas de funcionamiento y tras satisfacerse los criterios especificados, se libera el software para uso operativo.

Uso operacional del sistema. El sistema entrará en acción en el entorno previsto de acuerdo con la documentación de usuario.

Soporte al usuario. Se deberá proporcionar asistencia y consultoría a los usuarios cuando la soliciten. Estas peticiones y las acciones subsecuentes se deberán registrar y supervisar.

2.6 Mantenimiento

El mantenimiento del software se define (Norma IEEE 1219 para Mantenimiento de Software) como *la modificación de un producto de software después de la entrega para corregir los fallos, para mejorar el rendimiento u otros atributos, o para adaptar el producto a un entorno modificado*.

Existen cuatro tipos de mantenimiento del software que dependen de las demandas de los usuarios del producto software a mantener

Mantenimiento adaptativo. Con el paso del tiempo es posible que se produzcan cambios en el entorno original (CPU, sistema operativo, reglas de la empresa, etc.) para el que se desarrolló el software. El mantenimiento adaptativo tiene como objetivo la modificación del producto por los cambios que se produzcan, tanto en el hardware como en el software del entorno en el que se ejecuta. Este tipo de mantenimiento es el más usual debido a los rápidos cambios que se producen en la tecnología informática, que en la mayoría de ocasiones dejan obsoletos los productos software desarrollados.

Mantenimiento correctivo. Es muy probable que después de la entrega del producto, el cliente encuentre errores o defectos, a pesar de las pruebas y verificaciones realizadas en las etapas anteriores. Este tipo de mantenimiento tiene como objetivo corregir los fallos descubiertos.

Mantenimiento perfectivo. Conforme el cliente utiliza el software puede descubrir funciones adicionales que le pueden aportar beneficios. El mantenimiento perfectivo es la modificación del producto de software orientado a incorporar nuevas funcionalidades (más allá de los requisitos funcionales originales) y nuevas mejoras en el rendimiento o la mantenibilidad del producto.

Mantenimiento preventivo. Consiste en la modificación del producto de software sin alterar las especificaciones del mismo, con el fin de mejorar y facilitar las tareas de mantenimiento. Este tipo de mantenimiento hace cambios en programas con el fin de que se puedan corregir, adaptar y mejorar más fácilmente, por ejemplo, se pueden reestructurar los programas para mejorar su legibilidad o añadir nuevos comentarios que faciliten la comprensión del mismo. A este mantenimiento también se le llama reingeniería del software.

3. Modelos del ciclo de vida del software

Diversos autores han planteado distintos modelos de ciclos de vida, pero los más conocidos y utilizados son los que aparecen a continuación:

Siempre se debe aplicar un modelo de ciclo de vida al desarrollo de cualquier proyecto software.

3.1 Modelo en Cascada

Es el modelo de vida clásico del software.

Es prácticamente imposible que se pueda utilizar, ya que requiere conocer de antemano todos los requisitos del sistema. Sólo es aplicable a pequeños desarrollos, ya que las etapas pasan de una a otra sin retorno posible. (se presupone que no habrá errores ni variaciones del software).

Características

Requiere conocimiento previo y absoluto de los requerimientos del sistema.

No hay retorno en las etapas: si hay algún error durante el proceso hay que empezar desde el principio.

No permite modificaciones ni actualizaciones del software.

Es un modelo utópico

3.2 Modelo en Cascada con Realimentación

Es uno de los modelos más utilizados. Proviene del modelo anterior, pero se introduce una realimentación entre etapas, de forma que podamos volver atrás en cualquier momento para corregir, modificar o depurar algún aspecto. No obstante, si se prevén muchos cambios durante el desarrollo no es el modelo más idóneo.

Es el modelo perfecto si el proyecto es rígido (pocos cambios, poco evolutivo) y los requisitos están claros.

