

Unidad didáctica 1. Actividad 1. Proyectos de desarrollo software

Ciclo Superior Desarrollo de aplicaciones multiplataforma
IES Muralla Romana



Unidad didáctica 1. Actividad 1. Índice

Ciclo Superior Desarrollo de aplicaciones multiplataforma
IES Muralla Romana

Índice

1. Software.
2. Clasificación software.
3. Desarrollo de software.
4. Paradigma del ciclo de vida.
 - 4.1 Modelo en cascada.
 - 4.2 Modelo evolutivo.
 - 4.2.1 Modelo en espiral.
 - 4.2.2 Modelo incremental.
5. Metodologías.
 - 5.1 Metodologías ágiles.
 - 5.1.1 Programación extrema.
 - 5.1.2 Scrum.
 - 5.2 Métrica v.3.
6. Bibliografía.



Unidad didáctica 1. Actividad 1.

1.- Software

Ciclo Superior Desarrollo de aplicaciones multiplataforma
IES Muralla Romana

1.- Software

- Según la Real Academia Gallega, informática es la Ciencia del **tratamiento automático de la información** por medio de máquinas electrónicas.
- Este tratamiento necesita de:
 - Soporte físico o hardware** por todos los componentes electrónicos tangibles involucrados en el tratamiento. RAG: componentes del ordenador.
 - Soporte lógico o software** que hace que el hardware funcione, y está formado por todos los elementos intangibles involucrados en el tratamiento: programas, datos y documentación. RAG: conjunto de órdenes y programas que permiten utilizar el ordenador.

1.- Software

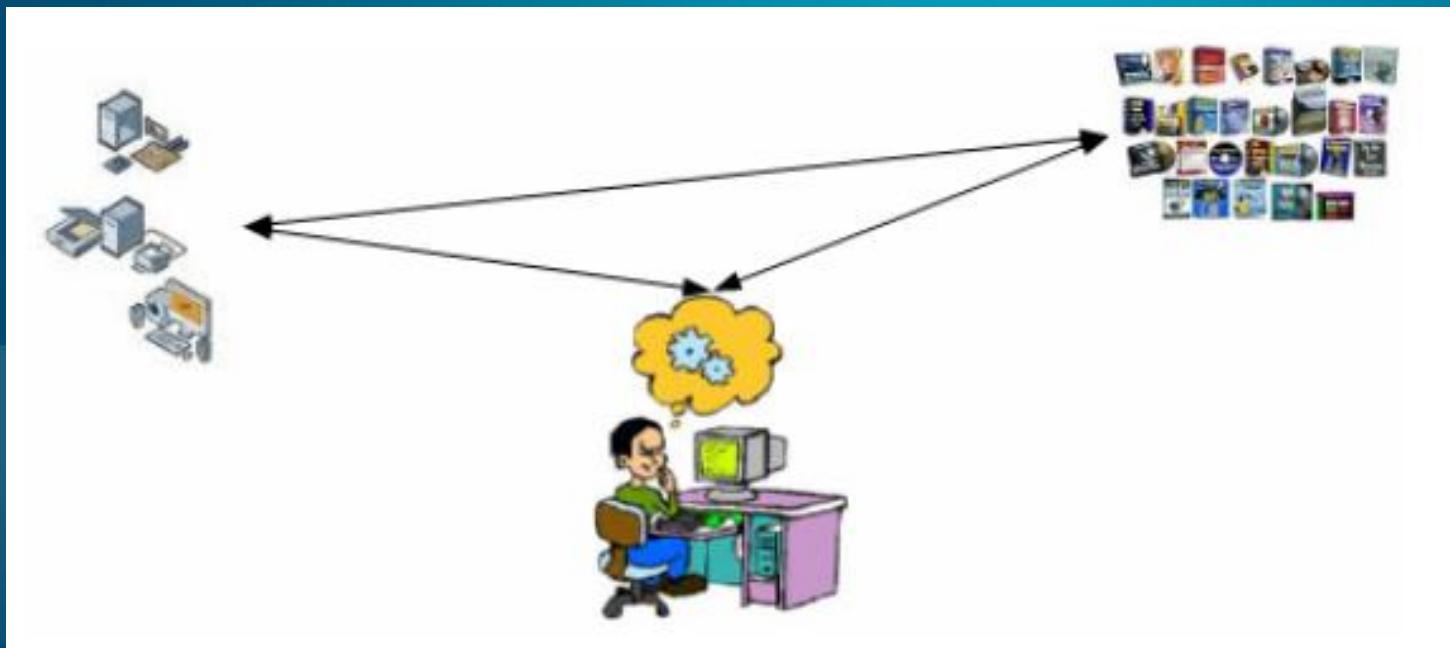
- El Software:
 - **Instrucciones** que, cuando se ejecutan, proporcionan la funcionalidad deseada.
 - **Estructuras de datos** que facilitan a las instrucciones manipular adecuadamente la información.
 - **Documentos** que describen el desarrollo, uso, instalación y mantenimiento de los programas.
- Software: "*programas de computador, procedimientos, y, posiblemente, la documentación asociada y los datos pertenecientes a las operaciones de un sistema de computación*".
- Incluye: **Formación, soporte al consumidor e instalación.**

1.- Software - características

- Elemento lógico, no físico.
- Desarrollado, no ‘fabricado’.
- No se ‘estropea’, ¡se deteriora!
(deterioro por ‘cambios’)
- Mayoritariamente *cerrado*:
usar todo o nada
(poco ensamblaje de componentes: reutilización--)

1.- Software

- Equipamiento humano o personal informático que maneja el equipo físico y el lógico para que todo el tratamiento se lleve a cabo.



1.- Software

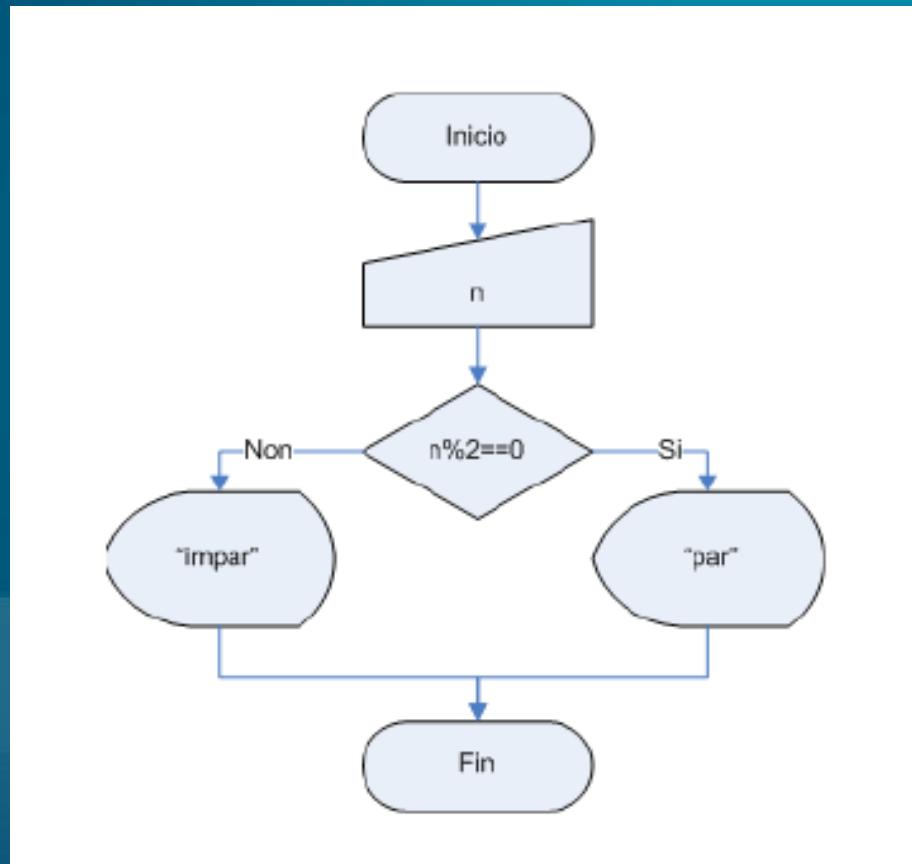
El concepto de programa informático está muy ligado al concepto de algoritmo.

¿Qué es un algoritmo?

- Conjunto de instrucciones o reglas bien definidas, ordenadas y finitas que permiten resolver un problema.

1.- Software

Diagrama de flujo de un algoritmo que permite ver si un número es par o impar. El número 0 se considera par.



1.- Software

Un algoritmo puede escribirse en un lenguaje de programación utilizando alguna herramienta de edición y deberá ser grabado en una memoria externa no volátil para que perdure en el tiempo.

```
package parimpar;

import java.util.Scanner;

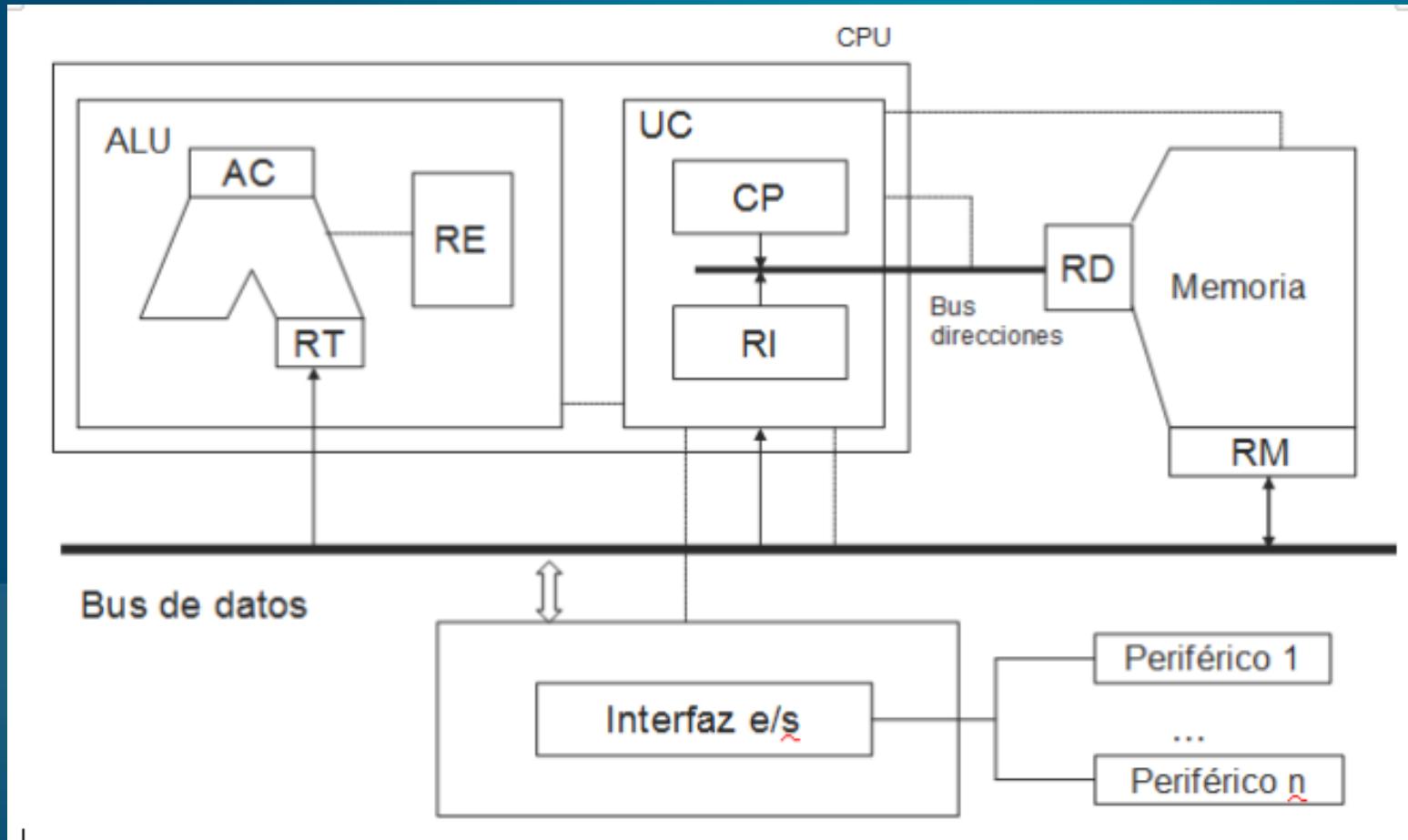
public class Main {
    public static void main(String[] args) {
        short n;
        Scanner teclado = new Scanner(System.in);
        System.out.printf("Teclee o número enteiro entre %d e %d:",
        Short.MIN_VALUE, Short.MAX_VALUE);
        n = teclado.nextShort();
        if(n%2==0) {
            System.out.printf("%d é par\n",n);
        }
        else{
            System.out.printf("%d é impar\n",n);
        }
    }
}
```

1.- Software

- El código sufre una serie de **transformaciones** para que finalmente se obtenga el programa que pueda ser ejecutado en un ordenador.
- En la ejecución es preciso que el programa esté almacenado en memoria interna y volátil del ordenador, así el procesador irá leyendo cada orden del programa, resolviéndola y ejecutándola.
- Si es necesario, se accede a memoria interna para manipular variables, a periféricos de entrada, salida o almacenamiento, hace los cálculos necesarios y resuelve las expresiones lógicas, hasta que finaliza con éxito la última instrucción u ocurre un error.

1.- Software

- Arquitectura de un ordenador



1.- Software

- En la ejecución del programa del código anterior, el procesador necesitaría:
 - Leer un número entero a través de teclado (dispositivo/periférico de entrada).
 - Realizar la operación aritmética (resto de la división entera de n entre 2).
 - Resolver la expresión lógica (resto 0).
 - Ejecutar la instrucción alternativa para que en el caso de que el resto sea 0 salga por pantalla (dispositivo/periférico de salida) que n es par o impar.

1.- Software

- Ejemplo ejecución código anterior y aspecto del resultado por pantalla cuando n=11

```
run:  
Teclee o número entero entre -32768 e 32767:11  
11 é impar  
BUILD SUCCESSFUL (total time: 7 seconds)
```

1.- Software

- Década 50-60:
 - “Software como un añadido”.
 - Desarrollo artesanal, a medida.
 - Lenguajes de bajo nivel.
- Década 60-70:
 - Software como producto.
 - Década lenguajes y compilación.
 - “Crisis del software”.
- Década 70-80:
 - Programación estructurada.
 - **Ingeniería del Software.**
 - Primeros métodos estructurados.
- Década 80-90:
 - Tecnología de SGBDs, SOs...
 - Nuevos paradigmas de programación y de producción de programas:
- 90's - actualidad:
 - Análisis/Diseño OO.
 - Tecnología CASE
 - Componentes y reutilización
 - Interoperabilidad (CORBA, .NET...)
 - Internet
 - ISw. distribuida
 - repositorios de componentes reutilizables
 - e-business; e-commerce
 - ...

