

# Ingeniería del Software I

## Tema 3. Procesos del Software (Metodologías)

El proceso de ingeniería del software. Modelos de procesos. Modelo Lineal Secuencial (cascada). Modelo de Prototipos. Modelo Rápido de Aplicaciones (DRA). Modelos Evolutivos (incremental, espiral, victoria victoria, DUM, etc.). Modelo de desarrollo concurrente y basado en componentes. Modelos de Métodos Formales. Proceso Unificado de Desarrollo (PUDs). Modelo Metodologías ágiles (scrum, xp, crystal, etc.). Herramientas.

Como el software, al igual que el capital, es el conocimiento incorporado, y puesto que el conocimiento está inicialmente disperso, el desarrollo del software implícito, latente e incompleto en gran medida, es un proceso social de aprendizaje. El proceso es un diálogo en el que se reúne el conocimiento y se incluye en el software para convertirse en software. El proceso proporciona una interacción entre **los usuarios y los** diseñadores, entre los usuarios y las herramientas de desarrollo, y entre **lo diseñadores y las herramientas** de desarrollo [tecnología]. **Es un proceso interactivo donde la herramienta de desarrollo se usa como** medio de comunicación, con cada iteración del diálogo se obtiene mayor conocimiento de las personas involucradas. Hward Baetjer

# Ingenieria del Software: una tecnologia estratificada

- La ingeniería del software es el establecimiento y uso de principios robustos de la ingeniería a fin de obtener económicamente software que sea fiable y que funcione eficientemente sobre máquinas reales.
- ¿Cuáles son los «principios robustos de la ingeniería» aplicables al desarrollo de software de computadora?
- ¿Cómo construimos el software «económicamente» para que sea «fiable»?
- ¿Qué se necesita para crear programas de computadora que funcionen «eficientemente» no en una máquina si no en diferentes «máquinas reales»?

El fundamento de la ingeniería del software es la capa de *proceso*.

*El proceso de la ingeniería del software* es la unión que mantiene juntas las capas de tecnología y que permite un desarrollo racional y oportuno de la ingeniería del software.

El proceso define un marco de trabajo para un conjunto de *Áreas clave de proceso* que se deben establecer para la entrega efectiva de la tecnología de la ingeniería del software. Las áreas claves del proceso forman la base del control de gestión de proyectos del software y establecen el contexto en el que se aplican los métodos técnicos, se obtienen productos del trabajo (modelos, documentos, datos, informes, formularios, etc.), se establecen hitos, se asegura la calidad y **el cambio se gestiona adecuadamente**.

# Métodos

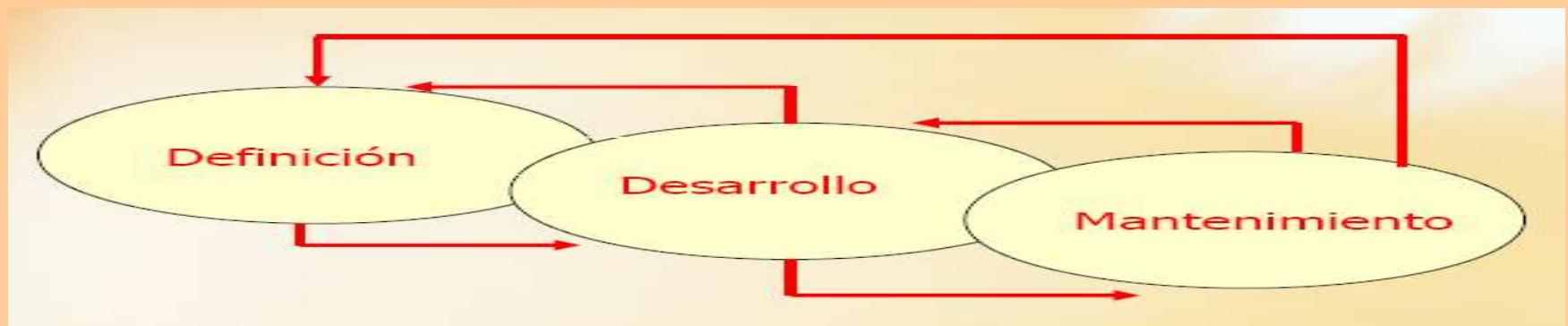
- Los *métodos de la ingeniería del software indican* «cómo» construir técnicamente el software. Los métodos abarcan una gran gama de tareas que incluyen **análisis** de requisitos, diseño, construcción de programas, pruebas y mantenimiento. Los métodos de la ingeniería del software dependen de un conjunto de principios básicos que gobiernan cada área de la tecnología e incluyen actividades de modelado y otras técnicas descriptivas.

# Métodos

- Los métodos indican cómo construir técnicamente el software.
- Tareas que componen los métodos.
  - Planificación; Estimación de proyectos.
  - Análisis de requerimientos del software hardware.
  - Diseño de estructuras de datos, Arquitectura de los programas.
  - Procedimientos algorítmicos.
  - Codificación; Prueba; y Mantenimiento.

El trabajo que se asocia a la ingeniería del software se puede dividir en tres fases genéricas, con independencia del área de aplicación, tamaño o complejidad del proyecto. Cada fase se encuentra con una o varias cuestiones de las destacadas anteriormente.

- *La fase de definición.* Se centra sobre el *qué*. Es decir, durante la definición, el que desarrolla el software intenta identificar qué información ha de ser procesada, qué función y rendimiento se desea, qué comportamiento del sistema, qué interface van a ser establecidas, qué restricciones de diseño existen, y qué criterios de validación se necesitan para definir un sistema correcto.
- *La fase de desarrollo.* Se centra en el *cómo*. Es decir, durante el desarrollo un ingeniero del software intenta definir cómo han de diseñarse las estructuras de datos, cómo ha de implementarse la función dentro de una arquitectura de software, cómo han de implementarse los detalles procedimentales, cómo han de caracterizarse interfaces, cómo ha de traducirse el diseño en un lenguaje de programación (o lenguaje no procedural) y cómo ha de realizarse la prueba.
- *La fase de mantenimiento.* se centra en el *cambio* que va asociado a la corrección de errores, a las adaptaciones requeridas a medida que evoluciona el entorno del software y a cambios debidos a las mejoras producidas por los requisitos cambiantes del cliente.



## En Mantenimiento ...

- **Corrección.** Incluso llevando a cabo las mejores actividades de garantía de calidad, es muy probable que el cliente descubra los defectos en el software. El *mantenimiento correctivo cambia el software para corregir los defectos*.
- **Adaptación.** Con el paso del tiempo, es probable que cambie el entorno original (por ejemplo: CPU, el sistema operativo, las reglas de empresa, las características externas de productos) para el que se desarrolló el software. El *mantenimiento adaptativo produce modificación en el software para acomodarlo a los cambios de su entorno externo*.
- **Mejora.** Conforme se utilice el software, el cliente/ usuario puede descubrir funciones adicionales que van a producir beneficios. El *mantenimiento perfectivo* lleva al software más allá de sus requisitos funcionales originales.
- **Prevención.** El software de computadora se deteriora debido al cambio, y por esto el *mantenimiento preventivo* también llamado *reingeniería del software*, se debe conducir a permitir que el software sirva para las necesidades de los usuarios finales. En esencia, el mantenimiento preventivo hace cambios en programas de computadora a fin de que se puedan corregir, adaptar y mejorar más fácilmente.

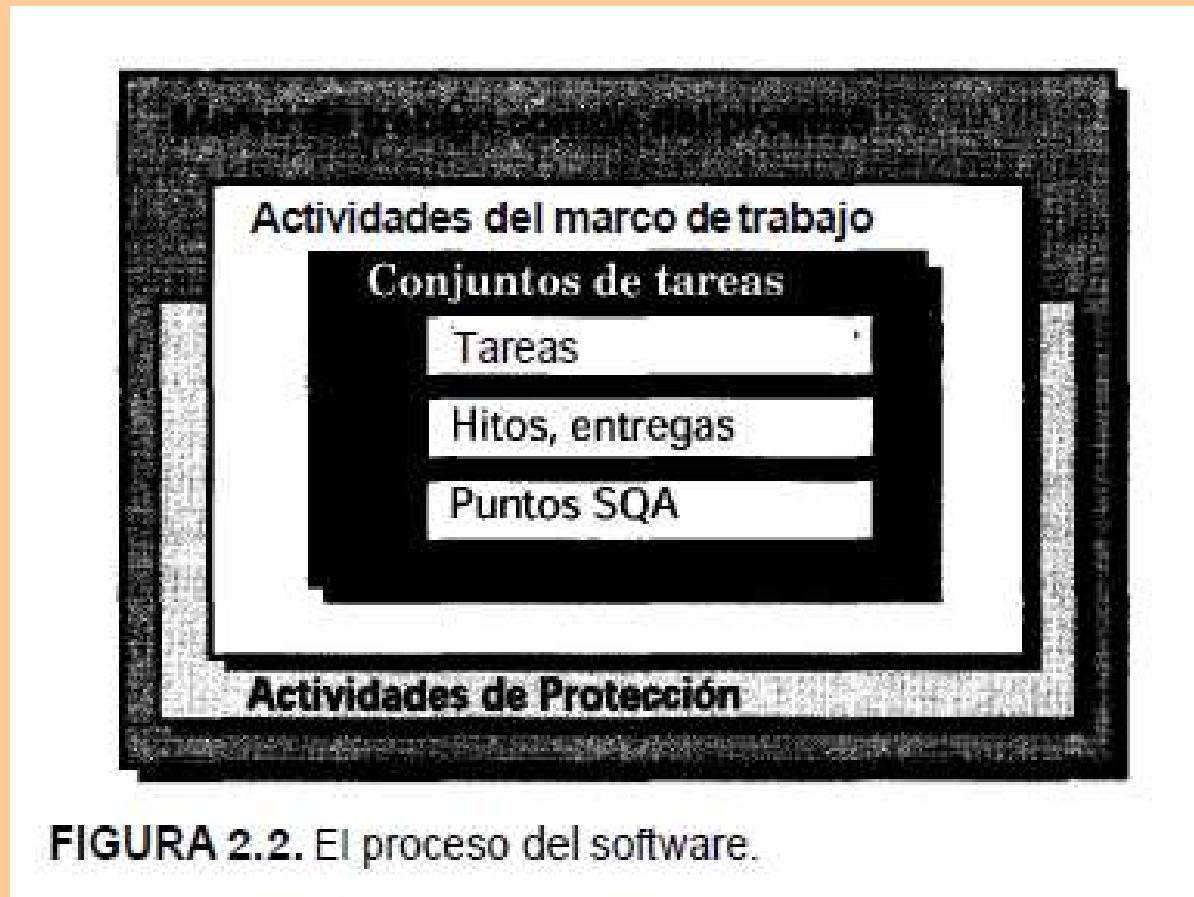
# PROCESO

Son los encargados de integrar los métodos y herramientas, además de definir la secuencia en la que se aplican los métodos, las entregas que requieren, los controles de calidad y las guías para el desarrollo.

Los métodos y procedimientos → un metodología de desarrollo para el modelo

Las herramientas ayudan en cualquiera de los modelos

# El Proceso del software



# Fases Genéricas de la IS

## □ Definición. Tareas que la componen:

- Análisis del sistema.
- Planificación del Proyecto.
- Análisis de requisitos.