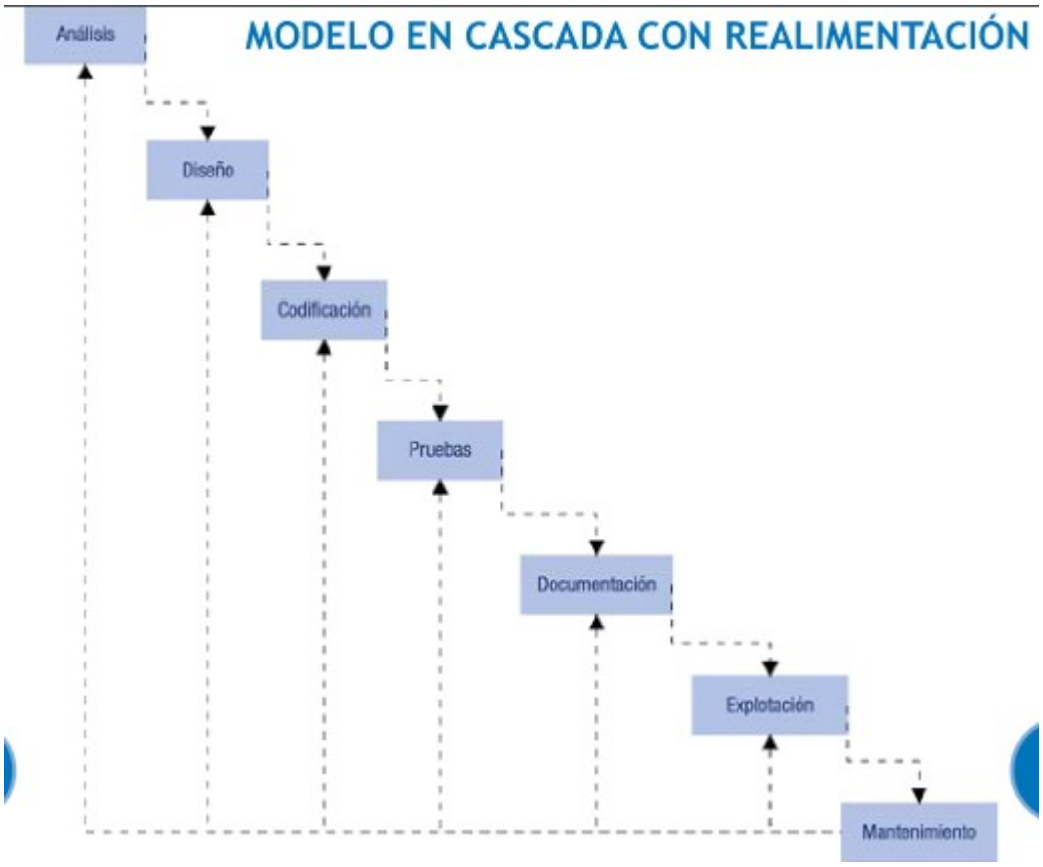
Características

Es uno de los modelos más utilizados.

Se puede retornar a etapas anteriores para introducir modificaciones o depurar errores.

Idóneo para proyectos más o menos rígidos y con requisitos claros.

Los errores detectados una vez concluido el proyecto pueden provocar que haya que empezar desde cero.



3.3 Modelo Diseño en V

Es el estándar utilizado para los proyectos de la Administración Federal Alemana y de defensa. Como está disponible públicamente muchas compañías lo usan.

Es una variante del paradigma de desarrollo en cascada, en la que se empieza a trabajar en las pruebas desde el comienzo del proyecto. La parte izquierda representa la definición y construcción del sistema. La parte derecha, la comprobación del mismo.

V = Verificación y Validación.

Describe los procesos y los resultados.

En la izquierda van las fases y en la derecha las validaciones necesarias.