1.- Software – principios de ingeniería

- **Abstracción**
 - Permite parcelar la complejidad. Por ello se olvidan aspectos irrelevantes del sistema y se potencian los fundamentales.
- **Encapsulamiento u Ocultación de la información**
 - Esconder todos los detalles que no afecten a otros módulos, definiendo interfaces estrictos que sirvan de interacción entre los distintos modelos.
- **Modularidad**
 - Sirve para parcelar la solución en módulos independientes con fuerte cohesión interna.
- **Localización**
 - Deben estar agrupados todos aquellos elementos que están afectados por un mismo hecho.
- **Uniformidad**
 - Tódos los módulos deben tener una notación similar.
- **Completitud**
 - Deben estar desarrollados todos los aspectos del sistema.
- **Validación y Verificabilidad**
 - El producto final debe ser fácilmente validable y verificable:
 - ¿Estamos desarrollando el programa correcto?
 - ¿Estamos desarrollando correctamente el programa?



Unidad didáctica 1. Actividad 1.

2.- Clasificación del software

Ciclo Superior Desarrollo de aplicaciones multiplataforma
IES Muralla Romana

2.- Clasificación software

- Se puede clasificar en dos tipos:
 - Software de sistema.
 - Software de aplicación.

2.- Clasificación software

- Software de sistema

Controla y gestiona el hardware haciendo que funcione, ocultando al personal informático los procesos internos del funcionamiento.

Ejemplos

- Sistemas operativos.
- Controladores de dispositivos.
- Herramientas de diagnóstico.
- Servidores (correo, web, DNS...).
- Utilidades del sistema (compresión de información, rastreo de zonas defectuosas en soportes, etc.).

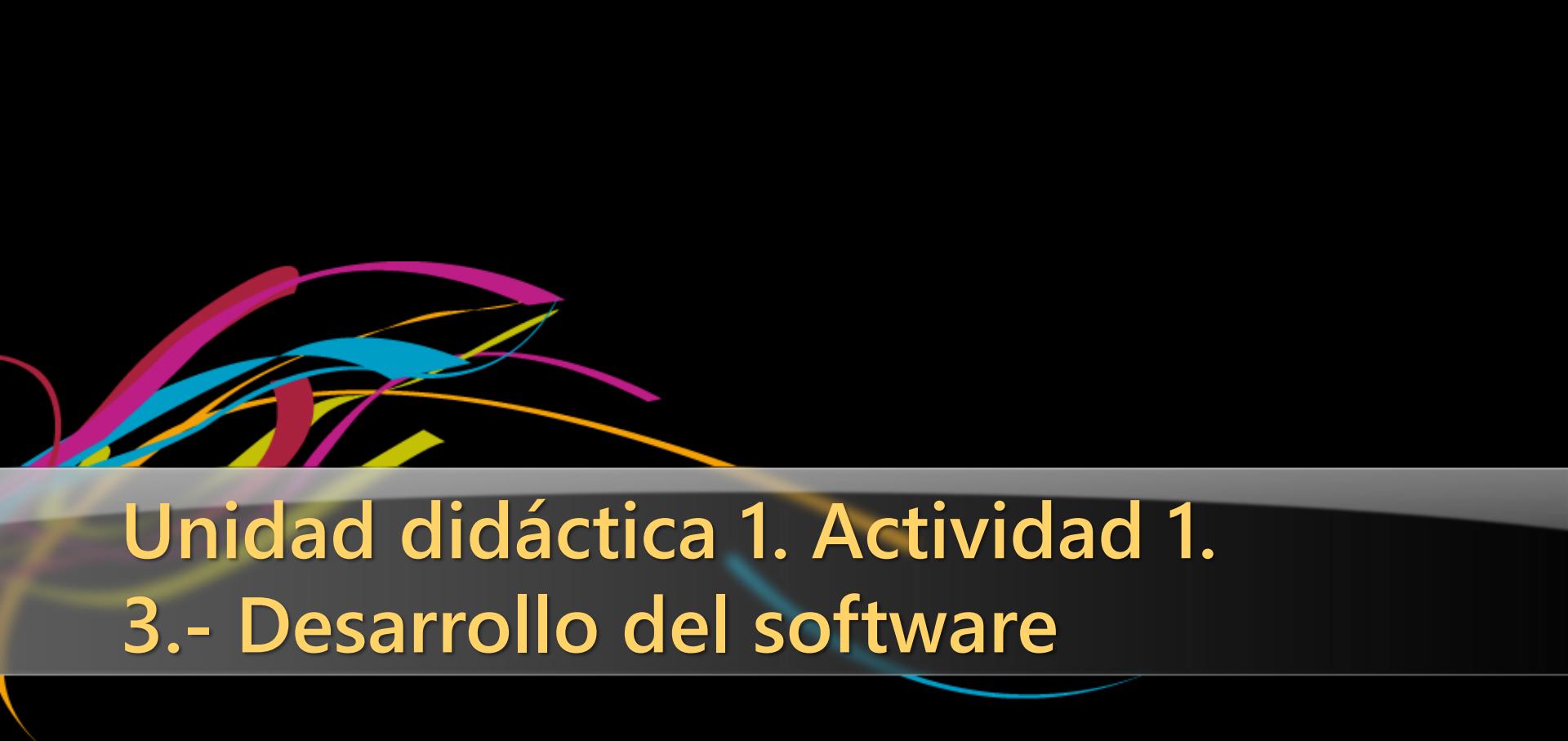
2.- Clasificación software

- Software de aplicación

- Software médico.
- Software de cálculo numérico.
- Software de diseño asistido por ordenador.
- Software de programación: conjunto de herramientas que permiten al programador desarrollar programas informáticos como:
 - Editores de texto.
 - Compiladores.
 - Intérpretes.
 - Enlazadores.
 - Depuradores.
 - Entornos de desarrollo integrado (IDE) que agrupa las anteriores herramientas en un mismo entorno facilitando al programador desarrollar programas informáticos y que tengan una avanzada GUI.

2.- Clasificación software

- El Software de aplicación se puede dividir en:
 - Software horizontal o genérico: se pueden utilizar en diversos entornos como por ejemplo las aplicaciones ofimáticas.
 - Software vertical o a medida: sólo se pueden utilizar en un entorno específico como por ejemplo la contabilidad hecha a medida para una empresa en particular.



Unidad didáctica 1. Actividad 1.

3.- Desarrollo del software

Ciclo Superior Desarrollo de aplicaciones multiplataforma
IES Muralla Romana

3.- Desarrollo de software

- El proceso de desarrollo de software dependerá de la complejidad del software.
- Para el desarrollo de un sistema operativo es necesario un equipo disciplinado, recursos, herramientas, un proyecto a seguir y alguien que gestione todo.
- En el extremo opuesto, para hacer un programa que visualice si un número es par o impar sólo es necesario un programador y un algoritmo de resolución.

3.- Desarrollo de software - Problemas

(COMUNES)

- Proyectos fuera de plazo y de presupuesto
- Excesiva dependencia de los desarrolladores Cuando los proyectos dependen de un desarrollador específico
- Falta de control del desarrollo del proyecto
- Escasa integración de las diferentes fases del desarrollo
- Escaso control de calidad del producto
- Escasa documentación actualizada de los proyectos
- No utilizar una metodología formal

3.- Desarrollo de software

- Se fueron desarrollando y mejorando paradigmas (métodos, herramientas y procedimientos para describir un modelo) que fueron sistemáticos, predecibles y repetibles, y así mejorar la productividad y calidad del software.
- La ingeniería del software es imprescindible en grandes proyectos de software, debe ser aplicada en proyectos de tamaño medio, y se recomienda aplicar alguno de sus procesos en proyectos pequeños.

3.- Desarrollo de software

I

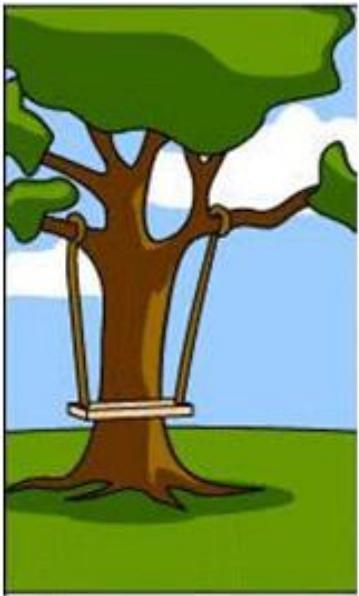


Lo que el cliente nos cuenta



Lo que entienden los
analistas

3.- Desarrollo de software



Producto resultante de la
fase de diseño

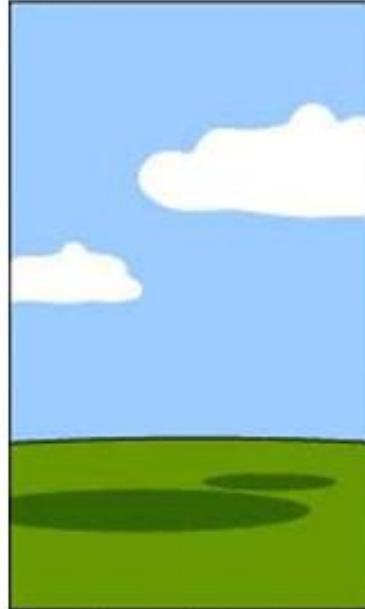


Producto final que vendió el
comercial

3.- Desarrollo de software

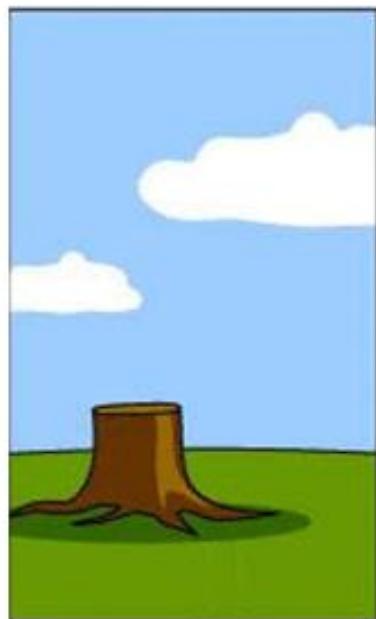


Producto final desarrollado
por los programadores



Documentación del proyecto

3.- Desarrollo de software



Servicio técnico a servicio
del cliente



Lo que se le cobra al cliente

3.- Desarrollo de software



Lo que el cliente quería....

3.- Desarrollo de software – proceso resolución

1. Identificar el problema
2. Definir y representar el problema
3. Explorar las posibles estrategias
4. Aplicar y mejorar las estrategias
5. Mirar atrás y evaluar los efectos de la actividad realizada (¿se ha resuelto el problema?)

3.- Desarrollo de software – proceso resolución

- Decidir **qué** hacer (cuál es el problema)
- Decidir **cómo** hacerlo
- **Hacerlo**
- **Probar** el resultado
- **Usar** el resultado

3.- Desarrollo de software – proceso resolución



3.- Desarrollo de software – proceso resolución

- 1. Qué hacer (Especificación de Requisitos y Análisis)**
- 2. Cómo hacerlo (Diseño del Sistema Software)**
- 3. Hacerlo (Codificación)**
- 4. Probar el resultado**
- 5. Usar el resultado (Instalado)**
- 6. Mantenimiento**

3.- Desarrollo de software – conceptos

- Cliente – usuario - stakeholder
- Producto: lo que se obtiene
- Proyecto: la pauta a seguir para desarrollar un producto
- Proceso: la pauta a seguir para desarrollar un proyecto

3.- Desarrollo de software – conceptos

Producto

Software y documentación asociada

3.- Desarrollo de software – conceptos

Proyecto Software

Un proyecto es un conjunto de actividades relacionadas para lograr un fin específico, con un comienzo y fin claros, sujeto a tres "restricciones" principales: Tiempo, Presupuesto y Alcance.

3.- Dev Software – visión general

- Con independencia del área de aplicación, tamaño o complejidad del proyecto, cualquier sistema se encontrará al menos en una de las siguientes fases genéricas:
 - *Definición ~ Requisitos y análisis (del sistema, del sw.)*
 - *Desarrollo ~ Diseño, codificación, prueba*
 - *Mantenimiento.*

3.- Dev Software – fase de definición

- *¿Qué debe hacer el sistema?*
 - información que ha de manejar el sistema
 - necesidades de rendimiento
 - restricciones de diseño
 - interfaces del sistema con los usuarios y con otros sistemas
 - criterios de validación
- Se elaboran los documentos de requisitos del sistema (SyRS) y del software (SRS)

3.- Dev Software – fase de desarrollo

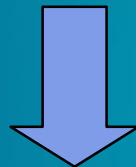
- *¿Cómo construir el sistema?*
- Se diseñan las estructuras de los datos y los programas
 - cómo se caracterizan las interfaces,
 - cómo realizar el paso del diseño al lenguaje de programación,
 - cómo ha de realizarse la prueba,
- Se escriben y documentan los programas,
- Se prueba el software construido.

3.- Dev Software – fase de mantenimiento

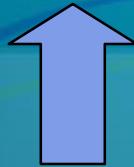
- Comienza una vez construido el sistema, cuando se empieza a utilizar.
- Se centra en el **cambio**.
- El software es sometido a reparaciones y **modificaciones** cada vez que se detecta un fallo o se necesita cubrir una nueva necesidad de los usuarios.
- En esta fase recae el mayor porcentaje del coste de un sistema.

3.- Dev Software – fase de mantenimiento

Un buen sistema no es sólo un conjunto de programas que funcionan.