III → QUÉ

## □ Desarrollo. Tareas que la componen:

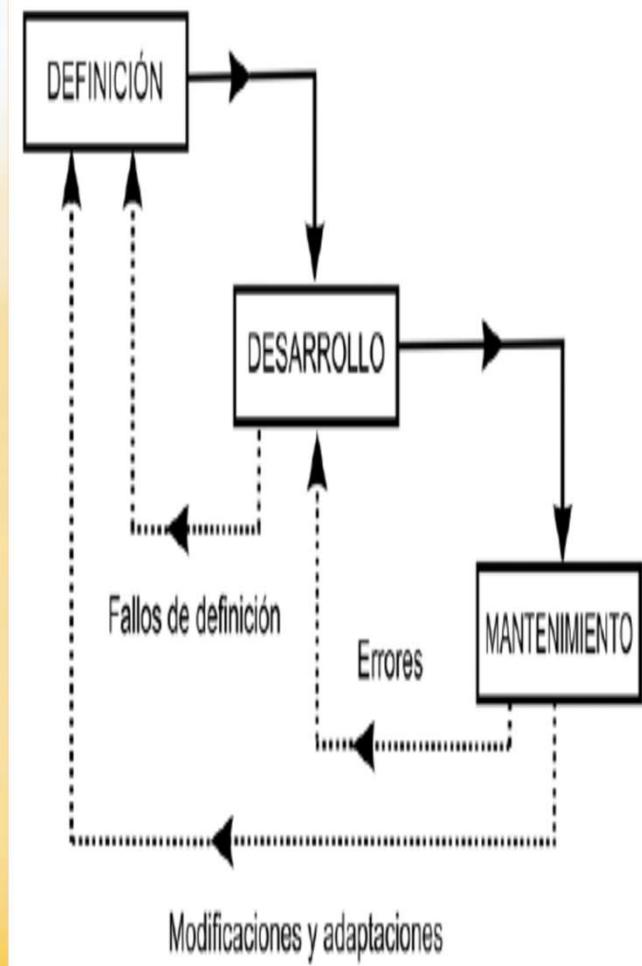
- Diseño del software.
- Codificación.
- Prueba del Software.

III → CÓMO

## □ Mantenimiento. Tipos de cambios:

- Corrección.
- Adaptación.
- Mejora.
- Prevención o Reingeniería.

III → TIPO



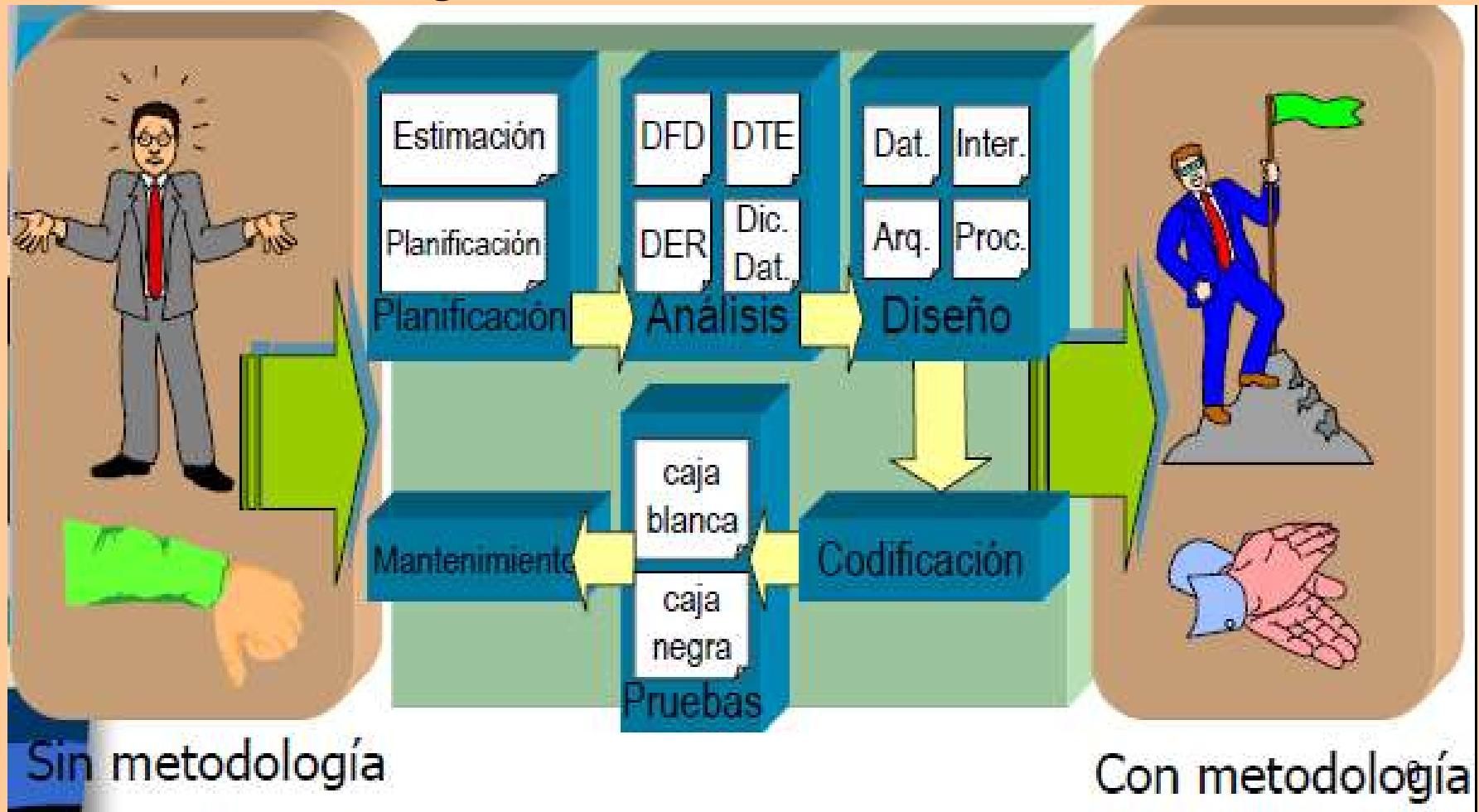
# Paradigma de la IS

- El modelo de proceso o paradigma de la IS es la estrategia que comprenden métodos, herramientas y procesos.
- El ingeniero debe seleccionar un modelo de proceso para ingeniería del software según la naturaleza del proyecto y de la aplicación, los métodos, las herramientas a utilizar, y los controles y entregas que se requieren.
- Los diferentes paradigmas lo que intentan es ordenar las actividades en el desarrollo del software, de manera que no sean llevadas a cabo de manera caótica.

# Paradigmas de la IS

- Ciclo de vida clásico – Modelo lineal secuencial – Modelo en cascada.
- Prototipos.
- Modelo DRA (RAD. Rapid Application Development).
- Modelos Evolutivos de Proceso
  - Incremental.
  - Espiral.
  - Espiral WINWIN.
  - Modelo de desarrollo concurrente.
- Modelo basado en componentes.
- Modelo basado en métodos formales.
- Técnicas de cuarta generación.
- Modelos ágiles. Programación extrema.

# Informática Metodologías: Aplicación a la Ingeniería del Software



# Metodologías: Elementos

Define una estrategia global para enfrentarse con el proyecto:

- **Fases.**
  - Tareas a realizar en cada fase.
- **Productos (final e intermedios).**
  - E/S de cada fase, documentos.
- **Procedimientos y herramientas.**
  - Apoyo a la realización de cada tarea.
- **Criterios de evaluación.**
  - Del proceso y producto. Saber si se han logrado los objetivos.

# Metodología

Su uso, desde distintas miradas:

- Gestión
- Desarrollador
- Cliente / usuario

# Metodologías: Ventajas

Desde el punto de vista de gestión:

- Facilitar la tarea de planificación.
- Facilitar la tarea de control y seguimiento de un proyecto.
- Mejorar la relación coste/beneficio.
- Optimizar el uso de recursos disponibles.
- Facilitar la evaluación de resultados y cumplimiento de los objetivos.
- Facilitar la comunicación efectiva entre usuarios y desarrolladores.
- Ayuda a la gestión del proyecto.

# Metodologías: Ventajas

Desde el punto de vista de los ingenieros del software:

- Ayudar a la comprensión del problema.
- Optimizar el conjunto y cada una de las fases del proceso de desarrollo.
- Facilitar el mantenimiento del producto final.
- Permitir la reutilización de partes del producto.

# Metodologías: Ventajas

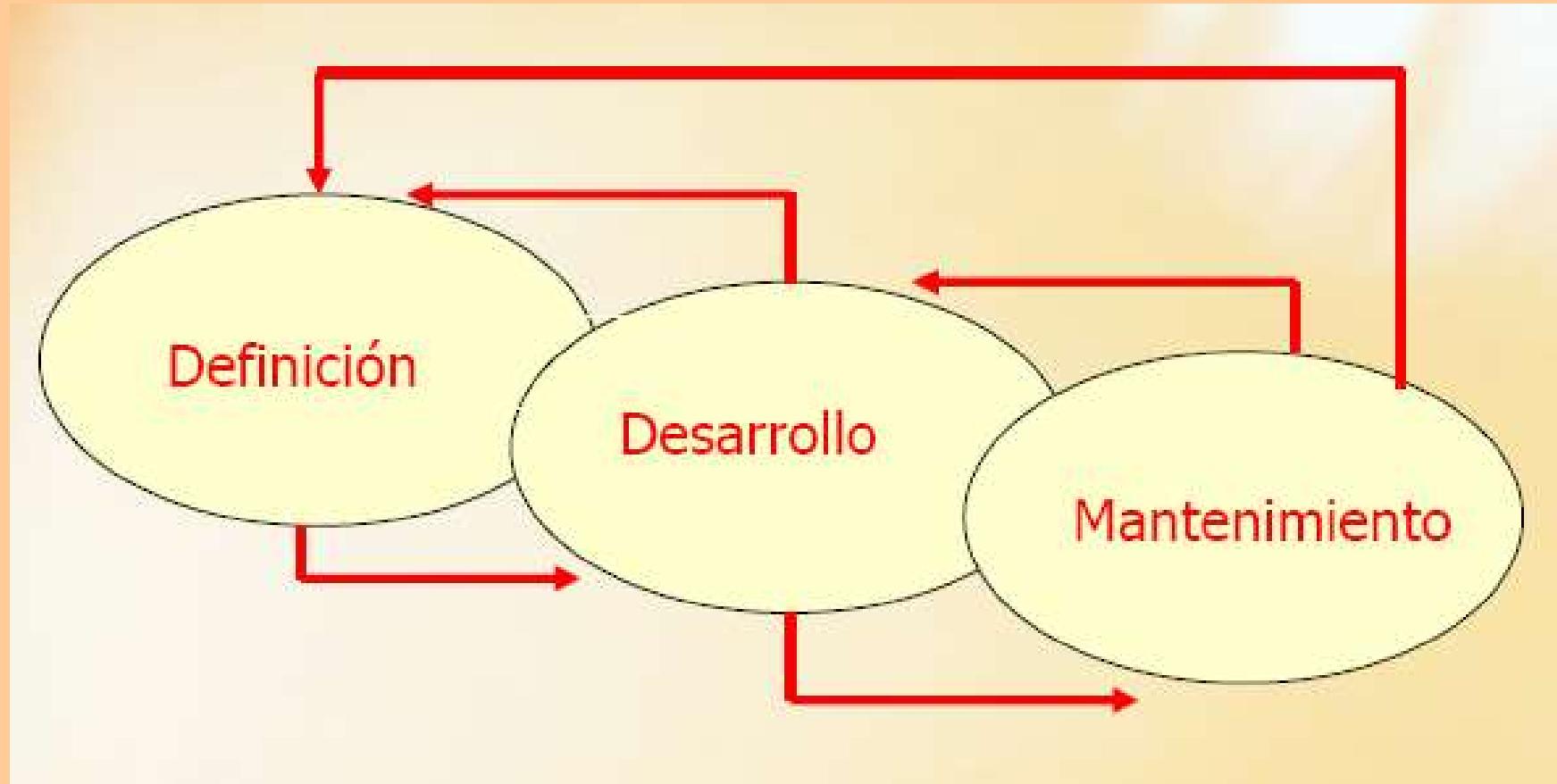
Desde el punto de vista del cliente o usuario final:

- Garantía de un determinado nivel de calidad en el producto final.
- Confianza en los plazos de tiempo fijados en la definición del proyecto.

# Metodologías: Funciones básicas

- Definir el ciclo de vida que más se adecue a las condiciones y características del desarrollo.
- Determinar las fases dentro del ciclo de vida especificando su orden de ejecución.
- Definir los resultados intermedios y finales.
- Proporcionar un conjunto de métodos, herramientas y técnicas para facilitar la tarea del ingeniero del software y aumentar su productividad.