Ventajas

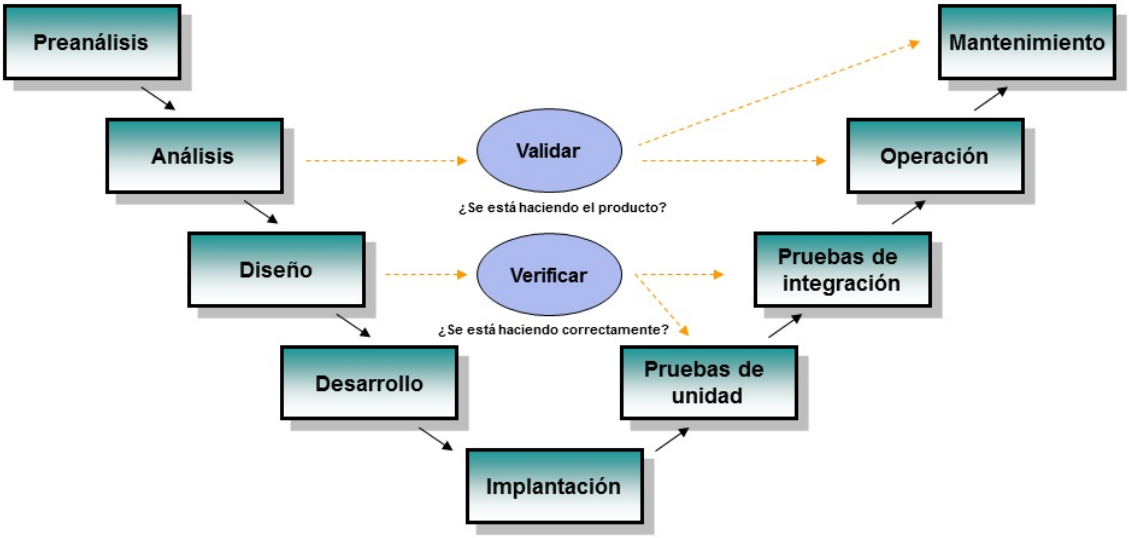
Ahorra tiempo a la par que permite detectar errores de diseño en etapas tempranas.

Los errores se corrigen en cada fase, es más difícil que se tenga que volver atrás.

Desventajas

Consume más recursos que el ciclo de desarrollo en cascada.

No se adapta bien a cambios de requisitos.



3.4 Modelos Evolutivos

Son más modernos que los anteriores. Tienen en cuenta la naturaleza cambiante y evolutiva del software.

Distinguimos tres variantes:

Modelo Iterativo Incremental

Está basado en el modelo en cascada con realimentación, donde las fases se repiten y refinan, y van propagando su mejora a las fases siguientes.

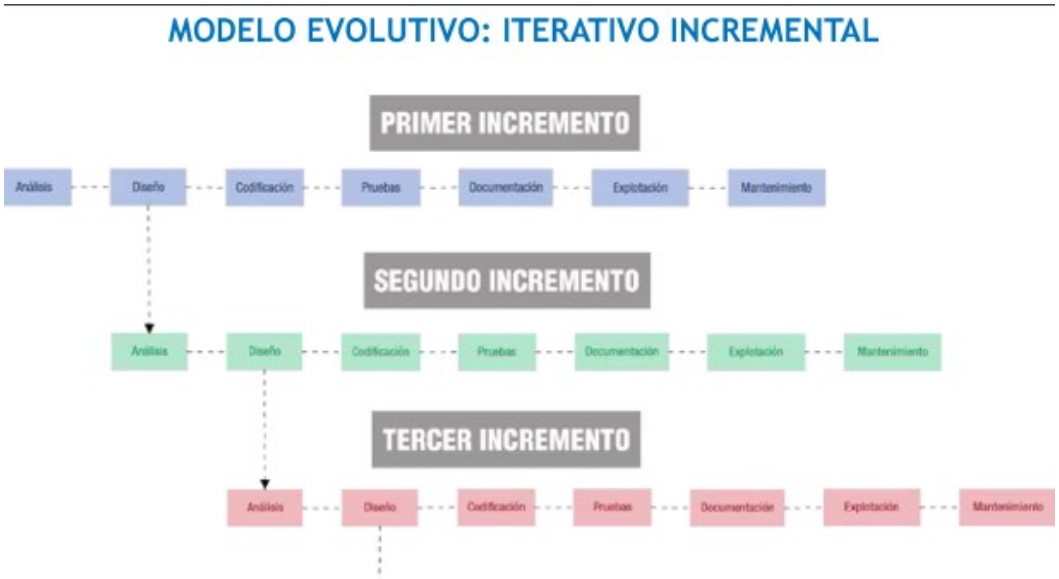
Características

Actualmente, no se ponen en el mercado los productos completos, sino versiones.

Permite una fácil evolución temporal.

Vemos que se trata de varios ciclos en cascada que se repiten y se refinan en cada incremento.