Debe ser *fácil de mantener*



Documentación esencial
(CASE, *Computer Assisted
Software Engineering*)

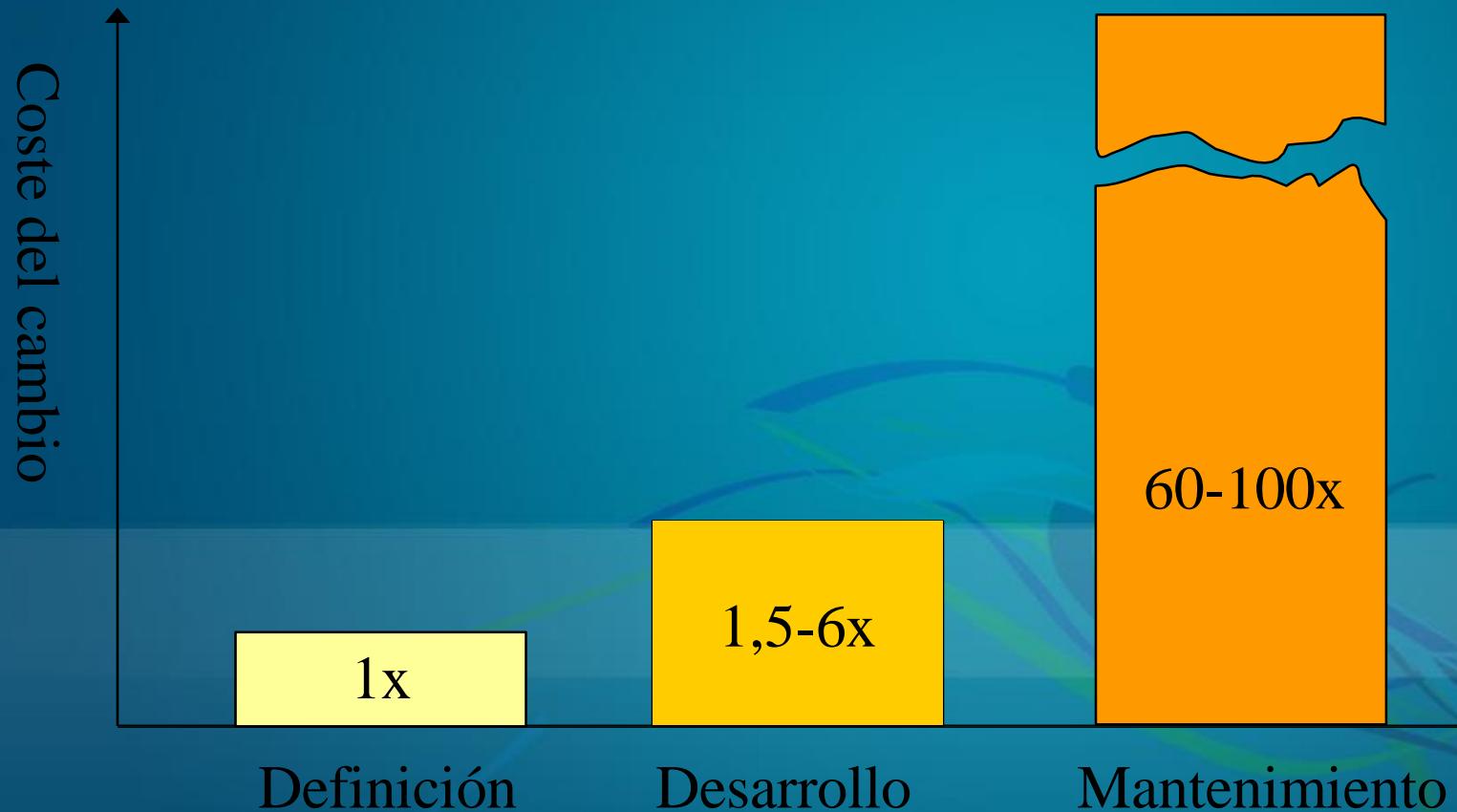
3.- Dev Software – tipos de mantenimiento

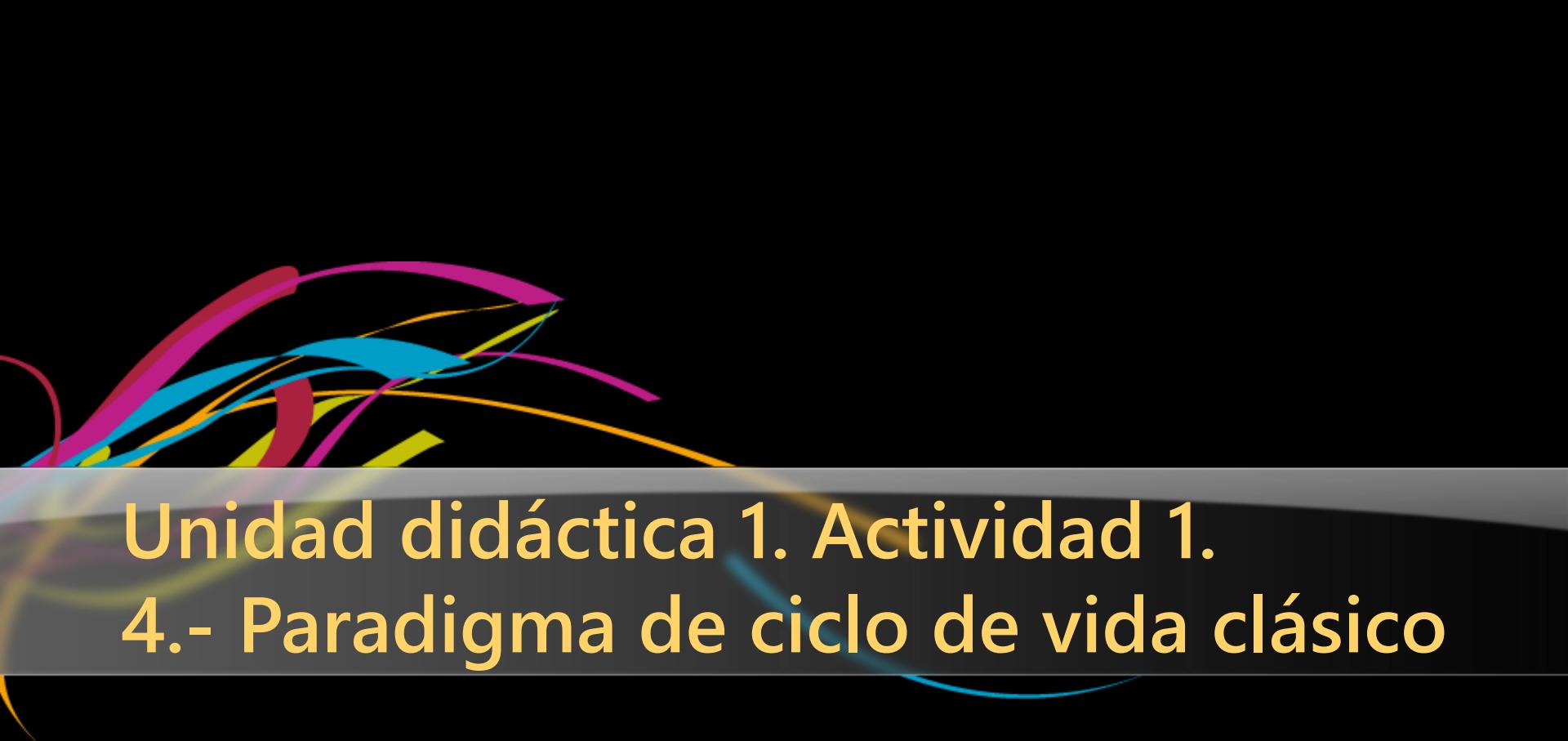
- **Correctivo:** un programa no realiza correctamente la aplicación para la que ha sido diseñado, y, por tanto, debe ser modificado.
- **Perfectivo:** modificaciones a los programas para conseguir mayor adecuación a los requisitos, mayor eficiencia, o simplemente recoger nuevas funcionalidades no expresadas en la fase de definición del sistema.

3.- Dev Software – tipos de mantenimiento

- **Adaptativo:** Adaptar los programas para acomodarlos a los cambios de su entorno externo (modificaciones en la legislación, CPU, SO, las reglas de negocio, etc.)
- **Preventivo:** El software se deteriora con los cambios, y este tipo de mantenimiento hace cambios en los programas para que se puedan corregir, adaptar y mejorar más fácilmente (**Reingeniería del software**).

3.- Dev Software – impacto del cambio





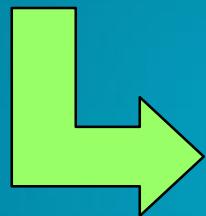
Unidad didáctica 1. Actividad 1.

4.- Paradigma de ciclo de vida clásico

Ciclo Superior Desarrollo de aplicaciones multiplataforma
IES Muralla Romana

4.- Ciclos de vida

Ciclo de vida ≠ Ciclo de desarrollo



Toda la vida del sistema:
desde la concepción hasta
el fin de uso

Desde el análisis
hasta la entrega
al usuario

4.- Ciclos de vida

Un ciclo de vida debe:

- Determinar el orden de las fases del Proceso Software
- Establecer los criterios de transición para pasar de una fase a la siguiente

4. Actualmente

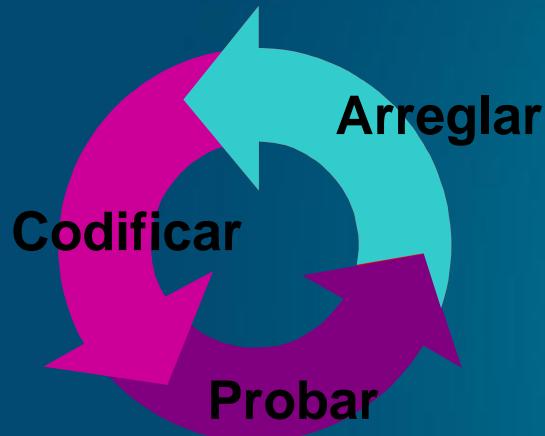
- En cualquier ciclo de vida actual se identifican tres macro-fases genéricas:
 - Definición (pensar): se centra en el **qué**.
 - Se intenta identificar **qué** información ha de ser procesada, **qué** función y rendimiento se desea, **qué** interfaces van a ser establecidas, **qué** restricciones de diseño existen y **qué** criterios de validación se necesitan para definir un sistema correcto.
 - Los métodos a aplicar variarán dependiendo del paradigma elegido, pero hay tres tareas principales: ingeniería de sistemas o de información, planificación del proyecto y análisis.
 - Desarrollo (hacer): se centra en el **cómo**.
 - Se intenta definir **cómo** han de diseñarse las estructuras de datos y la arquitectura del software, **cómo** han de implementarse los detalles procedimentales, **cómo** ha de traducirse el diseño a un lenguaje de programación y **cómo** ha de realizarse la prueba.
 - Los métodos aplicados variarán, pero hay tres tareas principales: diseño del software, generación del código y prueba del software.
 - Mantenimiento (mantener): se centra en el **cambio**.
 - Está asociada a la **corrección** de errores, a las **adaptaciones** requeridas por evolución del entorno del software y a los **cambios** debidos a las mejoras producidas por los requisitos del cliente.
 - Esta fase vuelve a aplicar los pasos de las anteriores, pero en el contexto de un software ya existente.

4.- Paradigma ciclo de vida

Existen muchos modelos a seguir para el desarrollo del software. El más clásico es el modelo en cascada o paradigma de ciclo de vida clásico.

- Destacan entre otros modelos:
 - **Modelo en espiral** basado en prototipos.
 - **Programación extrema** como representativo de los métodos de programación ágil.
 - **Métrica 3** como modelo a aplicar en grandes proyectos relacionados con las administraciones públicas.

Codificación Directa



- *También llamado CODE AND FIX*
- *Sinónimo de ARTE en el Desarrollo de Software*
- *Antónimo de INGENIERÍA en el Desarrollo de Software*
- ***En absoluto*** es recomendable para proyectos un poco más grande que “enanos”



4.- Modelos tradicionales



4.- Paradigma ciclo de vida

Modelo en cascada

El paradigma del ciclo de vida clásico del software o **Modelo en cascada** tiene las siguientes fases:

1. Requisitos
2. Análisis.
3. Diseño.
4. Codificación.
5. Pruebas.
6. Instalación.
7. Mantenimiento.
8. Documentación (implícito a todas las fases).

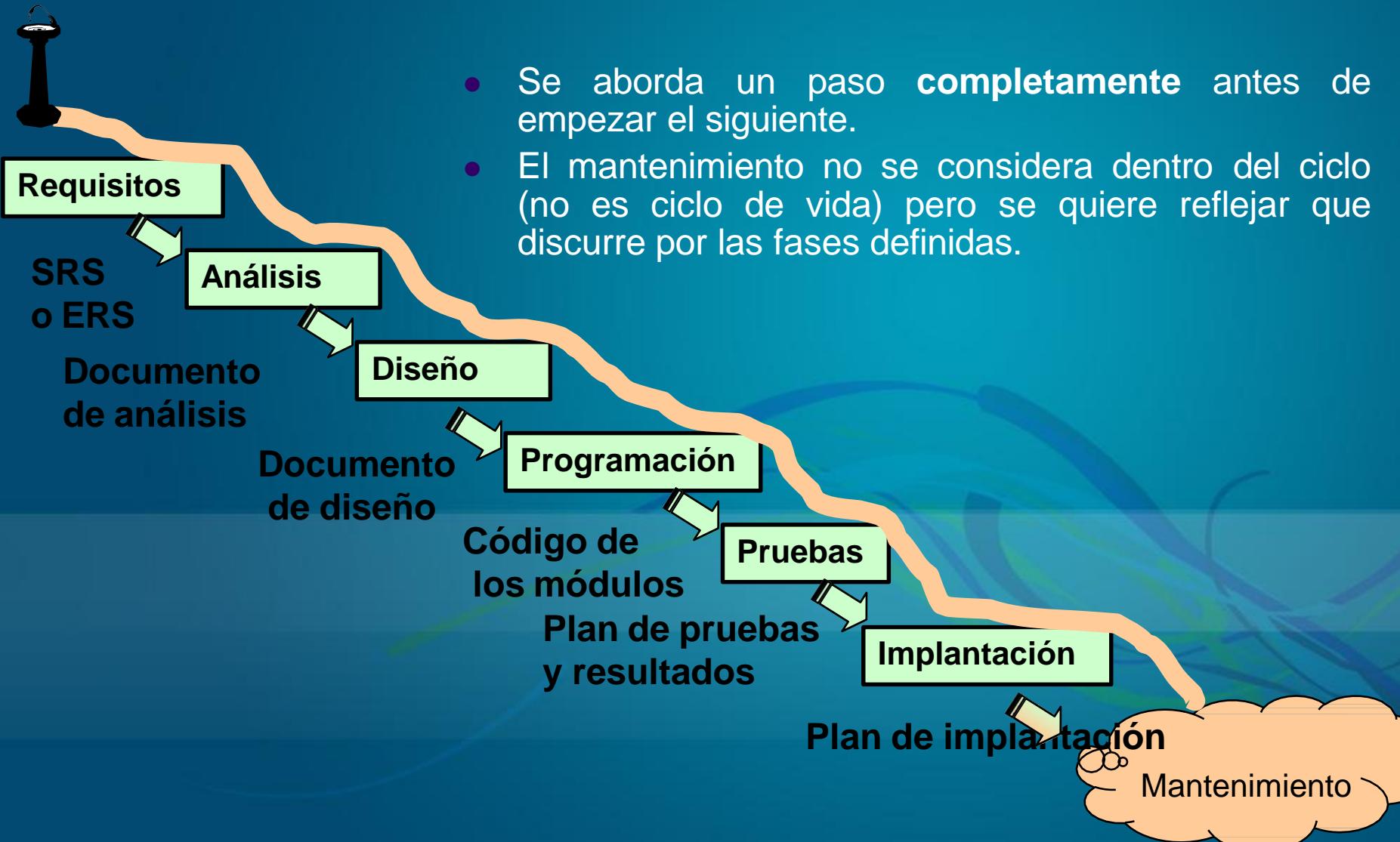
4.- Paradigma ciclo de vida

Modelo en cascada

- También conocido como Waterfall (cascada de agua) en inglés.
- Establecido durante los años 70 como alternativa al Code and Fix, que provocó la crisis del software.
- Es el que sigue un gran número de metodologías y es el más conocido, estudiado, difundido y empleado.
- Pone especial énfasis en la realización temprana de actividades de definición de requisitos y de documentación de análisis y diseño como paso previo a la codificación.
- Es una encadenación lineal y en secuencia de las siguientes actividades generales:
 - Analizar.
 - Diseñar.
 - Codificar.
 - Probar.
 - Implantar.
- Cada fase genera productos de salida que servirán a la siguiente fase para desarrollar la actividad de la misma como productos de entrada.