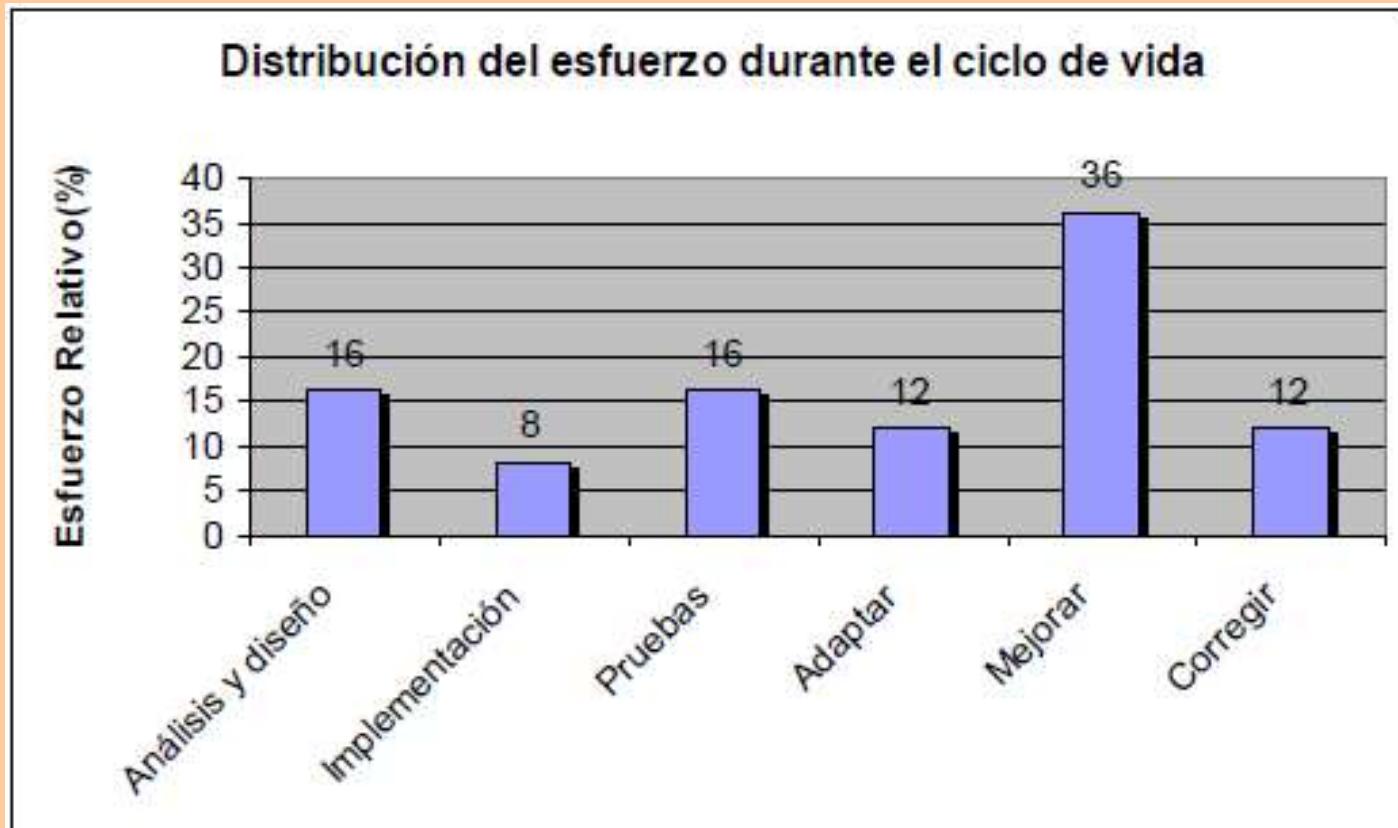
# Visión Generalizada de los Paradigmas



# Ciclo de vida: Definición

- Conjunto de fases por las que pasa el sistema que se está desarrollando desde que nace la idea inicial hasta que el software es retirado o reemplazado (“muere”).
- Se denomina a veces “*paradigma*”.
- Dos puntos de vista
  - Transformación del producto.
  - Proceso que transforma el producto.

# Ciclo de vida: Distribución del esfuerzo



# Modelo Clásico Secuencial

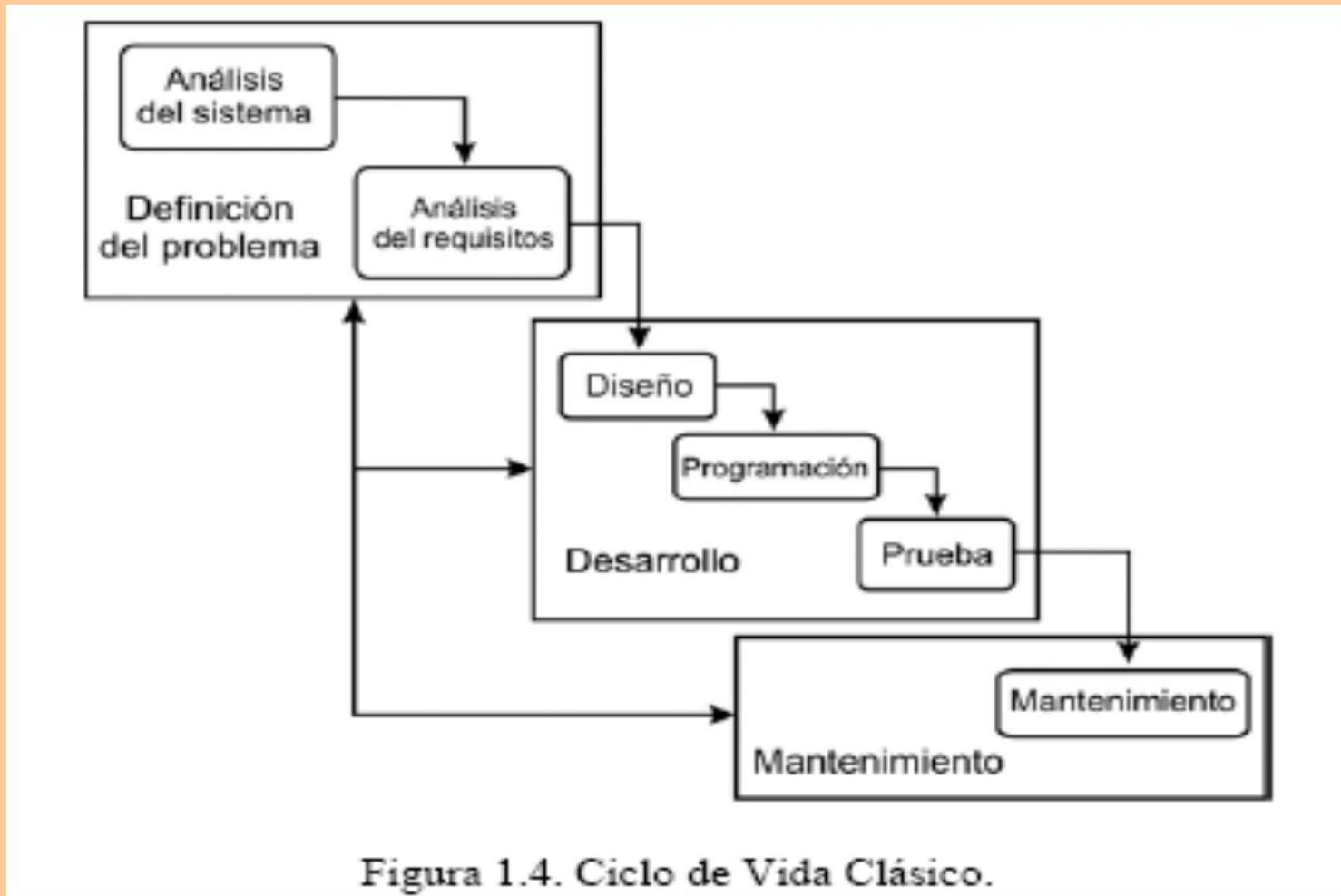


Figura 1.4. Ciclo de Vida Clásico.

# Problemas del Modelo Sec. Lineal

- Raramente los proyectos siguen este ciclo de vida.
- El cliente pocas veces establece todos los requerimientos al principio.
- El cliente no tiene un producto hasta el final.
- Tiempo largo de desarrollo

# Modelo Lineal Secuencial

## Fases

De definición (Análisis de Sistemas y Análisis de Requisitos)

- o Identificar las necesidades del cliente
- o Captar y definir el sistema de información como un conjunto de partes con un fin común y una serie de interrelaciones múltiples.
- o Dividir el sistema en subsistemas de información independiente pero sin perder de vista sus relaciones con otros subsistemas.
- o Escoger el subsistema a automatizar y evaluar la viabilidad técnica y económica de dicho proceso.
- o Establecer restricciones de coste y plazo.
- o Asignar las funciones y rendimientos esperados del hardware, software, personal y bases de datos.

# Modelo Lineal Secuencial

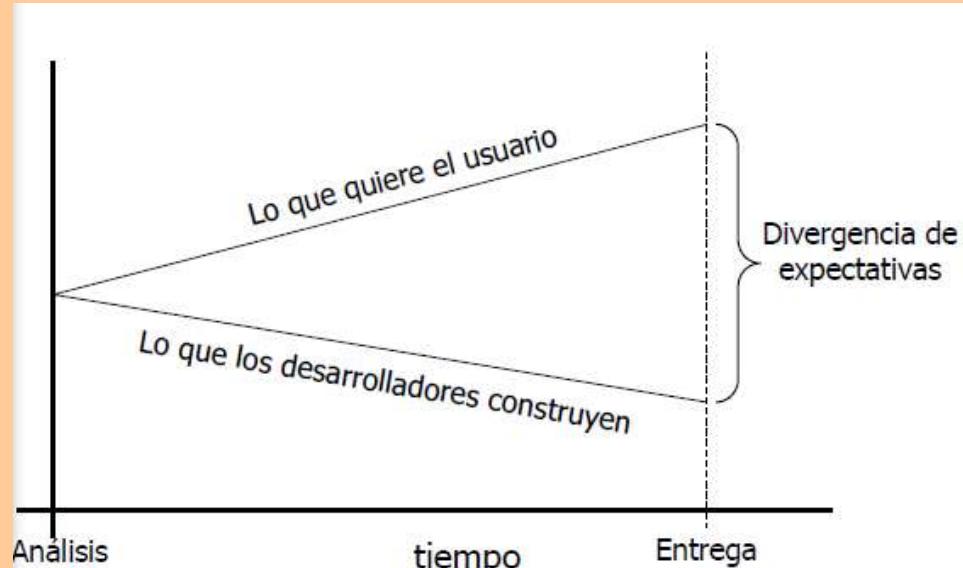
Fase de desarrollo

Diseño

Programación

Prueba

Fase de Mantenimiento



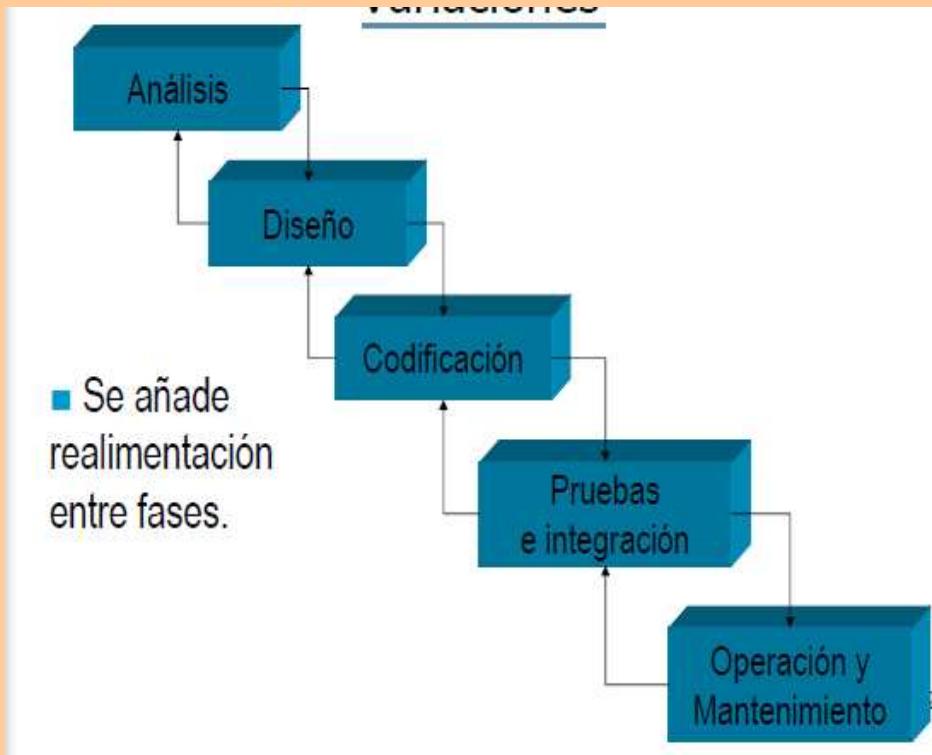
Fase de Paralelas a las diferentes etapas