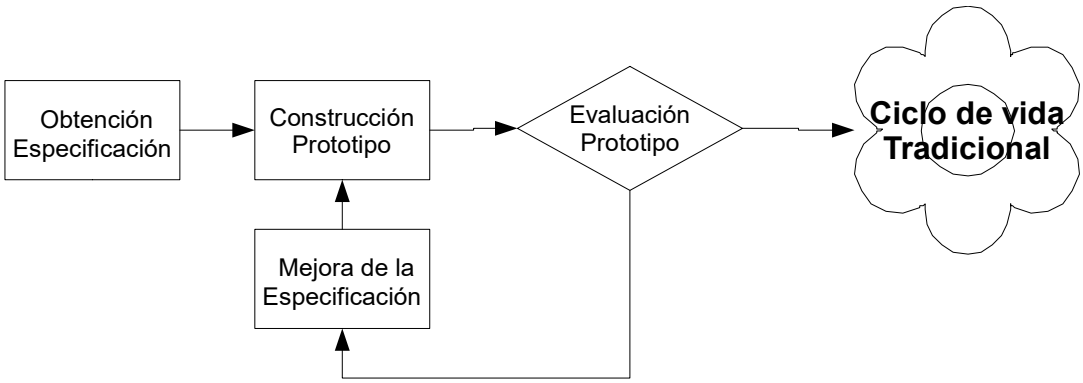
Las sucesivas versiones del producto son cada vez más completas hasta llegar al producto final.



Modelo de prototipos

En desarrollos complejos o innovadores, o que impliquen riesgo, las incertidumbres sobre lo que se puede alcanzar imposibilitan una planificación y desarrollo lineal del proyecto.

En esos casos, es más recomendable realizar una implementación parcial del producto a partir de unas especificaciones iniciales -> prototipo.



Ventajas

Apto para entornos de gran incertidumbre

Permite tener unos requisitos claros antes de comenzar el desarrollo

Desventajas

El desarrollo del prototipo es costoso

Es difícil determinar el esfuerzo de desarrollo antes de finalizar la fase de prototipado

Modelo en Espiral

Es una combinación del modelo anterior con el modelo en cascada. En él, el software se va construyendo repetidamente en forma de versiones que son cada vez mejores, debido a que incrementan la funcionalidad en cada versión. Es un modelo bastante complejo.

Características

Se divide en 6 zonas llamadas regiones de tareas: comunicación con el cliente, planificación, análisis de riesgos, representación de la aplicación, codificación y explotación y evaluación del cliente.

Se adapta completamente a la naturaleza evolutiva del software.

Reduce los riesgos antes de que sean problemáticos.

Ventajas

No necesita una definición completa de los requisitos para empezar a funcionar.

Al entregar productos desde el final de la primera iteración es más fácil validar los requisitos.

El riesgo en general es menor, porque si todo se hace mal, solo se ha perdido el tiempo y recursos invertidos en una iteración (las anteriores iteraciones están bien).