4.- Paradigma ciclo de vida

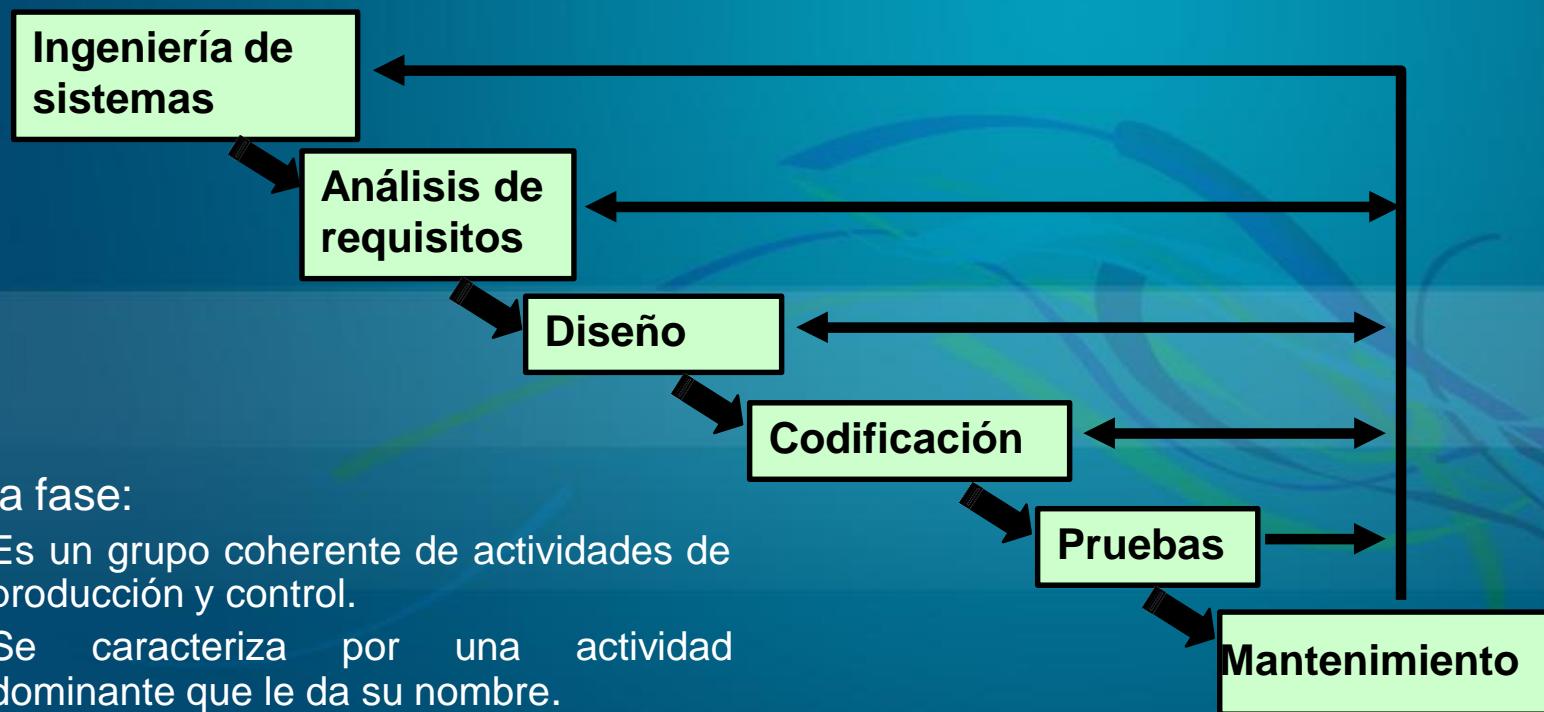
Modelo en cascada



4.- Paradigma ciclo de vida

Modelo en cascada

- El modelo original, propuesto por Winston Royce en 1970, preveía bucles de realimentación.
- La gran mayoría de las organizaciones que lo aplican, sin embargo, lo hacen como si fuese estrictamente lineal.



4.- Paradigma ciclo de vida

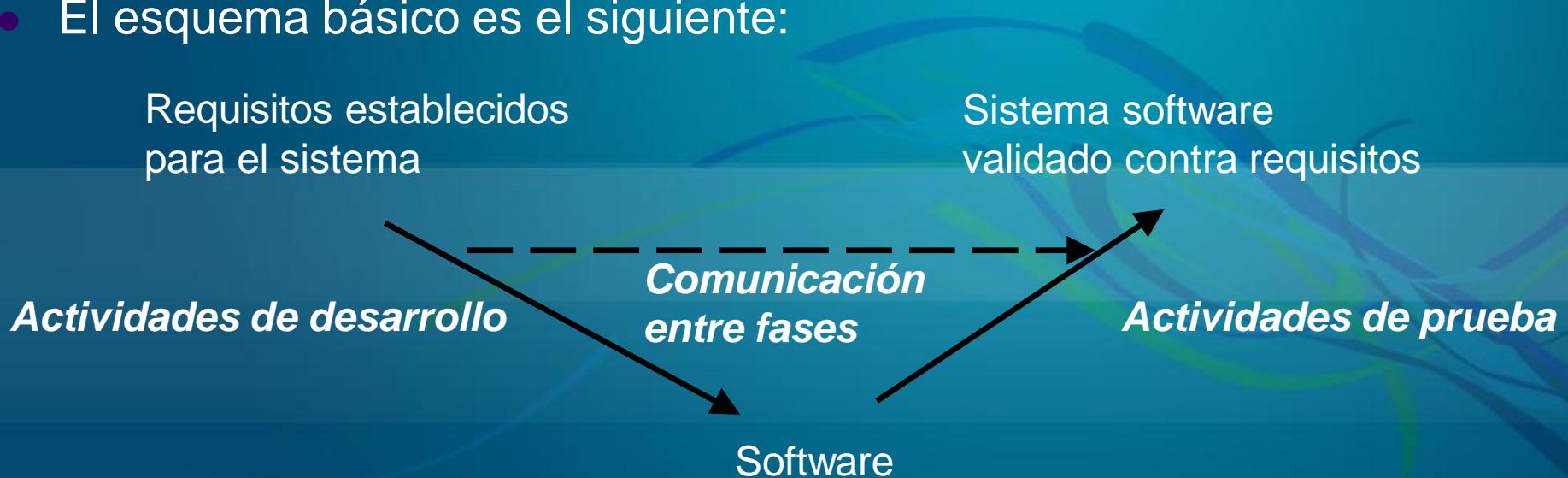
Modelo en cascada

- Ventajas:
 - Conforma un marco de referencia para asignar todas las actividades de desarrollo software.
 - Es un método muy estructurado que establece pautas de trabajo muy claras.
 - Facilita mucho la coordinación de los recursos implicados.
 - Facilita la disposición de hitos de seguimiento/control en el desarrollo de proyectos.
 - Facilita la estimación y el seguimiento/control del progreso de las actividades.
 - Facilita la detección de desviaciones y la realización de acciones correctivas.
 - Proporciona productos entregables intermedios que conforman el producto final.
- Inconvenientes:
 - Comienza estableciendo **todos** los requisitos del sistema a desarrollar:
 - Muchas veces esto no es viable:
 - Puede ser difícil para el propio usuario.
 - Hay cambios de parecer de los usuarios, habiendo finalizado ya la fase de requisitos.
 - Posee una gran rigidez: cada actividad es prerequisito de las que le siguen.
 - Su aplicación en el mundo real no contempla la “vuelta atrás”: es utópicamente lineal.
 - No soporta prácticas modernas de desarrollo (e.g., Prototipado).
 - Los posibles problemas se detectan tarde aunque se incluyan actividades de V&V.
 - Los únicos productos (parciales) aprovechables están en forma de documentos:
 - “Nada está hecho hasta que todo está hecho”.
 - Un cambio debido a un error puede suponer un gran coste.

4.- Paradigma ciclo de vida

Modelo en V

- Es un modelo similar al de Cascada, aunque “mejorado”.
- Tiene dos tiempos (tipos de actividades) claramente marcados:
 - Rama descendente: actividades de desarrollo.
 - Rama ascendente: actividades de prueba.
 - Ambas ramas se comunican en materia de pruebas:
 - Cada fase de la rama descendente tiene su homóloga en la rama ascendente para las actividades de prueba correspondientes.
- El esquema básico es el siguiente:



4.- Paradigma ciclo de vida

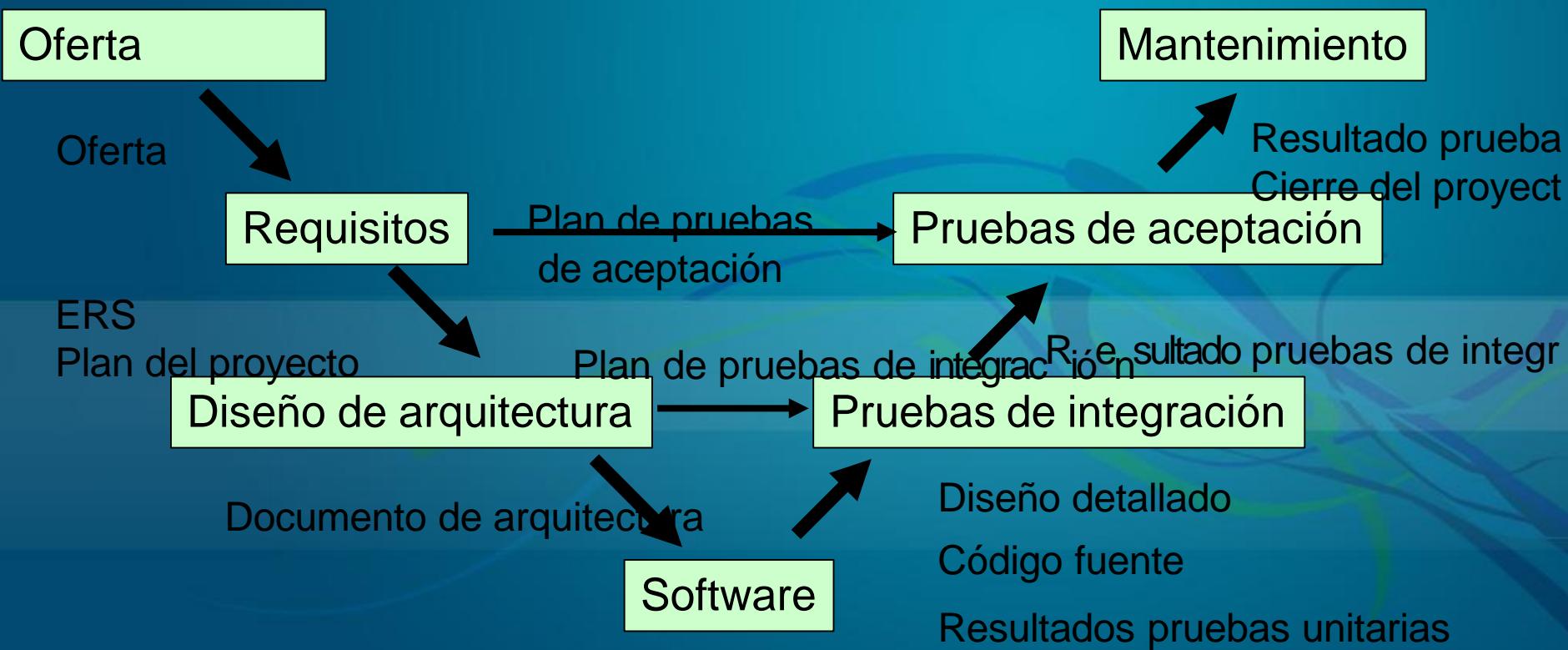
Modelo en V

- Cada fase de la rama descendente (rama de desarrollo):
 - Parte de las salidas de la fase previa y las verifica.
 - Aborda las actividades propias definidas para dicha fase.
 - Prepara las pruebas que van a permitir validar lo que se ha obtenido.
 - Estas pruebas se preparan aquí pero se ejecutarán en su fase homóloga situada a su mismo nivel de la rama ascendente.
 - Sirve de punto de partida a las actividades de la fase siguiente a través de los productos obtenidos.
- Cada fase de la rama ascendente (rama de pruebas):
 - Ejecuta las pruebas (ya preparadas) para validar lo definido durante la fase homóloga de su mismo nivel de la rama descendente.
 - Verifica las actividades llevadas a cabo durante la fase.
 - Sirve de punto de partida a las actividades de la fase siguiente.

4.- Paradigma ciclo de vida

Modelo en V

- Un ejemplo muy simplificado podría ser el siguiente:



4.- Paradigma ciclo de vida

Modelo en V

- Ventajas:
 - Al ser una variante del de Cascada, hereda todas sus ventajas.
 - Conforma un marco de referencia para asignar todas las actividades de desarrollo software, incluyendo las actividades V&V de lo que se hace en las sucesivas etapas.
 - Favorece la consideración de las pruebas lo antes posible (comunicación horizontal entre las dos ramas de la V).
- Inconvenientes:
 - Al igual que en Cascada, se comienza estableciendo **todos** los requisitos del sistema a desarrollar:
 - Muchas veces esto no es viable:
 - Puede ser difícil para el propio usuario.
 - Hay cambios de parecer de los usuarios, habiendo finalizado ya la fase de requisitos.
 - No soporta prácticas modernas de desarrollo (e.g., Prototipado).
 - Posee una gran rigidez, aunque menos que en el de Cascada por haber comunicación horizontal.
 - Su aplicación en el mundo real no contempla la “vuelta atrás”: es utópicamente lineal.
 - Los únicos productos (parciales) aprovechables están en forma de documentos:
 - “Nada está hecho hasta que todo está hecho”.
 - Un cambio debido a un error puede suponer un gran coste.