*Creación de la base de datos del organismo*

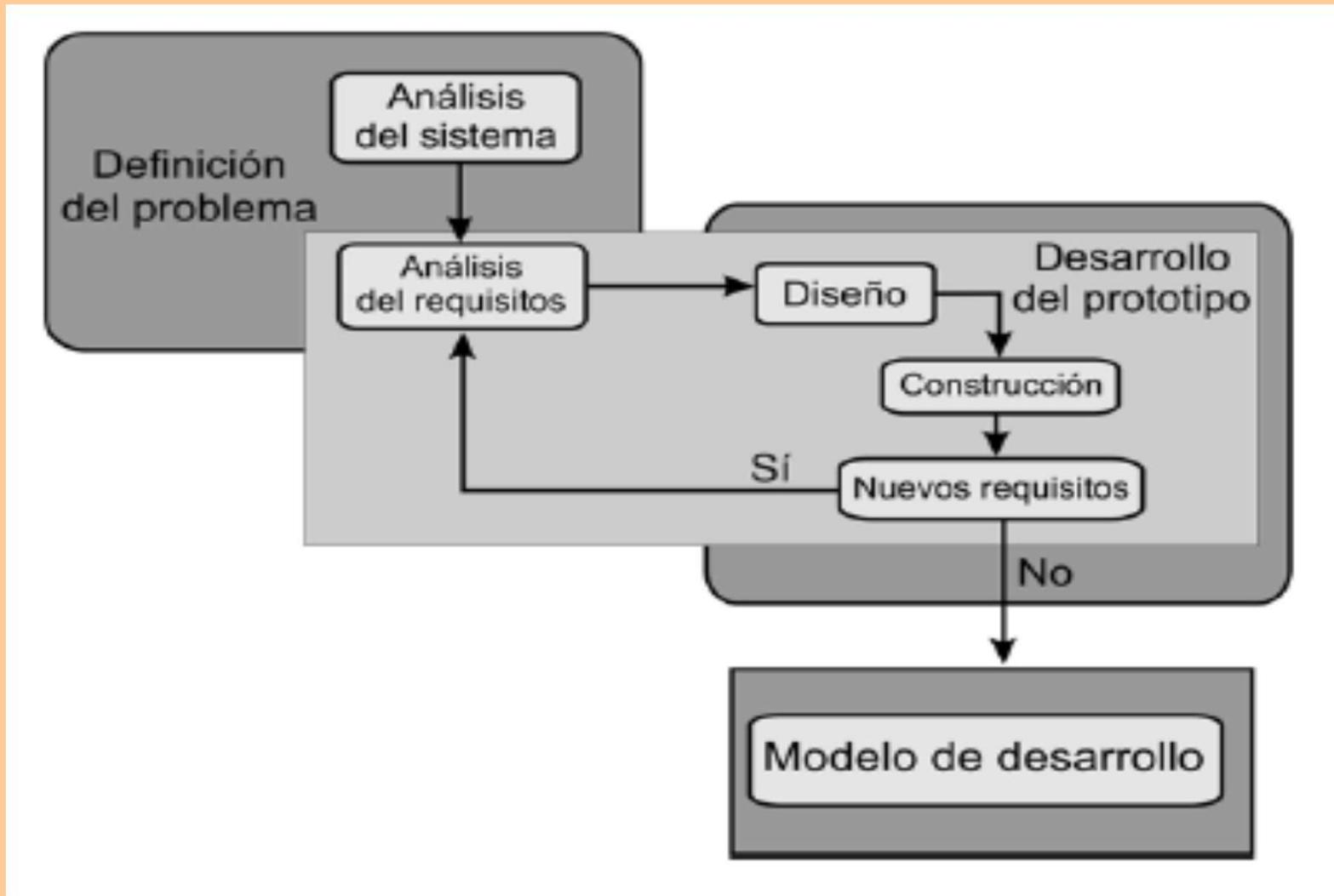
*Formación y manuales del administrador y del usuario*

*Petición de equipos y soportes, locales*

# Modelo de ciclo de vida en cascada. Variaciones



# Ciclo de Vida del Prototipado



# **Ciclo de Vida del Prototipado**

Fases dentro del ciclo de vida del prototipado:

1. Fase de Recolección de Requisitos
2. Fase de Diseño Rápido
3. Fase de Construcción del Prototipo
4. Fase de Evaluación del Prototipo
5. Refinamiento sucesivo
6. Terminación

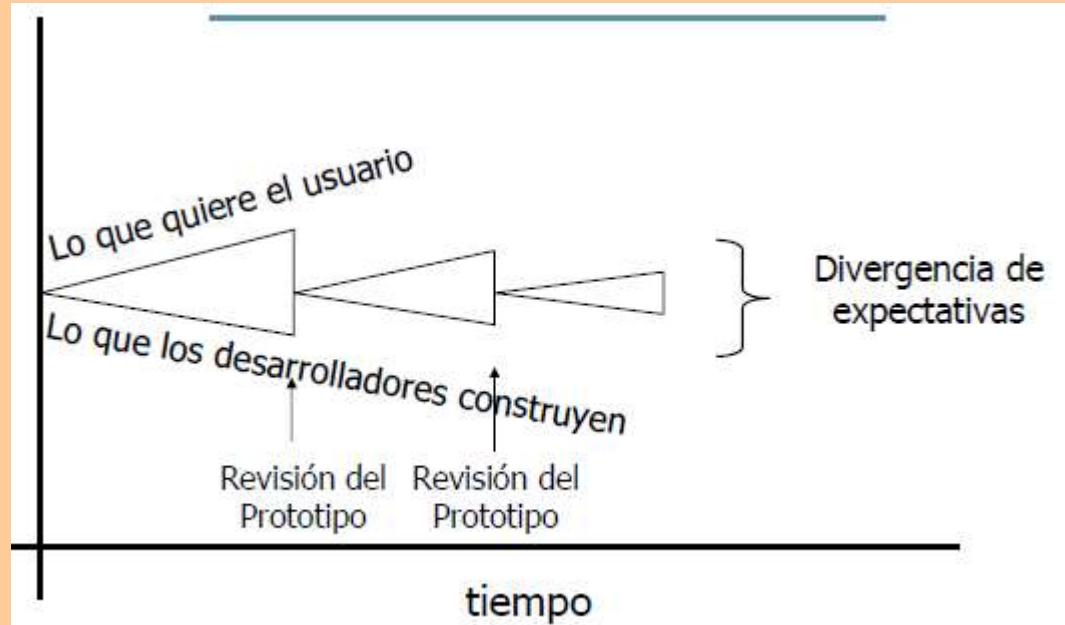
# Modelo de Construcción de Prototipos

¿Por qué se usa este Modelo?

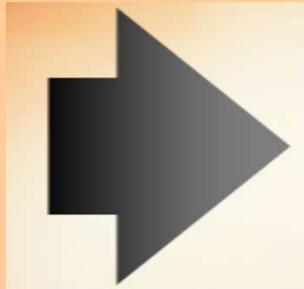
- El cliente no puede especificar todos los requerimientos al principio.
- Existen dudas de alguna parte del sistema.
- Facilita un modelo al programador.

# Tipos de Prototipos

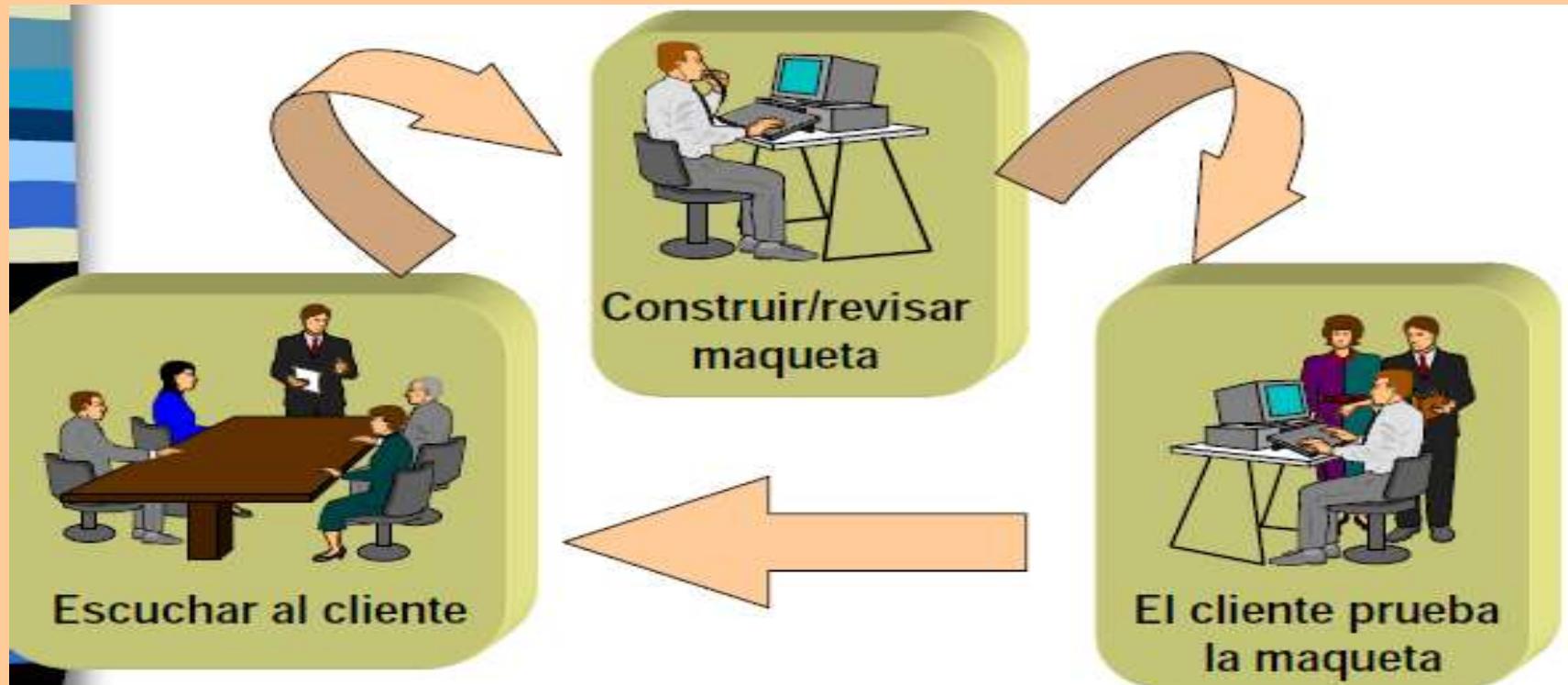
- Totales.
- Parciales.
  - Interfaces.
  - Modelos.
  - Estructuras de datos.



# Problemas del Prototipado



- El cliente lo quiere, aunque no es un producto software
- Módulos ineficientes se convierten en partes del sistema.