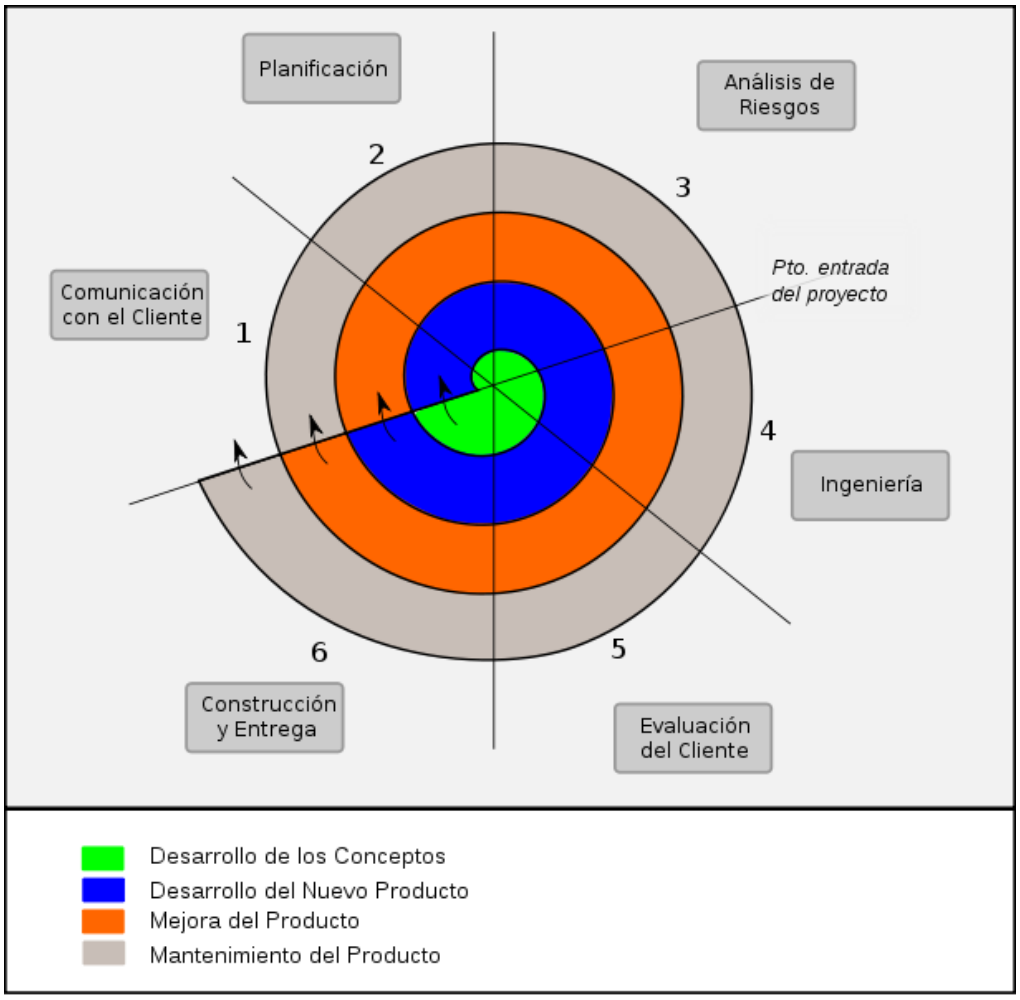
El riesgo de sufrir retrasos es menor, ya que al identificar los problemas en etapas tempranas hay tiempo de subsanarlos.

Desventajas

Es difícil evaluar los riesgos.

Necesita de la participación continua por parte del cliente.

Subcontratación complicada, ya que el nivel de definición no es muy alto.



En líneas generales, existen unos criterios más o menos aceptados, para ayudar a elegir el tipo de método a utilizar:

Si el problema perfectamente conocido, el usuario define claramente los requisitos, el equipo tiene amplia experiencia en la cuestión. Modelo en cascada.

Si el desarrollo conlleva muchos riesgos. Modelo en espiral.

Si es muy importante ir probando el producto a medida que se desarrolla, para mostrar al usuario y al cliente su utilidad. Modelo en prototipos.

Sin embargo, el problema en la vida real es que las situaciones se entremezclan.

Metodologías del desarrollo del software

Metodología es la manera sistemática de hacer cierta cosa. Una metodología de desarrollo de software define un estilo o una forma de guiar el proceso de desarrollo. Una metodología comprende el detalle de métodos, herramientas y procedimientos utilizados para el desarrollo.

Son una concreción de los ciclos de vida. Definen una estructura jerárquica de pasos a seguir. Dividen el proceso en partes, según el ciclo o ciclos de vida en los que se basen. Para cada una de esas partes, definen productos a obtener y técnicas concretas.

Existen muchas metodologías de desarrollo, debido a que cada una de ellas hace hincapié en determinados aspectos del desarrollo, o se adapta mejor a ciertos tipos de proyectos.

Sin embargo, existen unas metodologías generales y características que son las siguientes:

Análisis estructurado

DeMarco, 1979. Surgió como evolución de las técnicas de diseño estructurado. Parte de la crítica de las especificaciones monolíticas y textuales, y se basa más en métodos gráficos. Trata de descomponer el problema del desarrollo en partes más pequeñas y más abordables.

Sus principales características son:

Descomposición funcional del sistema: análisis descendente.

Representación del flujo de información.

Transformación de diagramas de flujo de datos en estructura modular de programa.

Construcción de modelos de datos.

Herramientas:

Diagrama de flujo de datos: la herramienta más importante.

Diccionario de datos: contiene las características lógicas de los almacenes de datos del sistema.

Diagrama de estructuras: diagrama que muestra la relación entre los diferentes módulos.

Ha sido evolucionado por diversos autores: Yourdon diagramas de estados, Woard y Mellor flujo de control. Se utiliza de forma habitual en la actualidad, con diversas adaptaciones.