4.- Paradigma ciclo de vida

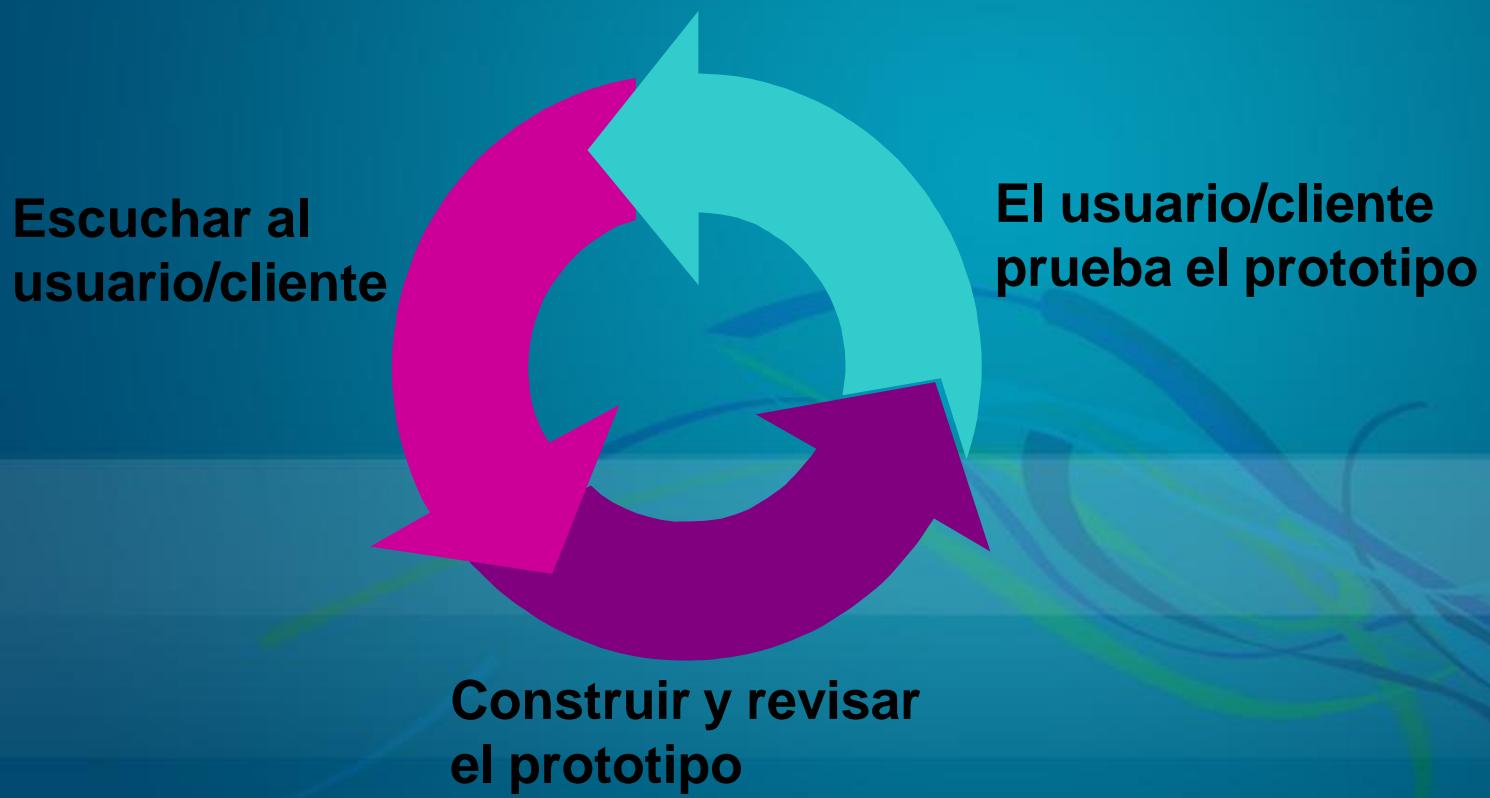
Prototipado

- Su sentido es análogo al que tiene en otras ingenierías:
 - El prototipado o construcción de un prototipo es un proceso que facilita al desarrollador la creación de un modelo del software a desarrollar.
- Un prototipo puede ser de muchos tipos:
 - En papel, describiendo por ejemplo la interacción hombre-máquina.
 - Funcional, implementando un subconjunto de las funciones requeridas.
 - Un programa ya existente, que ejecuta toda o parte de la funcionalidad deseada pero que tiene características que deben ser mejoradas o abordadas en el nuevo desarrollo.
- Es muy adecuado cuando no se sabe exactamente lo que se quiere construir. Algunas razones para no saberlo son:
 - El propio cliente/usuario no sabe exactamente lo que quiere.
 - El desarrollador no está seguro de la eficiencia de una aproximación.
 - Existen dudas en la interfaz hombre-máquina.
 - ...

4.- Paradigma ciclo de vida

Prototipado

- El proceso básico del prototipado involucra las siguientes tres fases que se detallarán a continuación:



4.- Paradigma ciclo de vida

Prototipado

- Escuchar al usuario/cliente:
 - Se corresponde con la recolección de requisitos.
 - Se definen los objetivos globales del sistema.
 - Se identifican los requisitos conocidos y las áreas donde se precisa una mayor definición.
- Construir y revisar el prototipo:
 - Se realiza un desarrollo rápido centrado exclusivamente en los aspectos del software visibles para el usuario/cliente:
 - Formatos de entrada.
 - Formatos de salida.
 - Pantallas.
 - Etc.
 - Este desarrollo rápido conforma el prototipo elaborado en este ciclo.
 - En esta fase resulta muy beneficioso emplear herramientas de 4^a generación (4GL). Por ejemplo:
 - Delphi.
 - Cristal Report para los informes.
- El usuario/cliente prueba el prototipo:
 - El prototipo es evaluado por el usuario/cliente.
 - Esta evaluación permite refinar los requisitos del software.
 - Este refinamiento es la entrada al siguiente ciclo del Prototipado.

4.- Paradigma ciclo de vida

Prototipado

- La pregunta surge cuando se ha finalizado el prototipo:
 - ¿Qué se hace con él?
- En la mayoría de los casos, el prototipo será:
 - Demasiado lento.
 - Demasiado grande.
 - Demasiado torpe en su operativa.
 - En definitiva y según Pressman, está hecho con chicles y alambres.
- Por lo tanto, idealmente, se debe de desechar el prototipo y empezar de nuevo.
 - El prototipo sólo debe servir para identificar los requisitos del software cuando no estén claros.
- Con los 4GL o herramientas similares, las pantallas y la dinámica de pantallas se podría aprovechar; pero siempre adecuándolos a los estándares y normas seguidos en un producto no prototipo.
- Después de un Prototipado, típicamente se suele continuar con un ciclo de vida clásico: combinación de ciclos de vida.

4.- Paradigma ciclo de vida

Prototipado

- Ventajas:
 - El prototipo es un mecanismo ideal para extraer requisitos cuando no están claros, incluso por parte del usuario.
 - “No sé lo que quiero hasta que veo lo que no quiero”.
- Inconvenientes:
 - Existe una clara tendencia del usuario/cliente a creer que el trabajo ya está hecho y que en breve dispondrá de un sistema funcional.
 - En realidad, se está simplemente en la primera fase.
 - El desarrollador toma necesariamente decisiones de implementación simplemente porque le son conocidas y le permiten desarrollar rápidamente el prototipo.
 - No son decisiones adecuadas para el sistema real.
 - El problema radica en que estas decisiones a menudo se mantienen en el sistema real.

4.- Modelos evolutivos



4.- Paradigma ciclo de vida

Modelos Evolutivos

- Se adaptan más fácilmente a los cambios introducidos a lo largo del desarrollo.
- Son iterativos.
- En cada iteración se obtienen versiones más completas del software.
- Distinguimos:
 - **Modelo en espiral.**
 - **Modelo incremental.**

4.- Paradigma ciclo de vida

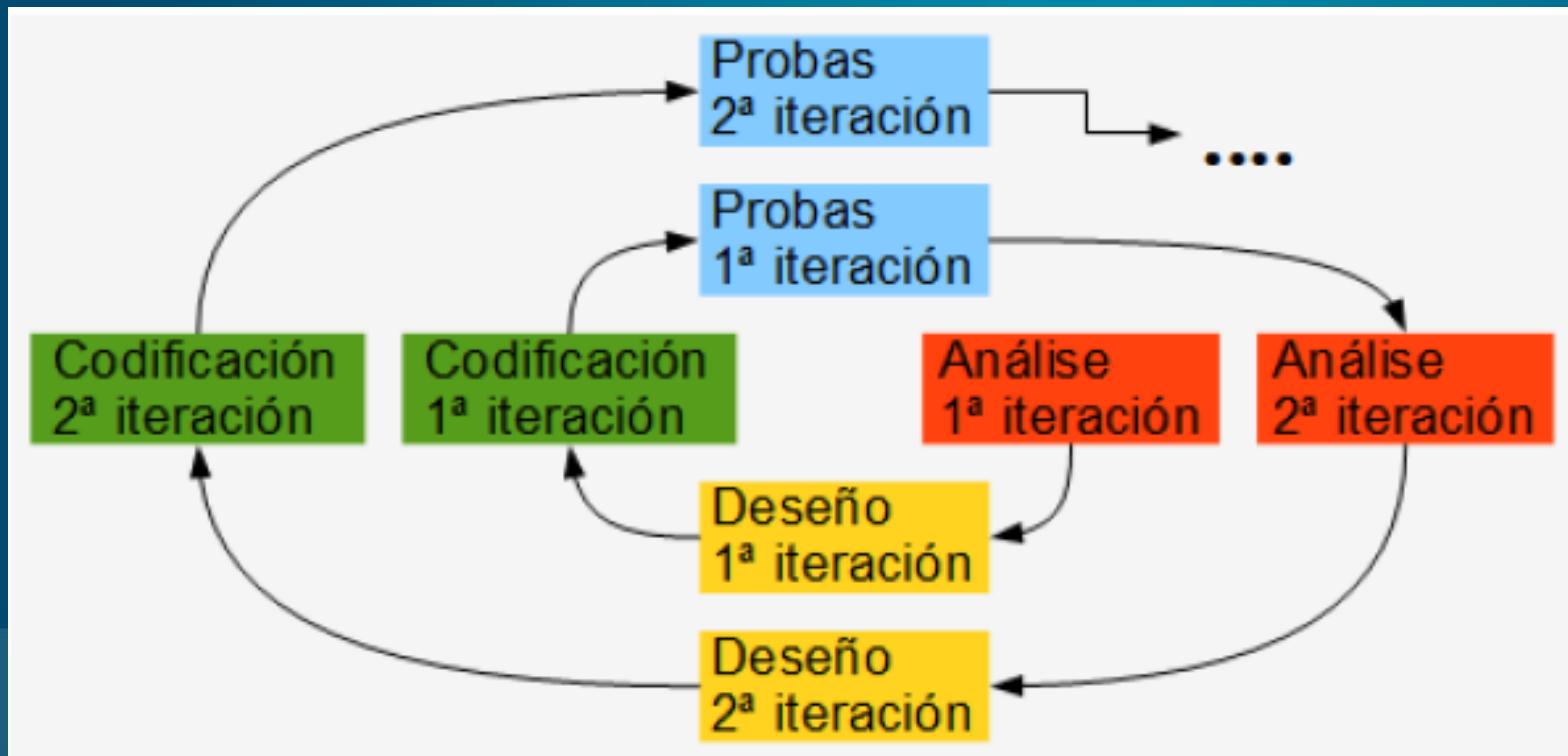
Modelos Evolutivos. Modelo en espiral

- Se basa en la creación de un prototipo del proyecto que se va a ir **perfeccionando** en **sucesivas iteraciones**, a medida que se añaden nuevos requisitos, pasando en cada iteración por el proceso de análisis, diseño, codificación y pruebas (modelo cascada).
- En cada vuelta a la espiral se suelen seguir los siguientes pasos:
 1. Fijar objetivos y determinar alternativas.
 2. Evaluar las alternativas y elegir la mejor.
 3. Desarrollo de la alternativa elegida, evaluación y validación del resultado.
 4. Planificación de la próxima iteración.

El modelo espiral es compatible con el modelo en cascada.

4.- Paradigma ciclo de vida

Modelos Evolutivos. Modelo en espiral



4.- Paradigma ciclo de vida

Modelos Evolutivos. Modelo en espiral

- Al final de cada iteración el equipo que desarrolla el software y el cliente, analizan el prototipo conseguido y acordarán si se inicia una nueva iteración.
- Siempre se trabaja sobre un prototipo por lo que en el momento que se decida no realizar nuevas iteraciones y acabar el producto, se refinará el prototipo para conseguir la versión final acabada, estable y robusta.

4.- Paradigma ciclo de vida

Modelos Evolutivos. Modelo en espiral

- Se va refinando el producto para acercarlo a lo que el cliente quiere.
- Este modelo gira en torno a los riesgos. Para cada iteración hay que evaluar los riesgos de afrontar una mejora o modificación y en caso de considerar que sería positivo afrontarla, tomar la opción que suponga un riesgo menor.
- Con cada iteración alrededor de la espiral (comenzando en el centro y siguiendo hacia el exterior), se construyen sucesivas versiones del software, cada vez más completas y, al final, el propio sistema software totalmente funcional.

4.- Paradigma ciclo de vida

Modelos Evolutivos. Modelo en espiral

- Puede comenzarse el proyecto con un alto grado de incertidumbre.
- Presenta bajo riesgo de retraso en caso de detección de errores, ya que se puede solucionar en la próxima rama del espiral.
- Utilizado en proyectos largos como puede ser la creación de un SO, que necesitan constantes cambios.

4.- Paradigma ciclo de vida

- Modelo en espiral

Ventajas

- Puede comenzarse el proyecto con un alto grado de incertidumbre.
- Permite iteraciones, vuelta atrás y finalizaciones rápidas.
- Eliminar errores y alternativas no atractivas al comienzo -> solución en la próxima espiral.
- Incorpora objetivos de calidad y gestión de riesgos.
- Permite acomodar otros modelos.

4.- Paradigma ciclo de vida

- Modelo en espiral

Desventajas

- Coste temporal que suma cada vuelta del espiral.
- Dificultad de evaluar los riesgos y necesidad de la presencia o la comunicación continua con el cliente o usuario.
- Difícil predecir el coste y duración del proyecto.

4.- Paradigma ciclo de vida

Modelos Evolutivos. Modelo incremental

- Es el proceso de construir una implementación parcial del sistema global y posteriormente ir aumentando la funcionalidad del sistema.
- Es la aplicación reiterada de varias secuencias basadas en el modelo en Cascada.
 - Cada aplicación del ciclo constituye un incremento del software.
 - Cada incremento resulta en un producto operativo.
 - Cada incremento puede ser:
 - El refinamiento de un incremento anterior (siguiendo la filosofía del Prototipado).
 - El desarrollo de una nueva funcionalidad no incorporada en incrementos anteriores.
 - En el primer incremento de un sistema se debe abordar el núcleo esencial del sistema (requisitos fundamentales).
 - En incrementos posteriores se abordarán funciones suplementarias (algunas ya conocidas y otras no).
- El usuario/cliente evalúa el resultado de un incremento y se elabora un plan para el incremento siguiente.
- El proceso se repite hasta que se elabore el producto completo.
- Al seguir un desarrollo incremental, el software debe construirse de tal forma que facilite la incorporación de nuevos requisitos.