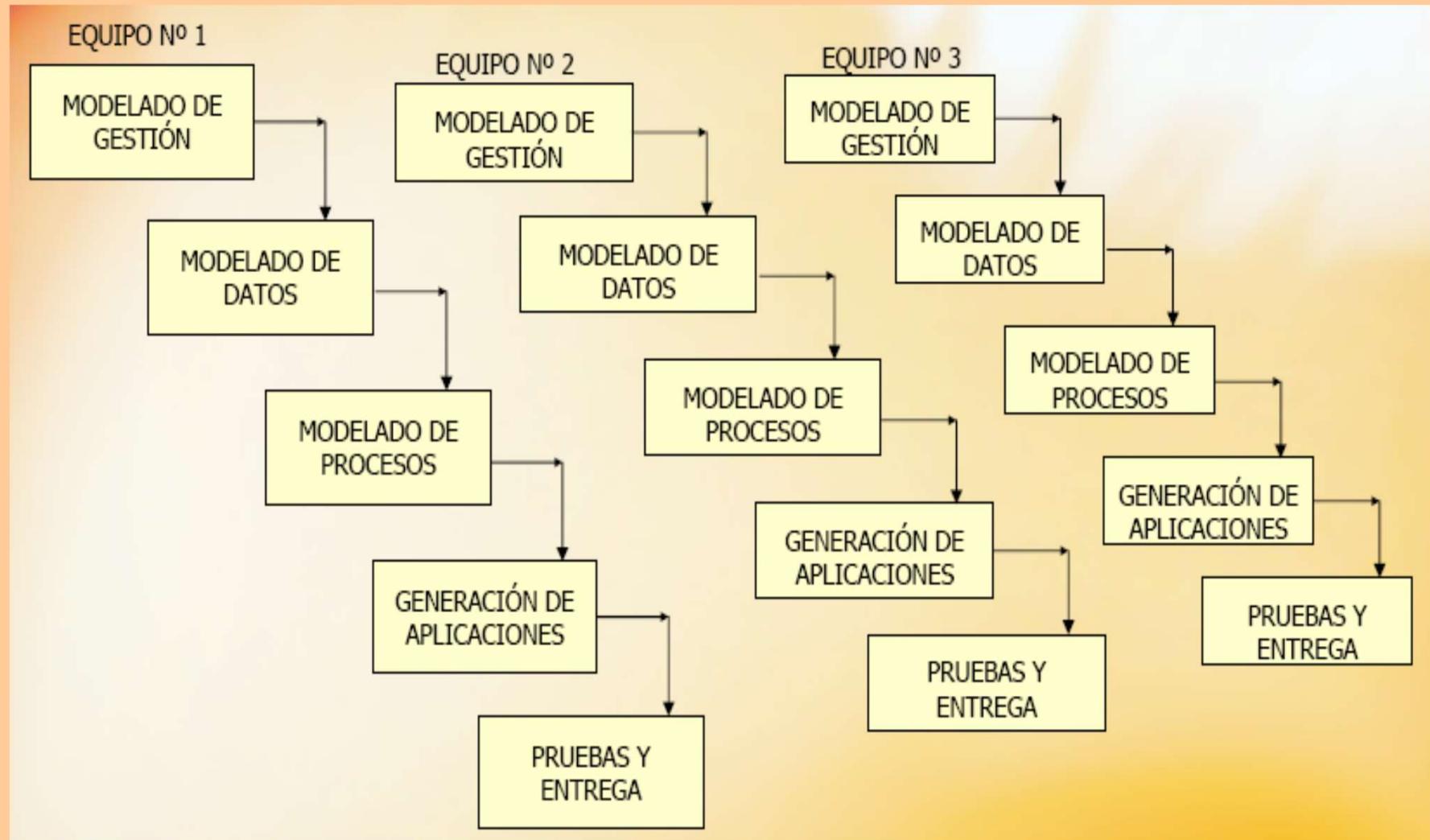
# Modelo DRA

El Modelo DRA consiste en un desarrollo rápido de aplicaciones basado en el modelo lineal secuencial, pero donde se enfatiza un ciclo de desarrollo extremadamente corto.

Es una adaptación a alta velocidad del modelo lineal secuencial, donde se puede aumentar la velocidad haciendo uso de componentes.

Si se comprenden bien los requisitos y se limita el ámbito del proyecto, el proceso DRA permite al equipo de desarrollo crear un sistema completamente funcional, dentro de períodos cortos de tiempo.

# Modelo DRA



# Problemas del Modelo DRA

- Para proyectos grandes necesitamos de recursos suficientes para formar los equipos necesarios.
- Compromiso de colaboración entre desarrolladores y clientes.
- No todas las aplicaciones son susceptibles de aplicar este modelo.
- Cuando los riesgos técnicos son altos, DRA no es apropiado.
- Cuando el grado de interoperatividad con programas ya existentes es alto, no es apropiado.

# Modelos Evolutivos

Los modelos evolutivos se caracterizan porque permiten a los ingenieros del software, desarrollar de manera iterativa, nuevas versiones del software cada vez más completas.

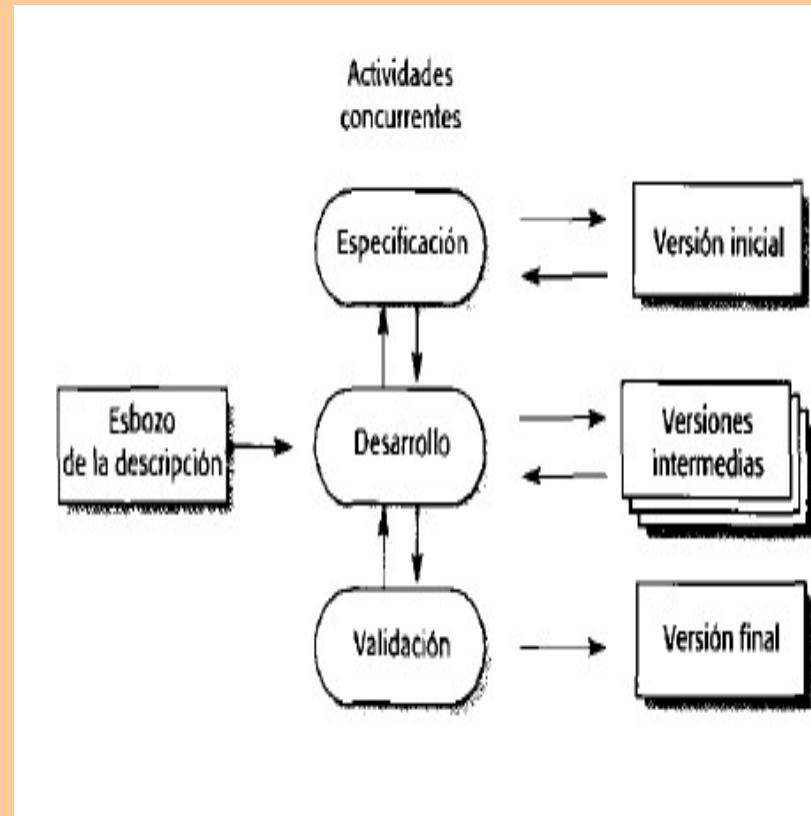
Los modelos que componen este tipo son:

Modelo Incremental.

Modelo en Espiral.

Modelo en Espiral Victoria-Victoria (WINWIN).

Modelo de Desarrollo Concurrente.



# Desarrollo Evolutivo

Existen dos tipos de desarrollo evolutivo:

- *Desarrollo exploratorio, donde el objetivo del proceso es trabajar con el cliente para explorar sus requerimientos y entregar un sistema final.* El desarrollo empieza con las partes del sistema que se comprenden mejor. El sistema evoluciona agregando nuevos atributos propuestos por el cliente.
- *Prototipos desecharables, donde el objetivo del proceso de desarrollo evolutivo es comprender los requerimientos del cliente y entonces desarrollar una definición mejorada de los requerimientos para el sistema.* El prototipo se centra en experimentar con los requerimientos del cliente que no se comprenden del todo.

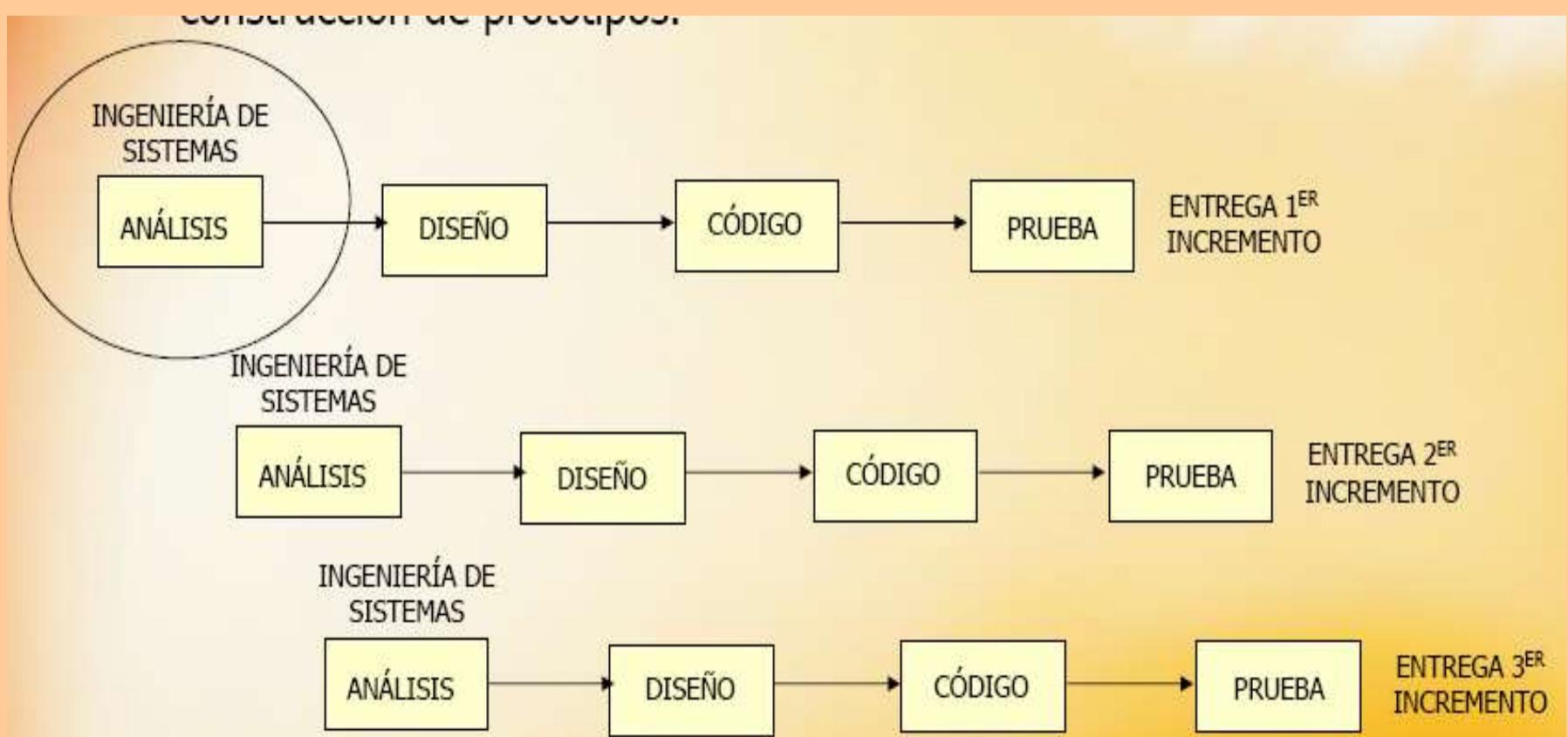
# Desarrollo Evolutivo

Problemas:

- *El proceso no es visible. Los administradores tienen que hacer entregas regulares para medir el progreso. Si los sistemas se desarrollan rápidamente, no es rentable producir documentos que reflejen cada versión del sistema.*
- *A menudo los sistemas tienen una estructura deficiente. Los cambios continuos tienden a corromper la estructura del software. Incorporar cambios en él se convierte cada vez más en una tarea difícil y costosa.*

# Modelo Evolutivo: Incremental

El modelo Incremental combina elementos del modelo lineal secuencial (aplicados repetidamente) con la filosofía interactiva de construcción de prototipos.