Rational Unified Proccess

El análisis estructurado no se adapta bien a la programación orientada a objetos, puesto que trata datos y procesos por separado. Debido a esta carencia, varios autores desarrollaron métodos de análisis orientado a objetos, que convergieron en el Proceso Unificado de desarrollo. Es un marco general y adaptable del que la implementación más completa y extendida es el Rational Unified Process, ahora propiedad de IBM.

Sus características son:

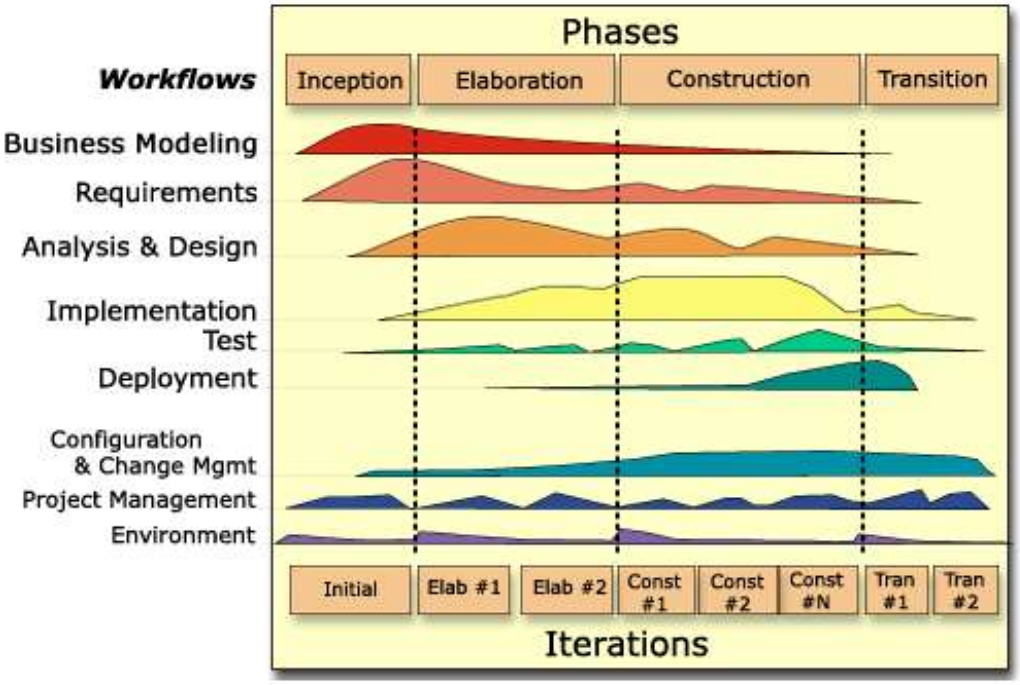
Proceso Dirigido por los Casos de Uso

Proceso Iterativo e Incremental

Proceso Centrado en la Arquitectura

Utiliza prototipos ejecutables que se enseñan al cliente

Utiliza la notación UML

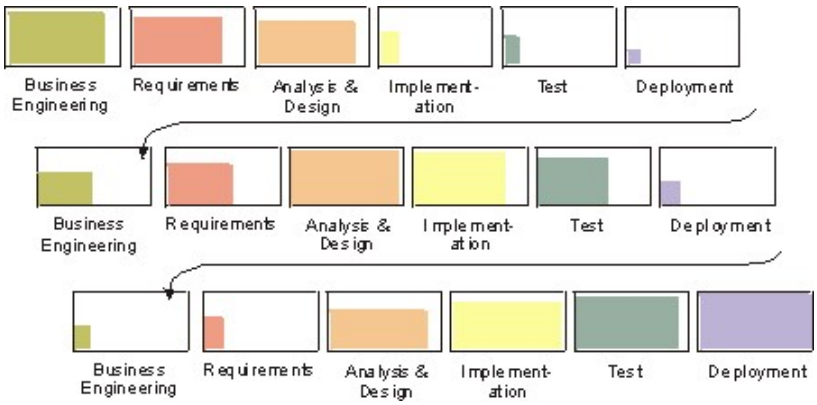


Elementos: artefactos o entregables, workers, y actividades.

Dos dimensiones: fases y disciplinas (unas principales y otros de apoyo).

Las actividades se encadenan en una mini-cascada con un alcance limitado por los objetivos de la iteración.