4.- Paradigma ciclo de vida

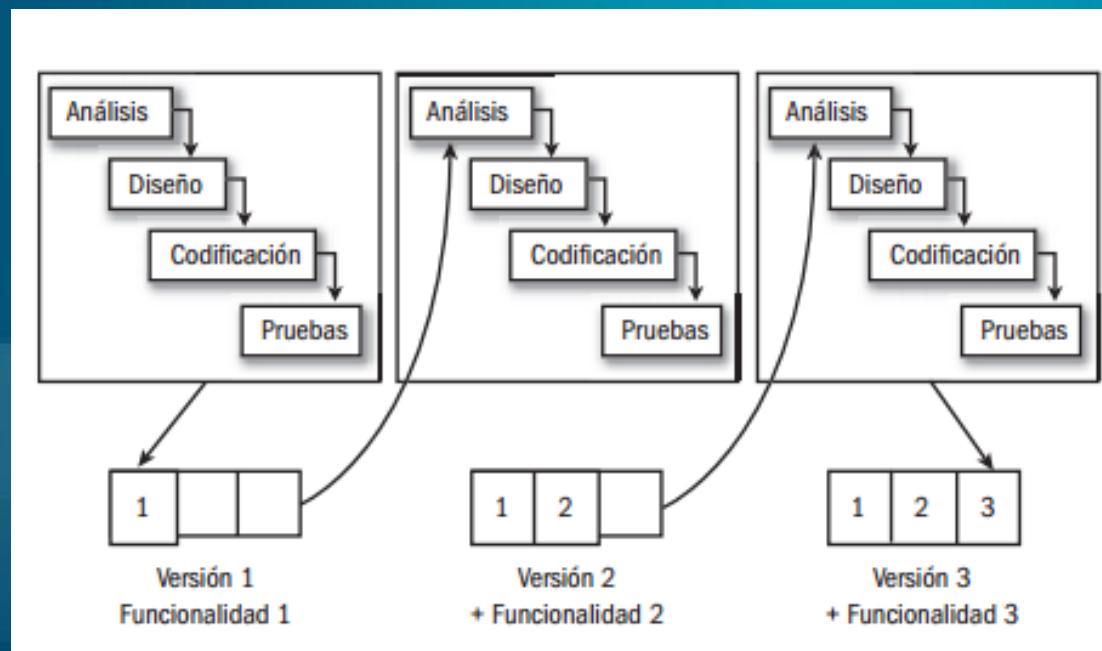
Modelos Evolutivos. Modelo incremental

- Se basa en construir incrementando las funcionalidades del programa.
- Se realiza **construyendo por módulos** que cumplen diferentes funciones del sistema, «divide y vencerás».
- Permite ir **aumentando gradualmente las capacidades del software**. El sistema se crea añadiendo componentes funcionales al sistema -> incrementos.
- Permite la tarea del desarrollo permitiendo a cada miembro del equipo desarrollar un módulo particular en el caso de que el proyecto se realice por un equipo-> Desarrollo en serie.
- **Repetición del ciclo de vida en cascada**, aplicándose este ciclo en cada funcionalidad a construir.

4.- Paradigma ciclo de vida

Modelos Evolutivos. Modelo incremental

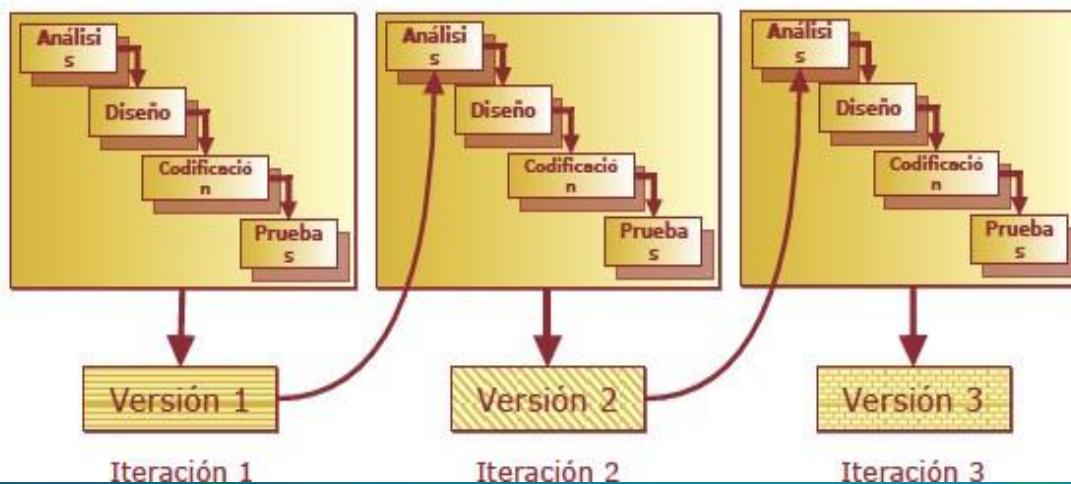
- Al final de cada ciclo se entrega una versión al cliente que contiene una nueva funcionalidad.
- Permite realizar entregas al cliente antes de terminar el proyecto.



4.- Paradigma ciclo de vida

Modelos Evolutivos. Modelo incremental

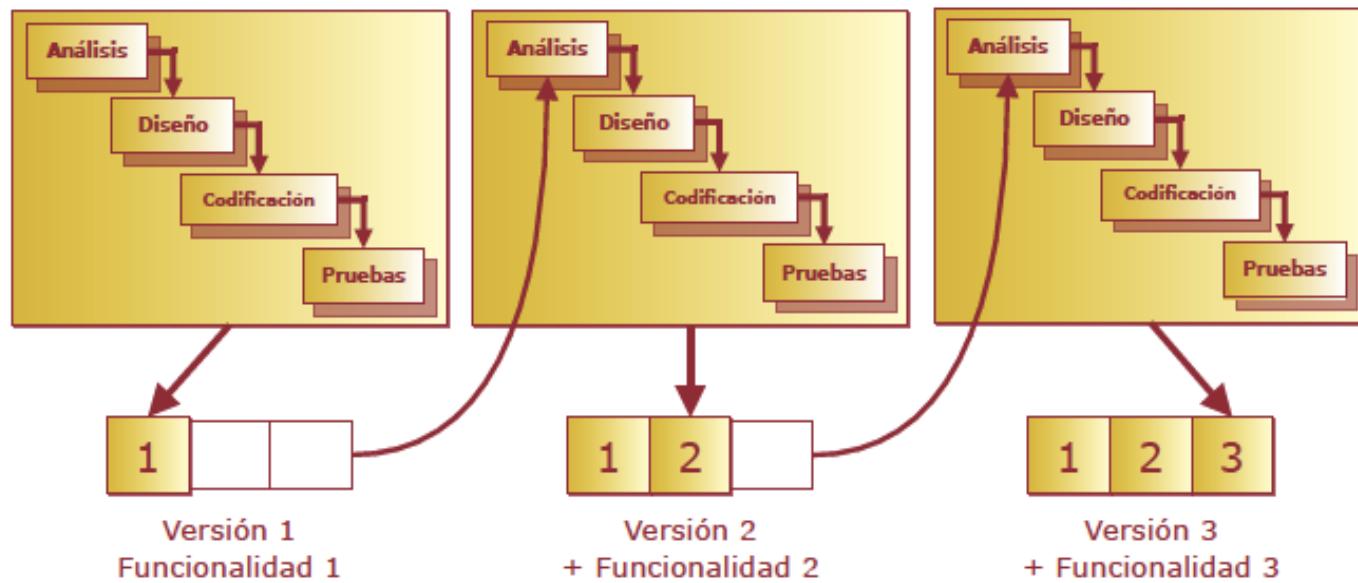
Ciclo de vida iterativo



4.- Paradigma ciclo de vida

Modelos Evolutivos. Modelo incremental

Ciclo de vida incremental



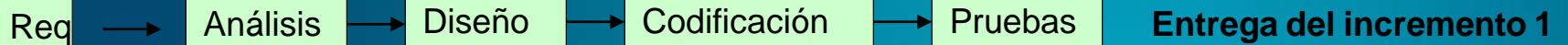
4.- Paradigma ciclo de vida

Modelos Evolutivos. Modelo incremental

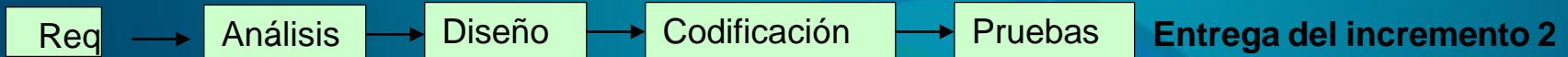
- Esquema básico:

Tiempo

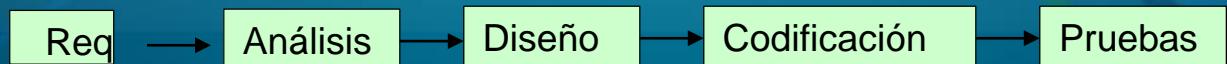
Incremento 1



Incremento 2



Incremento n



4.- Paradigma ciclo de vida

- Modelo incremental

Ventajas

- Construir un sistema pequeño siempre tiene menos riesgo que construir uno grande.
- Como se desarrollan funcionalidades independientes es más fácil revelar los requerimientos del usuario.
- Si se detecta un error grave, sólo se desecha la última iteración.
- No es necesario disponer de todos los requerimientos de todas las funcionalidades al comienzo del proyecto.
- Facilita el desarrollo de **divide y vencerás**.

4.- Paradigma ciclo de vida

- Modelo incremental

Ventajas

- Corrige la necesidad de una secuencia no lineal de pasos de desarrollo.
- Se evitan proyectos largos y se entrega "algo de valor" a los usuarios con cierta frecuencia.
- El usuario se involucra más.
- Mayor retorno de inversión.

4.- Paradigma ciclo de vida

- Modelo incremental

Desventajas

- Si no se seleccionan bien los requisitos a implementar en cada fase, podemos llegar a una fase incremental en la que haya que rehacer partes importantes del software y/o del diseño.

4.- Paradigma ciclo de vida

- Ejercicio

Considere el desarrollo de un sistema cuyo dominio de aplicación es conocido, sus objetivos y requerimientos funcionales son estables y simples de comprender desde un principio, la tecnología a utilizar ya está predeterminada y es bien conocida por el equipo de desarrollo.

¿Qué tipo de modelo de ciclo de vida elegiría para el desarrollo de dicho sistema?.

4.- Paradigma ciclo de vida

- Ejercicio

Considerar ahora el desarrollo de un sistema cuyo dominio de aplicación no es muy conocido por el equipo de desarrollo. En este caso, el cliente tampoco tiene muy claro qué es lo que quiere, de manera que los objetivos y requerimientos funcionales del sistema son inestables y difíciles de comprender. Además, el equipo de desarrollo va a utilizar una tecnología que le resulta completamente nueva.

Discutir qué modelo de ciclo de vida es más apropiado y qué etapas se deberían utilizar para desarrollar este sistema.



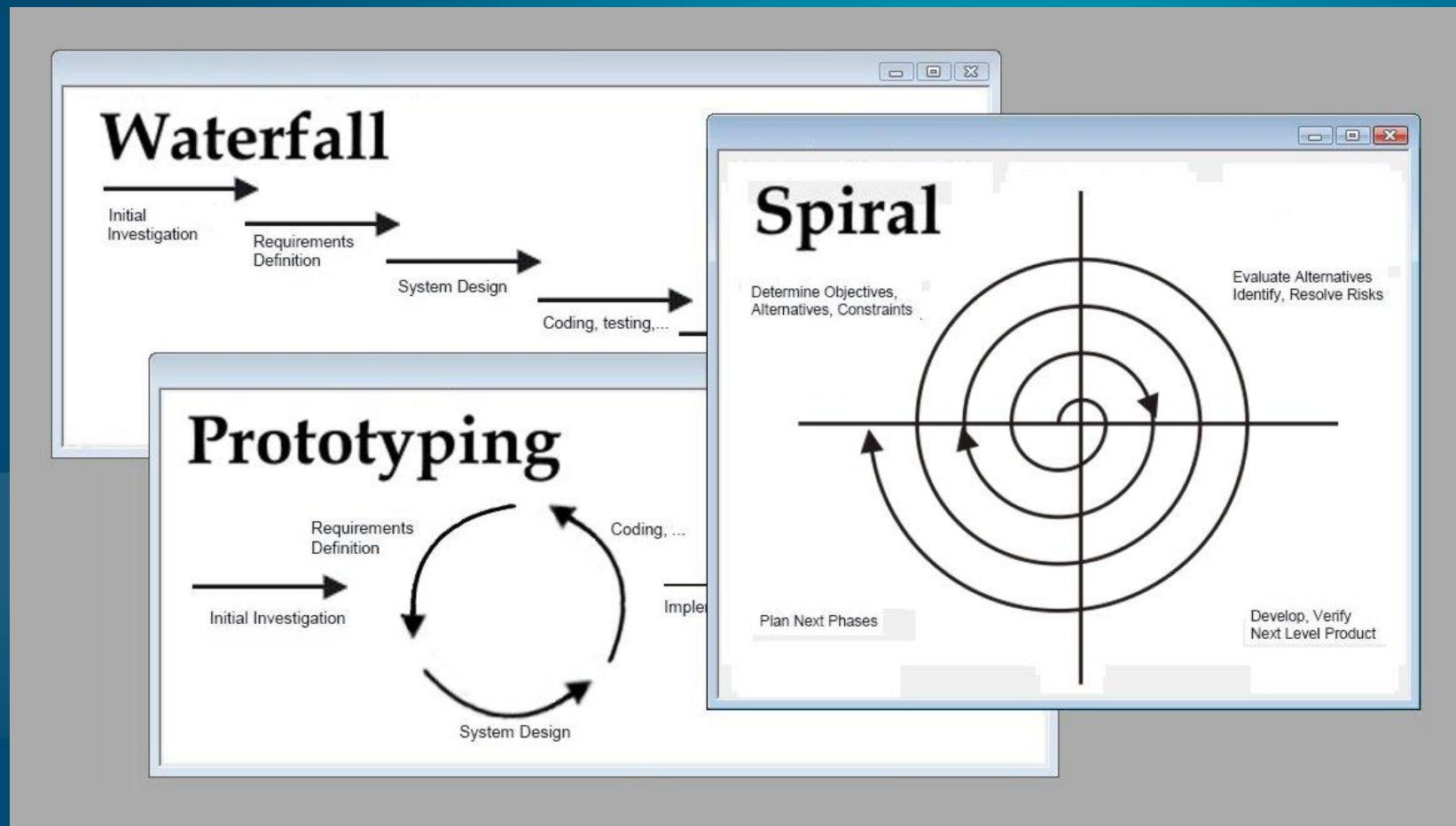
Unidad didáctica 1. Actividad 1.

5.- Metodologías

Ciclo Superior Desarrollo de aplicaciones multiplataforma
IES Muralla Romana

5.- Metodologías

- Una metodología de desarrollo del software es un marco de trabajo usado para estructurar, planificar y controlar el proceso de desarrollo de una aplicación.