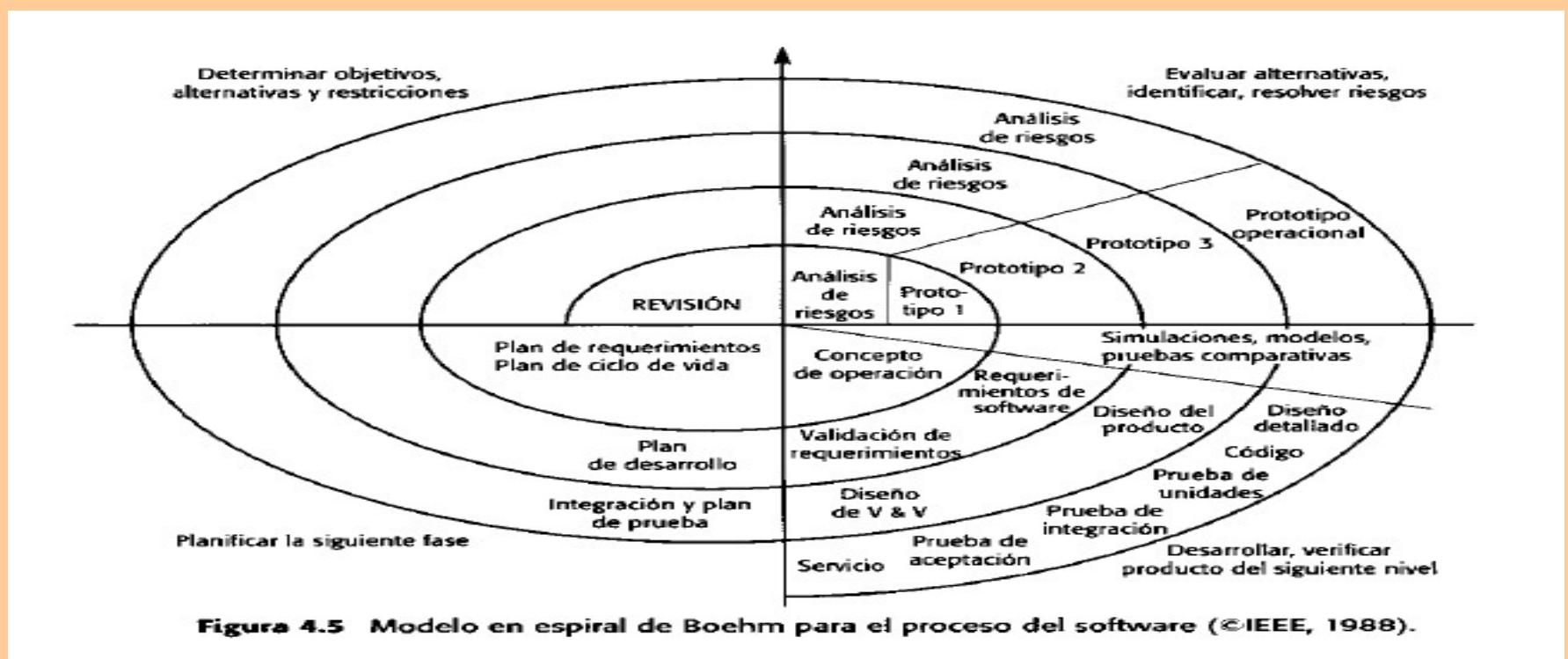


# Modelo en Espiral

El Modelo en Espiral, propuesto originalmente por Boehm, es un modelo de proceso de software evolutivo que conjuga la naturaleza iterativa de construcción de prototipos con los aspectos controlados y sistemáticos del modelo lineal secuencial.

Ideal para realizar versiones incrementales de manera rápida.

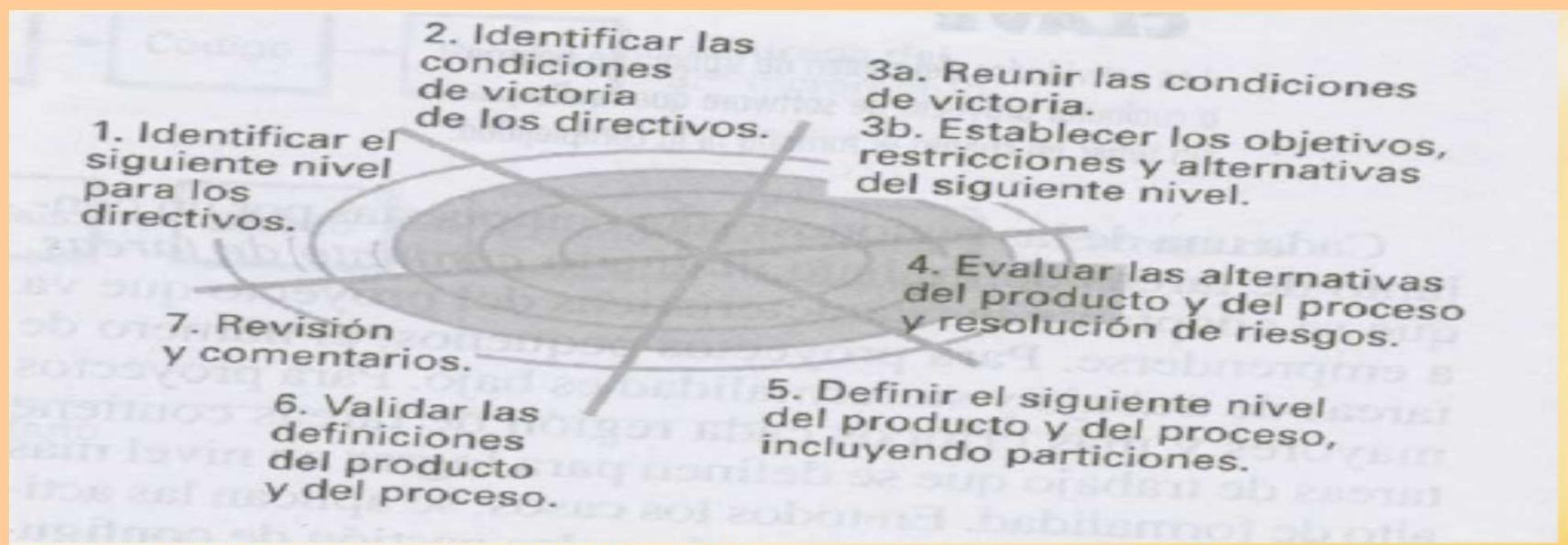
El software se desarrolla en una serie de versiones incrementales.



Mgter. Oscar Adolfo Vallejos – FaCENA - UNNE

# Modelo en Espiral Victoria-Victoria

- El modelo en espiral visto anteriormente, dispone de una actividad de comunicación con el cliente.
  - Comunicación Ideal. El desarrollador pregunta al cliente y el cliente facilita suficiente información para continuar.
  - Comunicación Real. El cliente y el desarrollador entran en un proceso de negociación, donde el cliente puede ser preguntado para sopesar la funcionalidad, rendimiento, y otras características.
- Las mejores negociaciones se esfuerzan en obtener <<victoria - victoria>>.
- Este modelo definido por Boehm, define un conjunto de actividades de negociación al principio de cada paso alrededor de la espiral.



# Metodología DUM

DUM es una metodología evolutiva e incremental de desarrollo del software que ha sido creada en el departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga. Basada en un enfoque iterativo incremental esta metodología realiza una especificación exhaustiva de todas las actividades y tareas que se realizan en las diferentes fases, prestado especial atención por alcanzar un nivel superior de madurez según el marc CMMI/Carnegie Mellon.

Las fases en las que se subdivide la metodología DUM son las siguientes:

- Preliminar
- Inicio
- Elaboración
- Construcción
- Transición
- Mantenimiento.

# **Metodología DUM**

## **Fase Preliminar**

En la Fase Preliminar se llevan a cabo una serie de pasos previos en los que se sientan las bases que permiten comenzar el proyecto. En esta fase el cliente proporciona los dos elementos básicos para comenzar un proyecto: una petición formal del mismo; y una definición del problema al que debe dar respuesta el Sistema a desarrollar. En base a estos dos elementos la organización de desarrollo definirá un primer equipo encargado del inicio del proyecto.

# Metodología DUM

## Fase de Inicio

En la Fase de Inicio se determina si el problema planteado tiene solución, lo cual se hace desde un punto de vista genérico, es decir, no se tienen en cuenta posibles restricciones relacionadas con el cliente como costes económicos o plazos de entrega, sólo se tienen en cuenta restricciones que afecten al problema en sí como pueda ser la legalidad vigente. Si al final de la Fase de Inicio se determina que el problema tiene solución, DUM denominará el Sistema a desarrollar un Sistema Realizable.

En fases posteriores se estudiará si el nuevo Sistema puede desarrollarse teniendo en cuenta las restricciones impuestas por el cliente, esto es lo que DUM denomina un Sistema Viable.

# Metodología DUM

## Fase de Elaboración

- En la Fase de Elaboración se determina si es posible desarrollar el Sistema teniendo en cuenta las restricciones impuestas por el cliente, y se obtiene un proyecto particular después de aplicarle las restricciones del cliente al proyecto genérico.
- Puede considerarse que en las Fases de Inicio y Elaboración se realiza un trabajo en base a objetivos similares pero tratados a distinto nivel. Así, en la Fase de Inicio se trabaja con un proyecto basado en un problema genérico y se comprueba que dicho proyecto genérico se puede llevar a cabo, mientras que en la Fase de Elaboración se trabaja con una versión particular del proyecto genérico y se comprueba que dicho proyecto particular se puede llevar a cabo.

# **Metodología DUM**

## **Fase de Construcción**

- o Terminada la Fase de Elaboración se cuenta con un especificación detallada de qué debe hacer el Sistema, de una arquitectura formada por los elementos del Sistema que guiará cómo cumplirá el Sistema con su cometido, y de una versión inicial del Sistema en base a su arquitectura que será la base sobre la que se construirá el resto del Sistema.
- o En la Fase de Construcción se completarán las labores de desarrollos pendientes para los casos de uso no incluidos en la arquitectura del Sistema de modo que al final de la fase se cuente con una versión completa del Sistema. Esta versión deberá satisfacer todos los requisitos indicados por el cliente y los criterios de calidad y seguridad establecidos por la organización de desarrollo, sin embargo, no será la versión definitiva del Sistema, será la denominada versión beta del Sistema.

# **Metodología DUM**

## **Fase de Transición**

Durante la Fase de Transición se realiza la prueba del Sistema con el fin de adaptar el mismo a un entorno de producción realizándose las modificaciones que se estimen necesarias. Los usuarios encargados de probar el sistema (en adelante usuarios beta) recibirán instrucciones acerca de aquellos aspectos del Sistema en los que deben hacer especial hincapié y sobre el proceso a seguir para la proposición, estudio y resolución de propuestas de modificación. A este respecto será necesario desarrollar la estructura necesaria para facilitar en la medida de lo posible dicho proceso.

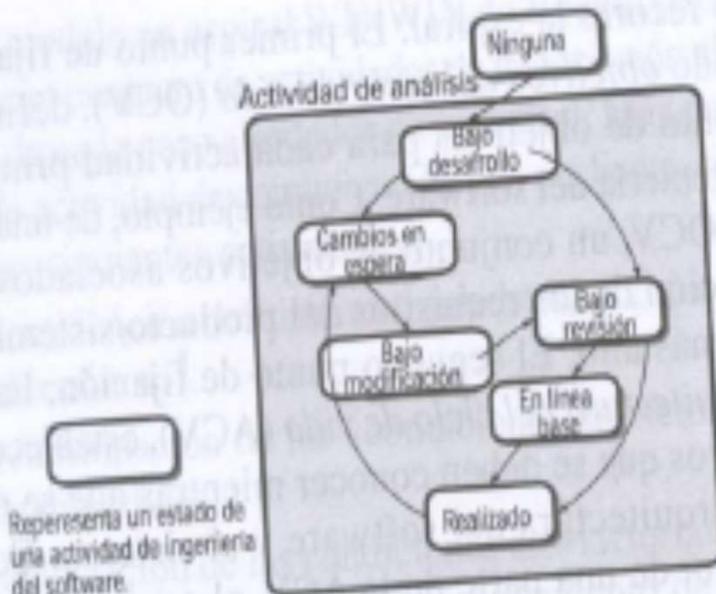
# **Metodología DUM**

## **Fase de Mantenimiento**

Tras la finalización del proyecto será necesario establecer un acuerdo respecto al mantenimiento, que puede ser llevado a cabo por la misma organización de desarrollo o por otra distinta.

# Modelo de Desarrollo Concurrente

- El Modelo de Desarrollo Concurrente dado por Davis y Sitaram, se puede representar en forma de esquem como una serie de actividades técnicas importantes tareas y estados asociados a ellas.
- El modelo Concurrente define una serie de acontecimientos que dispararán transiciones de estado a estado para cada una de las actividades de la Ingeniería del software.
- Este modelo se utiliza a menudo como el paradigma de desarrollo de aplicaciones cliente/servidor.