En cada una de las fases, se dedica un esfuerzo diferente a cada una de las disciplinas:



Metodologías Ágiles

Las anteriores metodologías son adecuadas para proyectos grandes. Sin embargo, introducen sobrecarga o no tienen la flexibilidad suficiente para proyectos pequeños. Para cubrir estas carencias surgieron las metodologías ágiles a partir de los valores del Manifiesto Ágil emitido en 2001 por la Agile Alliance. Estas metodologías se centran en el factor humano y en el producto software, más que en el proceso. Sus características principales son:

El usuario y los desarrolladores trabajan en el mismo equipo. El valor del grupo prevalece sobre el individual y en el que la gestión del cambio constituye el pilar fundamental sobre el que hacer pivotar su autogestión.

Se realizan entregas frecuentes e incrementales. Es un proceso iterativo e incremental.

Su premisa es poder revisar el desarrollo de forma frecuente, continuada e incremental.

Conseguir software que funcione, más que una documentación.

Simplicidad, buen diseño y calidad del código.

Beneficios

Gestionar periódicamente las expectativas del cliente, pudiendo adecuar el proyecto para asegurar el cumplimiento de sus necesidades.

El cliente puede contar con un producto funcionando de forma periódica e incremental.

Las metodologías ágiles más conocidas son:

SCRUM

Adaptative Software Development

Extreme Programming

METODOLOGÍAS DE LA UNIÓN EUROPEA

Las metodologías más extendidas en la Unión Europea, debido sobre todo por el apoyo de los organismos oficiales de ciertos países, son tres: Merise, de origen francés; SSADM, utilizado por la administración pública inglesa; y Métrica V3 en España.

Métrica V3

La metodología MÉTRICA Versión 3 es una iniciativa promovida por el Consejo Superior de Informática. Ha sido concebida para abarcar el desarrollo completo de Sistemas de Información sea cual sea su complejidad y magnitud, por lo cual su estructura y los perfiles de los participantes que intervienen deberán adaptarse y dimensionarse en cada momento de acuerdo a las características particulares de cada proyecto. Versión 3, año 2001.

Los elementos de métrica 3 son los siguientes:

1. Procesos. Tres procesos principales: Planificación, Desarrollo, Mantenimiento. Se dividen en procesos, y cada uno de ellos en actividades y tareas. Por ejemplo el proceso de desarrollo se divide en:

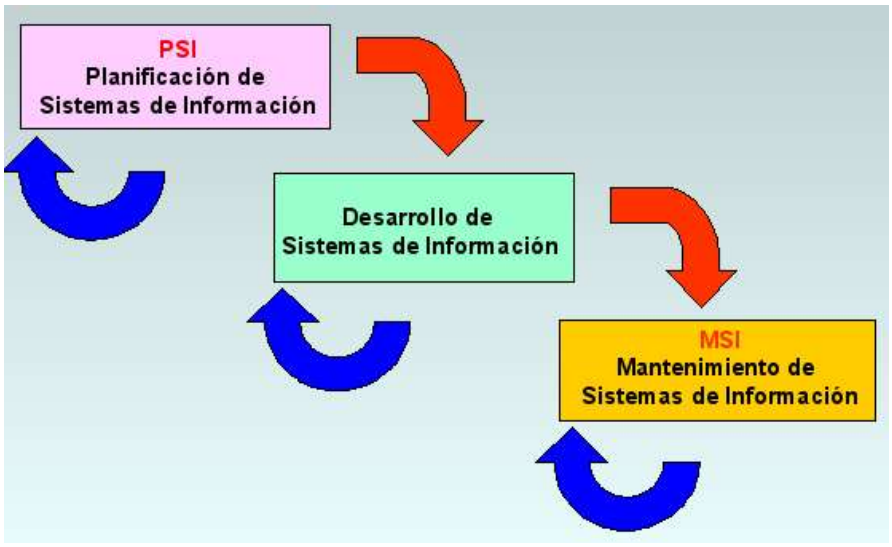
Estudio de la viabilidad del sistema (EVS)

Análisis del sistema de información (ASI)

Diseño del sistema de información (DSI)

Construcción del sistema de información (CSI)

Implantación y aceptación del sistema (IAS)



2. Interfaces: actividades de tipo organizativo que sirven de soporte (gestión de proyectos, calidad, seguridad, gestión de la configuración). Se dividen también en actividades y tareas.