5.- Metodologías

- Para el cumplimiento del ciclo de vida del software han sido estudiadas e implementadas algunas metodologías que garantizan que por medio del seguimiento de unos pasos específicos, pueda obtenerse con éxito el resultado esperado.
- Pueden clasificarse en **tradicionales** y **ágiles**.
- En las tradicionales no se pasa de una etapa del ciclo de vida antes de haber terminado por completo la anterior.
- En las ágiles se necesita un desarrollo rápido y eficaz en el que las etapas de requerimientos e implementación pueden hacerse simultáneamente.

5.- Metodologías.

Metodologías ágiles

- ¿Por qué aparecen?
- Los negocios operan en un entorno global que cambia rápidamente.
- Tienen que responder a nuevas oportunidades y mercados, condiciones económicas cambiantes y la aparición de productos y servicios competidores.
- Los procesos de desarrollo del software basados en una completa especificación de los requerimientos, diseño, construcción y pruebas del sistema no se ajustan al desarrollo rápido de aplicaciones.

5.- Metodologías.

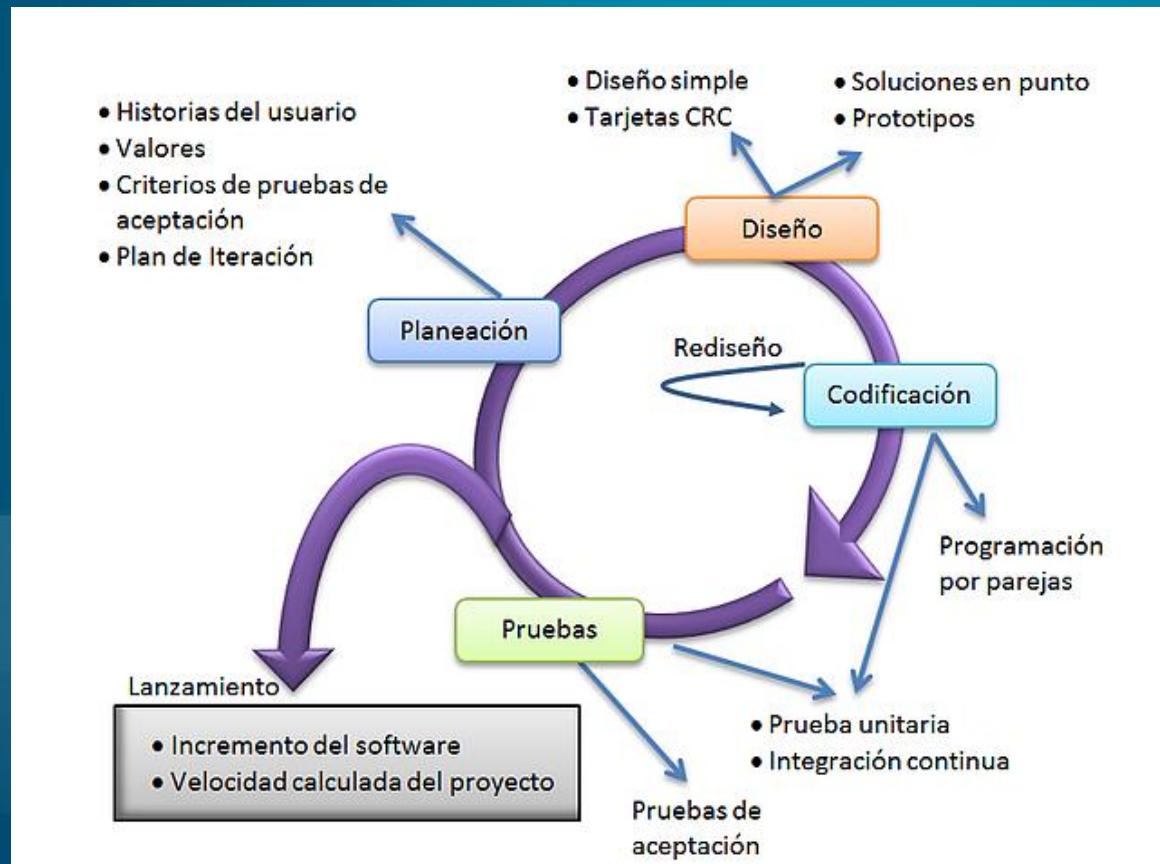
Metodologías ágiles

- ¿Por qué aparecen?
- Son aquellas que permiten adaptar la forma de trabajo a las condiciones del proyecto, consiguiendo flexibilidad e inmediatez en la respuesta para amoldar el proyecto y su desarrollo a las circunstancias específicas del entorno.
- Principios
- Valorar a los individuos y su interacción (desarrolladores y clientes).
- Desarrollar software que funciona, por encima de documentación exhaustiva.
- Colaboración con el cliente.
- Respuesta ante el cambio antes que el seguimiento de un plan.

5.- Metodologías.

Metodologías ágiles. Programación extrema

- Se establecen cuatro actividades estructurales:
- Proceso XP



5.- Metodologías.

Metodologías ágiles. Programación extrema

Planificación

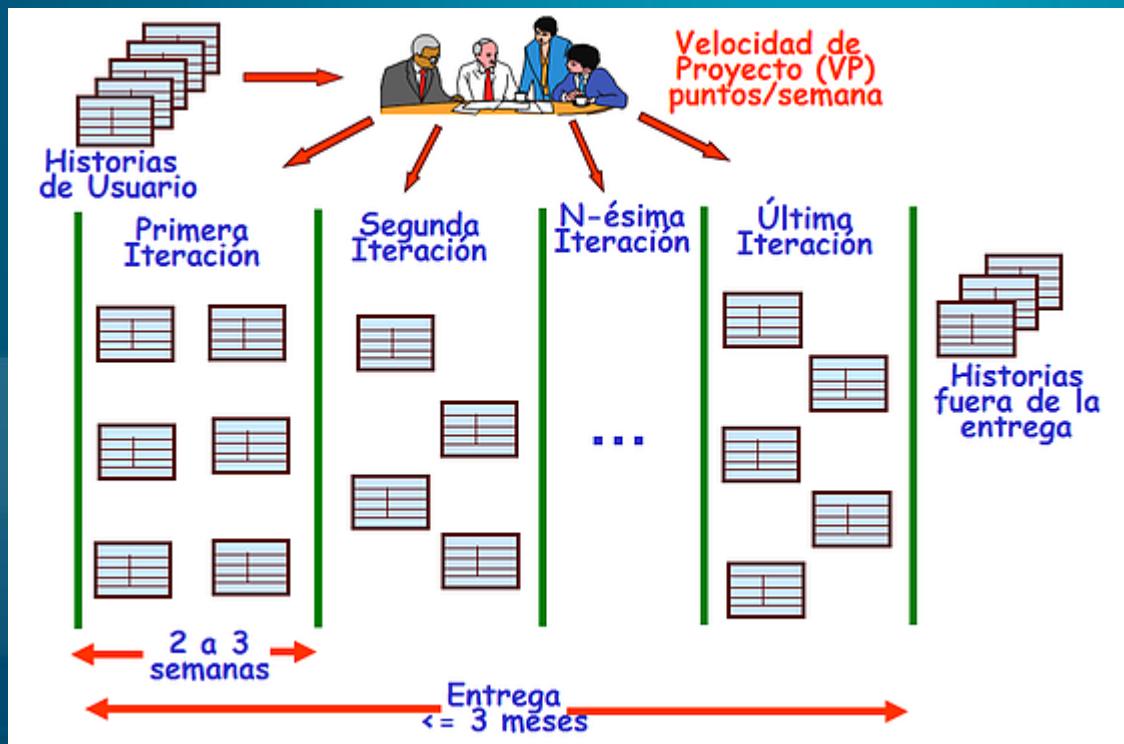
- Se realiza por etapas, es decir, que es iterativa.
- No puede haber una planificación sin que antes haya habido comunicación con el cliente. En esta reunión con el cliente, él establece sus requerimientos lo que hace que el equipo de software entienda cuáles son las características y funcionalidades que se necesitan.
- Cada reunión de planificación es coordinada por el gestor del proyecto.
- Cuando el desarrollador entiende lo que escucha del cliente, este puede ser capaz de hacer los casos de uso, en los cuales el cliente puede asignar prioridades entre todas las funciones del software

5.- Metodologías.

Metodologías ágiles. Programación extrema

Planificación

- Los desarrolladores convertirán cada prestación en tareas que duren como máximo 3 días de manera que la prestación completa no requiera más de 2 semanas.



5.- Metodologías.

Metodologías ágiles. Programación extrema

Planificación

- Entre todos se decide el número de prestaciones que formarán parte de cada iteración. Se denomina *velocidad del proyecto*.
- Al final de cada iteración se hará una reunión de planificación para que el cliente valore el resultado, si no se acepta, habrá que añadir las prestaciones no aceptadas a la siguiente iteración y el cliente deberá de reorganizar las prestaciones que faltan para que se respete la velocidad del proyecto.

5.- Metodologías.

Metodologías ágiles. Programación extrema

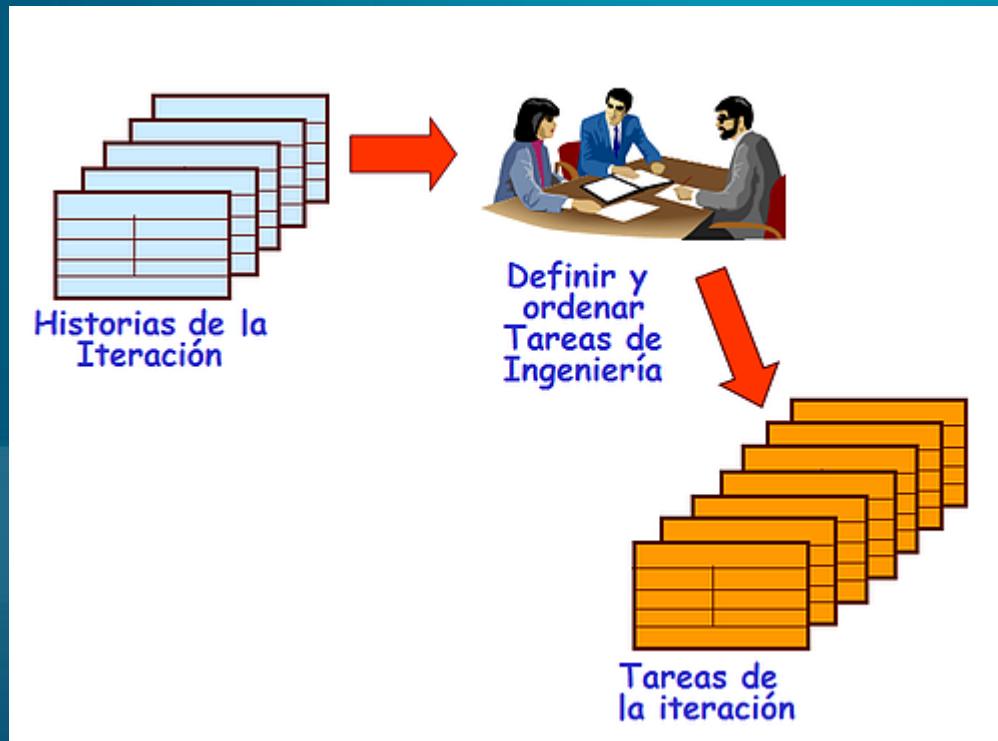
Diseño

- Es la guía para la implementación del sistema, por lo tanto debe ser claro, y para poder ser claro necesita de simplicidad, ya que no sólo será entendido por el programador sino que también en muchas ocasiones, por el usuario.
- En el diseño se pueden también asignar las responsabilidades y módulos de cada persona en el equipo.

5.- Metodologías.

Metodologías ágiles. Programación extrema

Diseño



5.- Metodologías.

Metodologías ágiles. Programación extrema

Diseño

- Se utilizará una tarjeta manual tipo CRC (*class, responsibilities, collaboration*) por cada objeto del sistema, en la que aparece el nombre de la clase, de la superclase, de las subclases, responsabilidades de la clase y los objetos que colaboran.

Historia de Usuario	
Número:	Usuario:
Nombre Historia:	
Prioridad en Negocio:	Riesgo en Desarrollo:
Puntos Estimados:	Iteración Asignada:
Programador Responsable:	
Descripción:	
Observaciones:	

5.- Metodologías.

Metodologías ágiles. Programación extrema

Diseño

- Las tarjetas se van colocando encima de una superficie formando una estructura que refleje las dependencias entre ellas.
- Se van completando y recolocando de forma manual a medida que avanza el proyecto.
- Los desarrolladores se reunirán periódicamente y tendrán una visión del conjunto y del detalle mediante estas tarjetas.

5.- Metodologías.

Metodologías ágiles. Programación extrema

Codificación y pruebas

- Los desarrolladores acuerdan unos estándares de codificación (nombres de variables, sangrías, alineamientos, etc.), que deben cumplirse.
- Se aconseja crear **tests unitarios** antes que el propio código ya que entonces se tiene una idea más clara de lo que se debe codificar.
- Permiten establecer los requerimientos primordiales.

5.- Metodologías.

Metodologías ágiles. Programación extrema

Codificación y pruebas

- Uno de los mejores mecanismos para hacer que la codificación funcione de manera correcta es la unión de **dos personas** del equipo, es decir, la **programación en parejas**, cada una de estas personas, con características distintas y especializadas en distintas áreas puede encargarse de tareas distintas dentro de un mismo código.
- El ritmo de trabajo es el mismo que por separado pero el resultado final es de mayor calidad.

5.- Metodologías.