# Modelo Basado en Componentes

- La tecnología de objetos proporciona el marco de trabajo técnico para un modelo de proceso basado en componentes para la IS.
- El paradigma orientado a objetos enfatiza en la creación de clases que encapsulan tanto los datos como los algoritmos que se utilizan para manejar los datos.
- Si se diseñan e implementan las clases correctamente, podrían ser reutilizables por las diferentes aplicaciones y arquitecturas de sistemas basados en computadores.
- El modelo de desarrollo basado en componentes incorpora muchas de las características del modelo en espiral.
- Es evolutivo por naturaleza y exige un enfoque iterativo para la creación de software.
- Configura aplicaciones desde componentes preparados de software.
- El modelo basado en componentes conduce a la reutilización del software, proporcionando beneficios a los ingenieros de software.

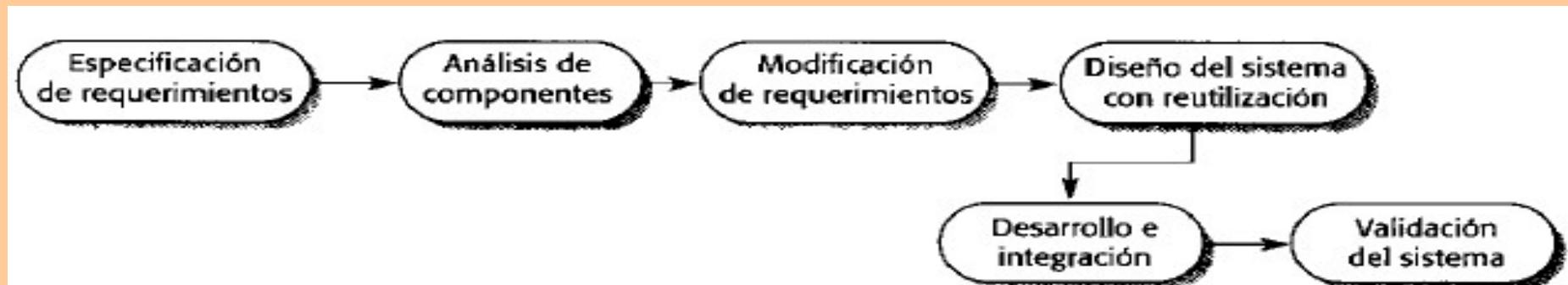
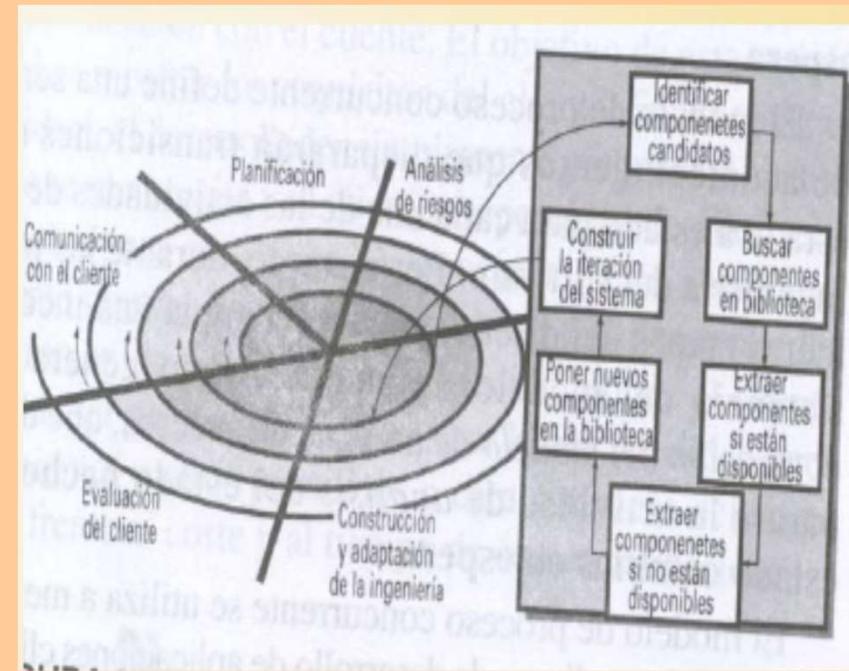
# Modelo Basado en Componentes

La reutilización según estudios:

Reduce el ciclo de vida en un 70%.

Reduce el coste del proyecto en un 84%.

Aumenta la productividad.



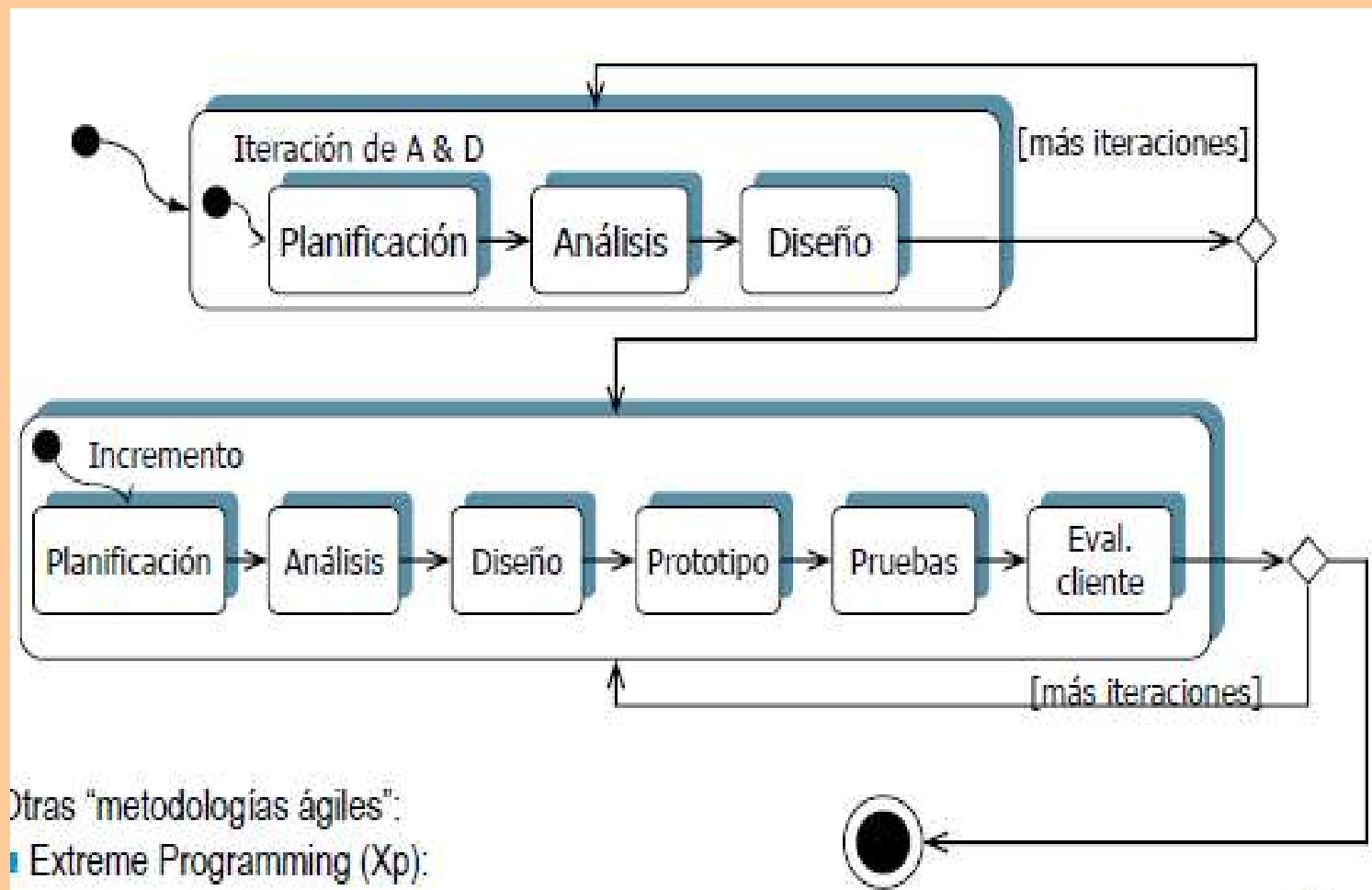
**Figura 4.3** Ingeniería del software basada en componentes.

## **PUD - Proceso Unificado de Desarrollo del Software**

El Proceso Unificado de Desarrollo del Software (PUDS) es un proceso basado en componentes que ha sido propuesto por la industria.

El proceso unificado dispone de un Lenguaje de Modelado Unificado (UML), que es utilizado construir el sistema y las interfaces que conectarán los componentes.

El PUDS combina un desarrollo incremental e iterativo, definiendo la función del sistema aplicando un enfoque basado en escenarios.



# **Proceso Unificado de Desarrollo del Software**

El PUDS se puede decir que es un proceso:

## **1.- Dirigido por los casos de uso.**

Basándose en el modelo de casos de uso y en su análisis los desarrolladores crean los modelos de diseño e implementación que llevan a cabo los casos de uso.

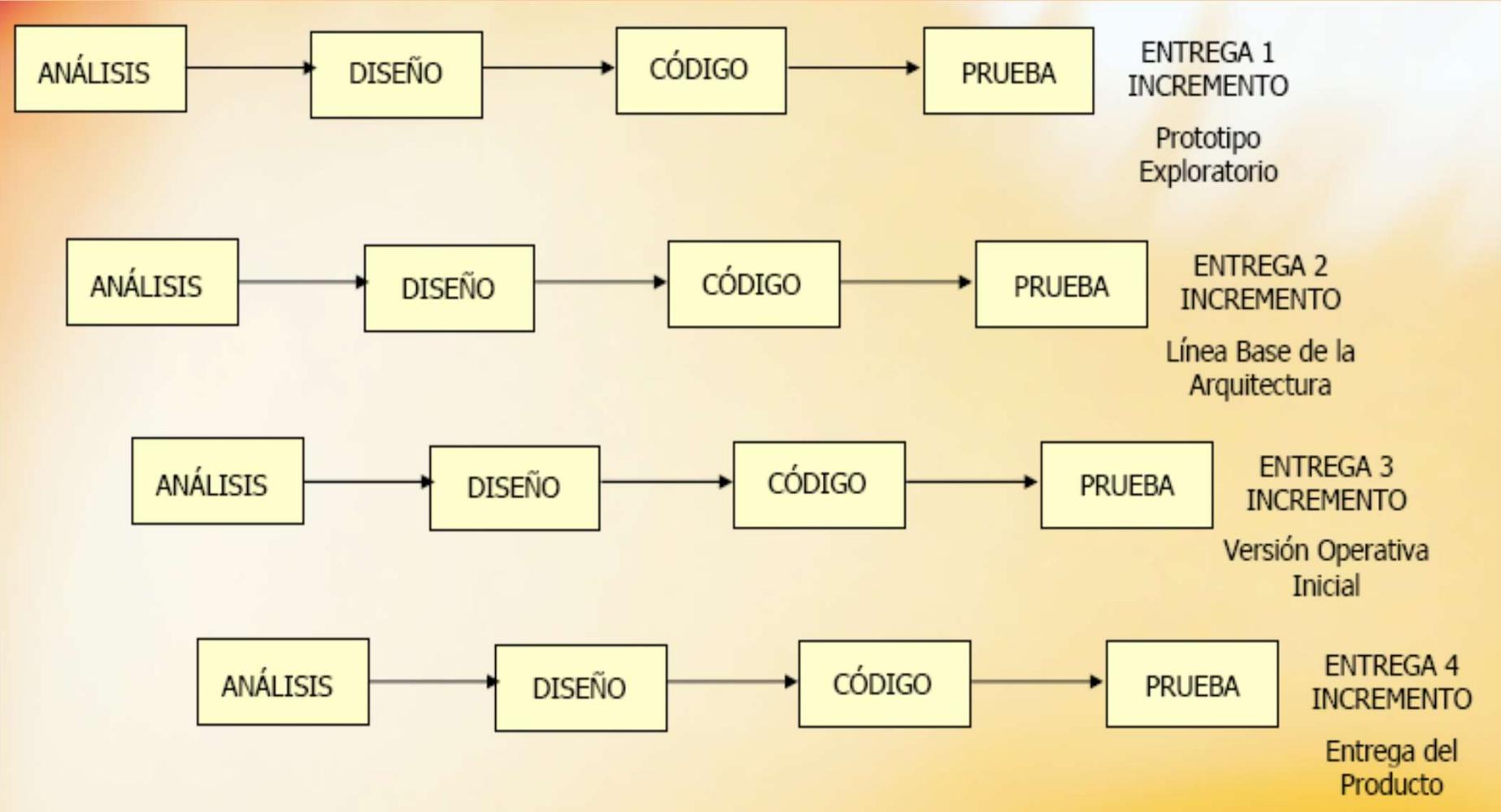
## **2.- Centrado en la arquitectura.**

La arquitectura de un sistema software se describe mediante diferentes vistas del sistema en construcción.