3. Productos: entrada y salida de las tareas. Ejemplo: ASI tiene de entrada el estudio de viabilidad y los estándares de la instalación, y como salida la especificación detallada del sistema de información.

4. Técnicas y Prácticas: conjunto de procedimientos específicos utilizados en las tareas. Son variadas, tanto para estructurado como para objetos: DFD, diagramas UML, diagramas de estructura, PERT, etc…

5. Participantes: personas afectadas por el proyecto. Se clasifican en perfiles: usuario, jefe de proyecto, analista, programador, etc…

Sirve tanto para desarrollos con enfoque de análisis estructurado como con enfoque OO.

Merise

Surge en Francia a final de los años setenta. Sus principales características son:

Desglose en etapas: estudio preliminar, detallado, realización, puesta en marcha.

Definición de los documentos estándar de cada una.

Estudio separado de datos y tratamientos.

Uso de E/R para la representación de datos.

Uso de los diagramas de encadenamiento de procedimientos.

Reparto de tareas entre roles.

Definición de los flujos de información entre las unidades del sistema.

El sistema se contempla desde diferentes niveles de abstracción y esto da lugar a una descripción del mismo a tres niveles: conceptual, lógico u organizativo y físico.

En la fase de concepción se trabaja básicamente sobre dos elementos: datos y tratamientos. La descripción de los datos reflejará la información existente en el entorno y las relaciones entre ellos. La representación de los tratamientos reflejará los procesos a realizar con los datos así como su secuencia en el tiempo.

Con la descripción de estos dos elementos habremos conseguido reflejar tanto el contenido del sistema como su funcionamiento.

Se podría establecer el siguiente cuadro en cuanto a los diferentes niveles, tanto de decisión como de descripción de datos y tratamientos:

Niveles de decisión	Nivel de descripción	
	Tratamientos	Datos
Conceptual	Conceptual	Conceptual
Organizativo	Organizativo	Lógico
Técnico	Operativo	Físico

Veremos en primer lugar lo que significan estos niveles y cuál es su contenido. Empecemos por los tratamientos.

En cuanto a la descripción de tratamientos, el nivel conceptual consiste en la descripción del qué hay que hacer, es decir, en la descripción en términos de operaciones y resultados de la gestión que debe resolver el sistema independientemente de quién sea el que la realice, hombre o máquina, y de qué modo.

En el nivel organizativo, se desglosan las operaciones descritas a escala conceptual en procedimientos funcionales, es decir, en tareas realizadas sucesivamente en un puesto de trabajo. A este nivel se concreta ya quién, cuándo y dónde se han de realizar estas tareas.

En el nivel operativo se responde a la pregunta de cómo hay que hacer las cosas. Se detallan, tanto para procedimientos automatizados como para los manuales, las normas para realizarlos correctamente.

Veamos a continuación como se estructura la descripción de los datos desde diferentes niveles de abstracción.

En la descripción de datos contemplamos en primer lugar un nivel conceptual, en el que se observa la información del sistema en términos de objetos o entidades, se describen sus propiedades, la información de cada uno de ellos y las relaciones entre los mismos. Este modelo conceptual de datos es, en principio, bastante estable a lo largo del ciclo de vida del sistema.

En el nivel lógico se traduce el modelo conceptual en agrupaciones o estructuras lógicas de datos para su tratamiento por el sistema. Todavía este nuevo modelo debe ser independiente de la opción técnica a elegir en cuanto a soporte software del mismo.

En el nivel más bajo de la descripción de datos, es decir, en el nivel físico, se concreta ya cual va a ser la estructura final de los datos de acuerdo al sistema gestor elegido (base de datos, tipos de ficheros, etc.). Una vez conocido esto se podrán hacer optimizaciones del modelo lógico para mejorar rendimientos.

SSADM

Las siglas son de: Sructured System Analysis and Design Method. Creado en los años ochenta para la administración pública en Gran Bretaña. Basado en el ciclo de vida en cascada.

Se divide en fases, etapas y tareas. Las fases son:

Viabilidad

Análisis

Diseño

Utiliza DFD, Entidad Relación, y modelo de eventos.

Ha ido evolucionando y está vigente en la actualidad.