Metodologías ágiles. Programación extrema

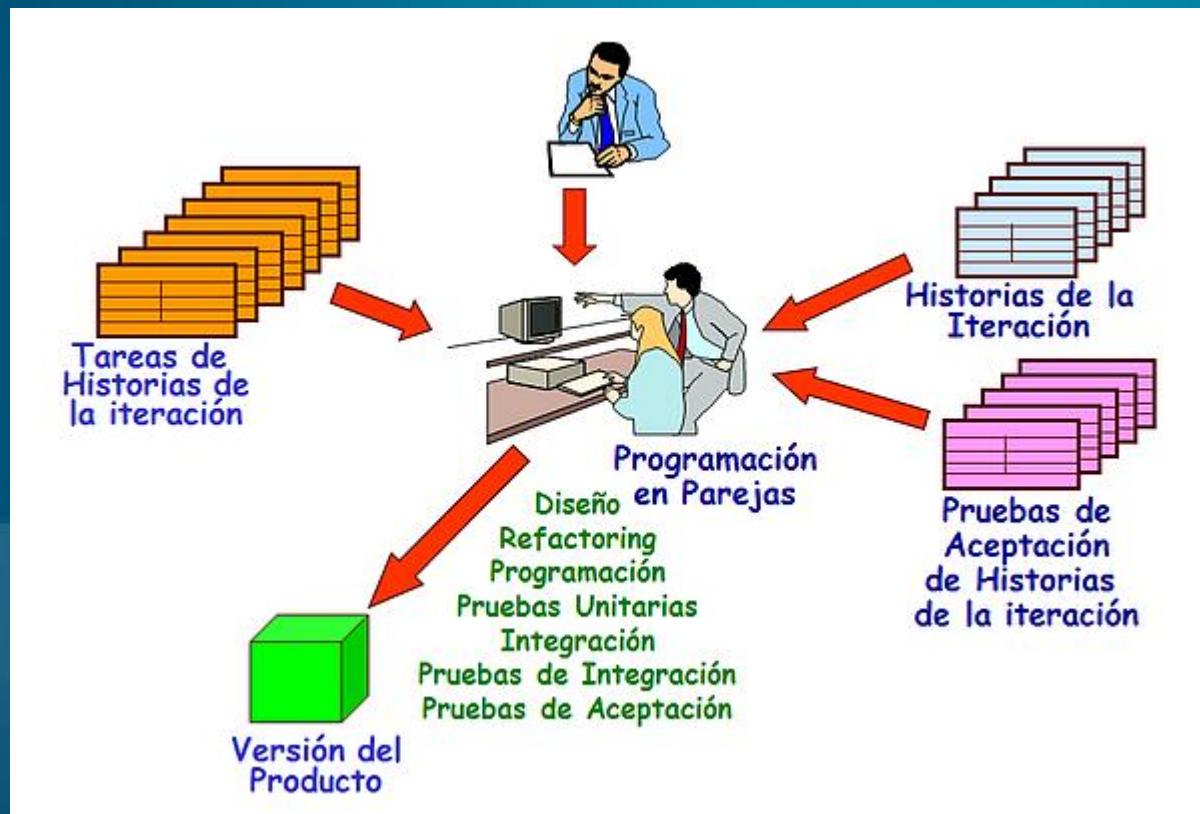
Codificación y pruebas

- Mientras uno está codificando, el otro puede pensar en cómo ese método afecta al resto de objetos y las dudas y propuestas que surjan reducen el número de errores y problemas en la integración posterior.

5.- Metodologías.

Metodologías ágiles. Programación extrema

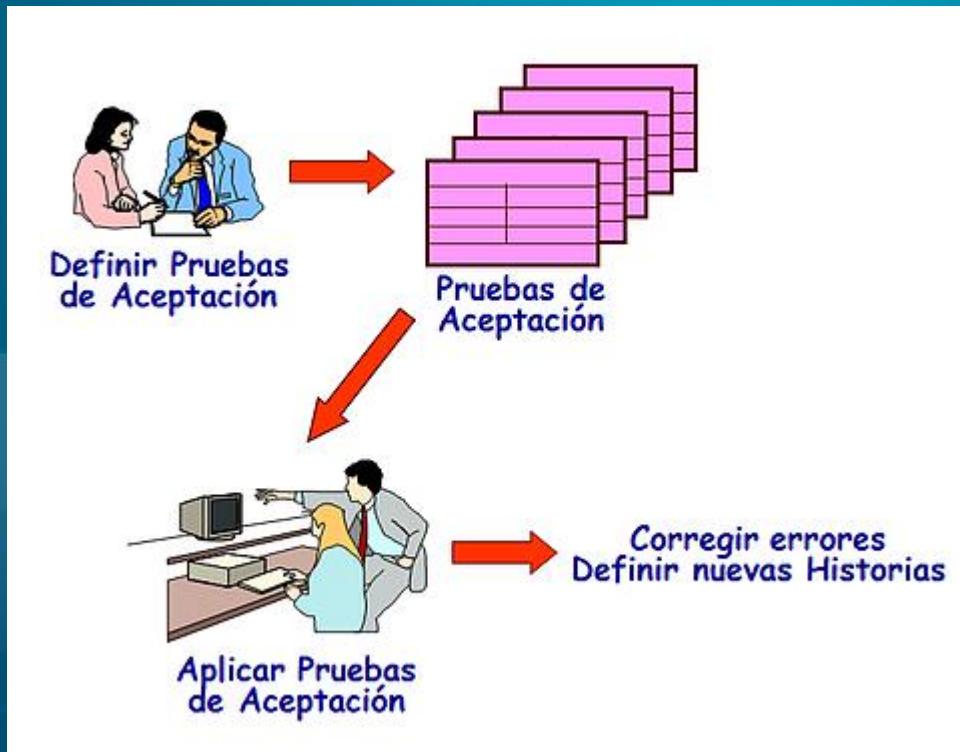
Codificación y pruebas



5.- Metodologías.

Metodologías ágiles. Programación extrema

- Las pruebas unitarias son la medida de comprobación de la funcionalidad de cada uno de los módulos o componentes del sistema, estas pueden ser ejecutadas a diario y brindan una información del avance que tiene el proyecto.



5.- Metodologías.

Metodologías ágiles. Programación extrema

- ¿Ventajas e inconvenientes?

5.- Metodologías.

Metodologías ágiles. Programación extrema

- Ventajas
 - Facilita la creación de un proyecto de software, ya que es eficiente y flexible, esta tiene técnicas y herramientas que le facilitan llegar a una meta la cual es satisfacer al cliente mediante la entrega oportuna de un software de calidad.
 - Uno de los más grandes ejemplos de sistemas de software que implementan la Programación extrema, es el software libre, el cual no tiene una excelente documentación, pero si, en muchos casos, excelencia de calidad.

5.- Metodologías.

Metodologías ágiles. Programación extrema

- Inconvenientes

- Las pruebas constantes podrían ocasionar una pérdida de tiempo en el diseño, por lo que el desarrollo podría volverse no tan ligero.

5.- Metodologías.

Metodologías ágiles. SCRUM

- Es una metodología incremental e iterativa que se utiliza a menudo en el **desarrollo ágil**. Es un marco de trabajo para el desarrollo ágil de software.
- Se pretende que un equipo con **funciones cruzadas** participe para sacar adelante el proyecto de tal manera que las **diferentes actividades a realizar se solapen** mucho.
- Aceptando que el problema no puede entenderse completamente desde el principio, se intenta que el equipo pueda realizar entregas de software cada poco tiempo y responder así a los cambios que se vayan produciendo.

5.- Metodologías.

Metodologías ágiles. SCRUM

- Los roles principales en Scrum son:
 - ScrumMaster, que mantiene los procesos y trabaja de forma similar al director de proyecto.
 - ProductOwner, que representa a los stakeholders (interesados externos o internos). Se asegura de que el equipo Scrum trabaje de forma adecuada desde la perspectiva del negocio.
 - Team que incluye a los desarrolladores.

5.- Metodologías.

Metodologías ágiles. SCRUM

- Los roles auxiliares en Scrum son:
 - Stakeholders (clientes, proveedores, vendedores, etc.), son las personas que hacen posible el proyecto y para quienes el proyecto producirá el beneficio acordado que justifica su desarrollo. Participan durante las revisiones del sprint
 - Managers, son los responsables de establecer el entorno para el desarrollo del proyecto.

5.- Metodologías.

Metodologías ágiles. SCRUM

- La **planificación del trabajo** se realiza en lo que se conoce como “**Sprint**”, cuya duración es recomendable que sea siempre la misma. Podemos establecer *sprints* de una semana, quince días o, como mucho, un mes.
- En cada Sprint se crea un incremento de software potencialmente entregable.
- El conjunto de características de cada Sprint viene de Product Backlog, que es un conjunto de requisitos que definen el trabajo a realizar.
- Los elementos del Product Backlog que forman parte del Sprint se determinan durante la reunión de Sprint Planning.
- En esta reunión, el Product Owner identifica los elementos del Product Backlog que se desean ver completados.

5.- Metodologías.

Metodologías ágiles. SCRUM

Stories	To Do		In Progress	Testing	Done
 This is a sample text. Replace it with your own text.	 This is a sample text. Replace it with your own text.	 This is a sample text. Replace it with your own text.	 This is a sample text.	 This is a sample text.	 This is a sample text. Replace it with your own text.
	 This is a sample text. Replace it with your own text.	 This is a sample text. Replace it with your own text.	 This is a sample text.	 This is a sample text.	 This is a sample text. Replace it with your own text.
 This is a sample text. Replace it with your own text.	 This is a sample text.	 This is a sample text.	 This is a sample text.	 This is a sample text.	 This is a sample text. Replace it with your own text.
	 This is a sample text.	 This is a sample text.	 This is a sample text. Replace it with your own.	 This is a sample text.	 This is a sample text.

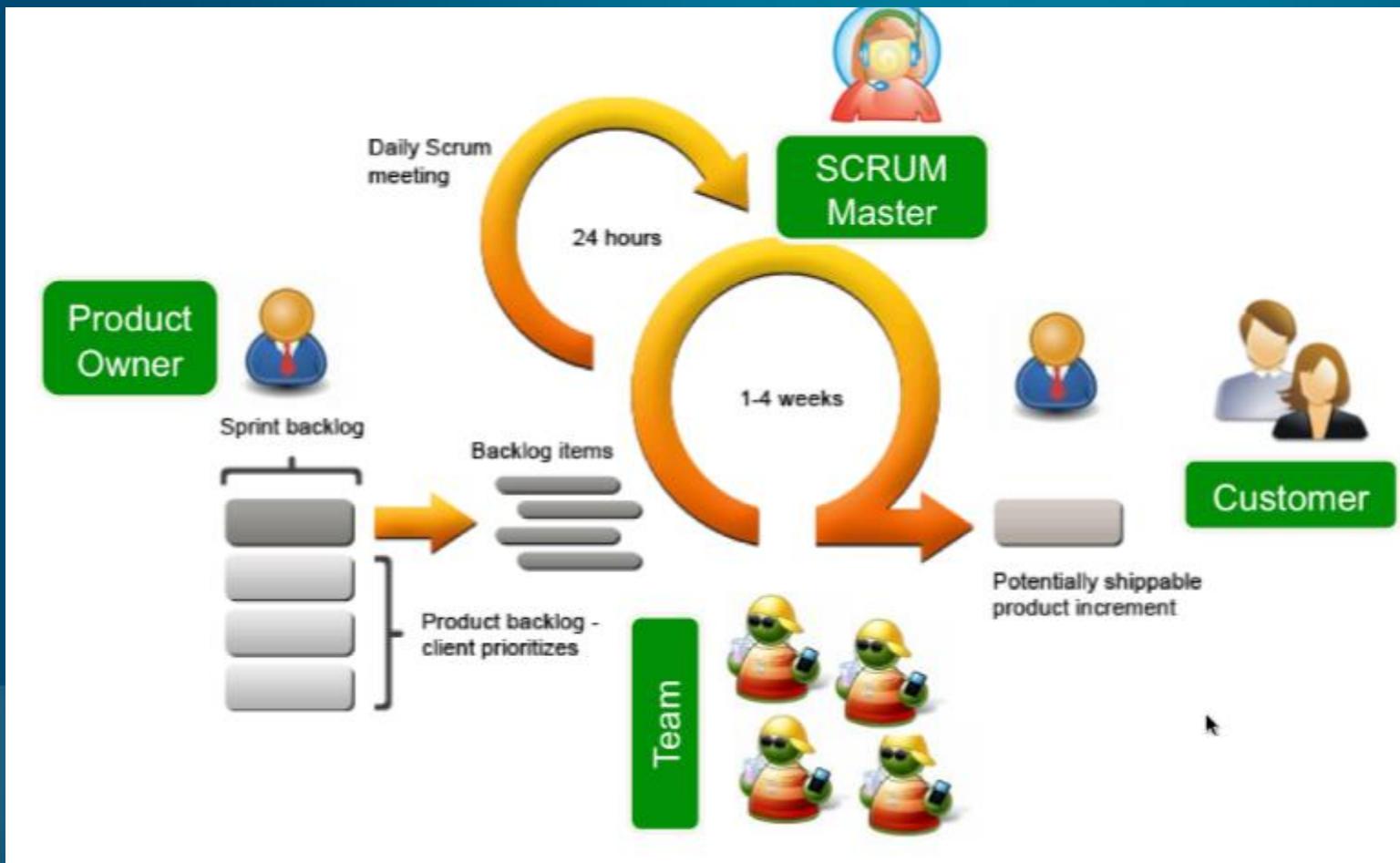
5.- Metodologías.

Metodologías ágiles. SCRUM

- El objetivo es que al final de cada sprint, el equipo deberá presentar los avances logrados y los resultados obtenidos, así como justificar porqué ciertas tareas no se han terminado en el caso de que haya sido así.

5.- Metodologías.

Metodologías ágiles. SCRUM



<https://www.youtube.com/watch?v=PlHc60egiQ&t=32s>

5.- Metodologías.

Métrica v.3

- Es una metodología de planificación, desarrollo y mantenimiento de sistemas de información promovido por la secretaría del Estado de Administraciones Pùblicas del Ministerio de Hacienda y Administraciones Pùblicas.
- Tiene como referencia el modelo del ciclo de vida de desarrollo propuesto por la norma **ISO 12.207**.
- Consta de 3 procesos principales: planificación, desarrollo y mantenimiento.
- Cada proceso se divide en actividades no necesariamente de ejecución secuencial y cada actividad en tareas.

5.- Metodologías.

Métrica v.3

Planificación

- Descripción de la situación actual.
- Un conjunto de modelos con la arquitectura de la información.
- Una propuesta de proyectos a desarrollar en los próximos años y la prioridad de cada uno.
- Una propuesta de calendario de ejecución de los proyectos.
- La evaluación de los recursos necesarios para desarrollar los proyectos del próximo año.
- Un plan de seguimiento y control de lo propuesto.

5.- Metodologías.

Métrica v.3

Desarrollo

- 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).

5.- Metodologías.

Métrica v.3

Mantenimiento

- Nueva versión del sistema de información a partir de las peticiones de mantenimiento que los usuarios realizan con motivo de problemas detectados o por la mejora del mismo.



Unidad didáctica 1. Actividad 1.

6.- Bibliografía

Ciclo Superior Desarrollo de aplicaciones multiplataforma
IES Muralla Romana

6.- Bibliografía

- FERNÁNDEZ LAMEIRO, María del Carmen, FERNÁNDEZ LÓPEZ, Máximo, DEL RÍO RODRÍGUEZ, Andrés. *Unidade didáctica 01 (Desenvolvemento de software), actividade 01 (Proxectos de desenvolvemento) do módulo Contornos de desenvolvemento.* Xunta de Galicia, Consellería de Cultura, Educación e Ordenación Universitaria. 2014.
- AYCART PÉREZ, David. GIBERT GINESTA, Marc HERNÁNDEZ MATÍAS, Martín, MAS HERNÁNDEZ, Jordi. Ingeniería de software en entornos de SL. Universitat Ouberta de Catalunya.

<http://openaccess.uoc.edu/webapps/o2/bitstream/10609/214/1/Ingenier%C3%A0del%20software%20en%20entornos%20del%20software%20libre.pdf>