## **3.- Iterativo e Incremental.**

El desarrollo de un proyecto se divide en iteraciones, encargadas de pequeñas partes del trabajo y que dan como resultado un crecimiento del producto, incremento. Estas iteraciones se planifican y se prueban cada vez que terminan.

# Proceso Unificado



# Métrica V. 3

La metodología Métrica 3 es un instrumento para la sistematización de las actividades que dan soporte al ciclo de vida del software.

Contempla el desarrollo de sistemas de información para las distintas tecnologías que actualmente conviven.

En la elaboración de Métrica 3 se han tenido en cuenta los métodos de desarrollo más extendidos, así como los últimos estándares de ingeniería del software y calidad, además de referencias específicas en cuanto a seguridad y gestión de proyectos.

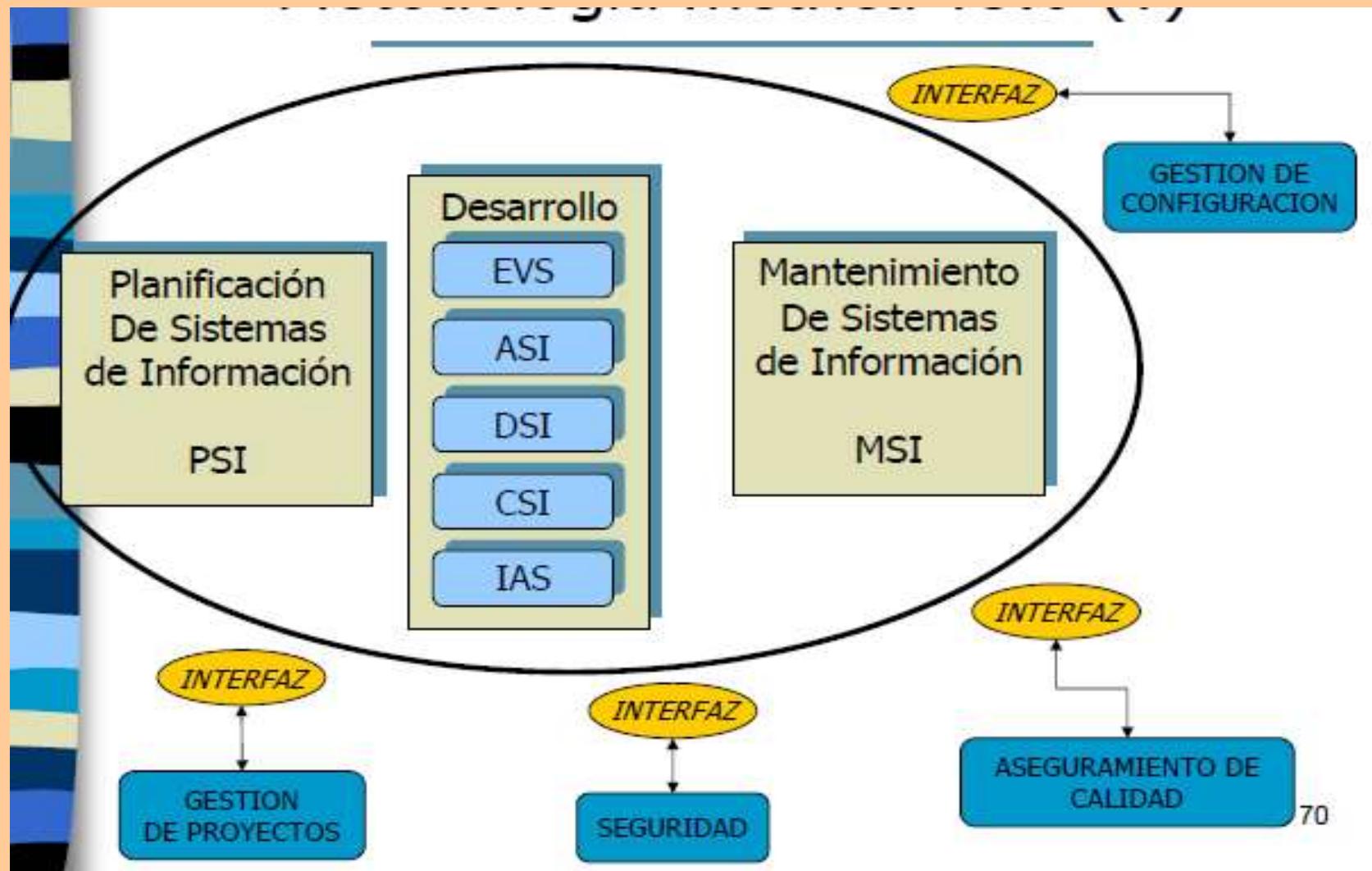
Cubre el desarrollo estructurado y orientado a objetos.

Facilita los procesos de apoyo y de organización a través de interfaces: Gestión de proyectos; Gestión de configuración; Aseguramiento de calidad y seguridad.

La automatización de sus actividades es posible ya que existe una amplia variedad de herramientas de ayuda al desarrollo.

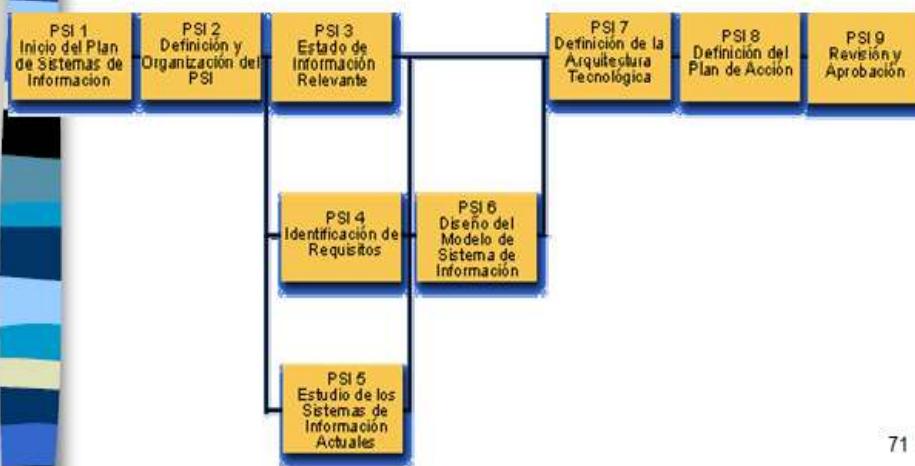
Existe una herramienta que adapta Métrica 3 a las condiciones específicas de cada proyecto, permitiendo el control y seguimiento desde diferentes perfiles.

Posee un curso de formación. [www.map.es/csi](http://www.map.es/csi)



## Planificación de Sistemas de Información (PSI)

### Esquema General de Actividades



71

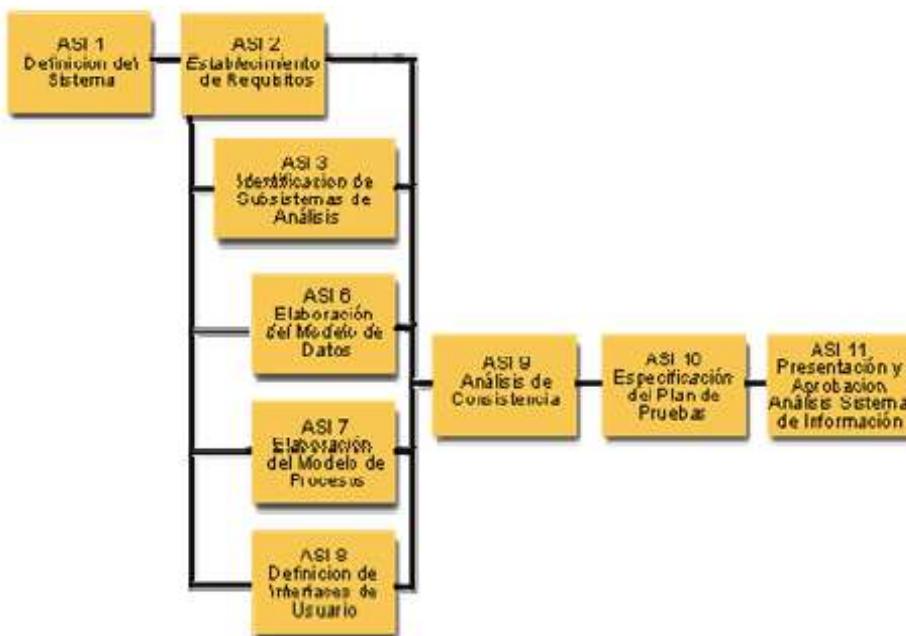
## Estudio de Viabilidad del Sistema (EVS)

### Esquema General de Actividades



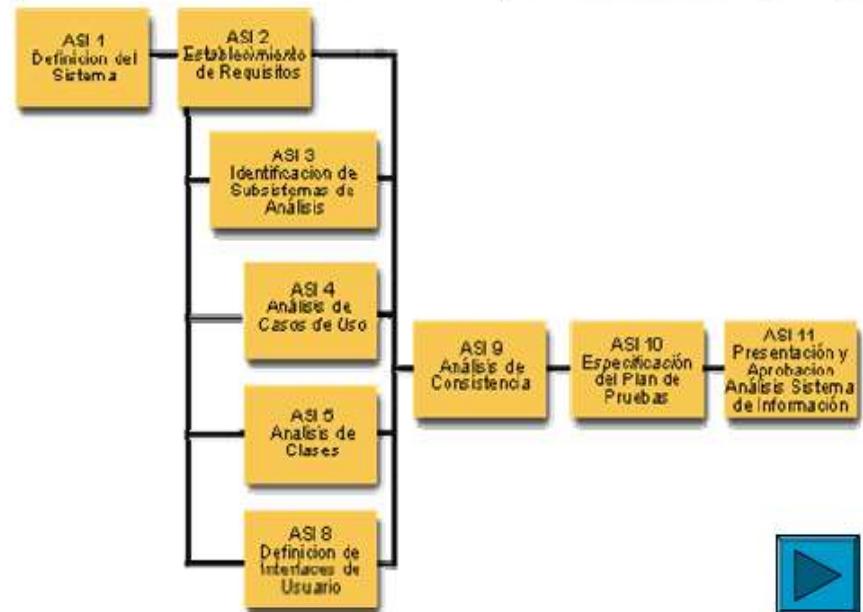
## Análisis del Sistema de Información (ASI)

### Esquema General de Actividades (Estructurado)



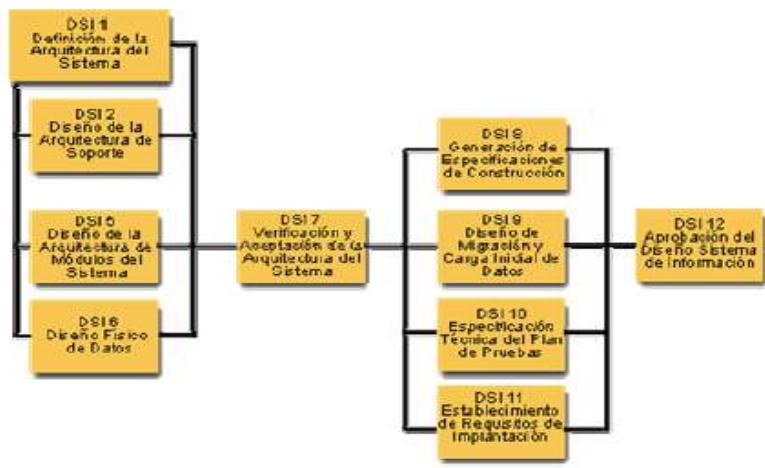
## Análisis del Sistema de Información (ASI)

### Esquema General de Actividades (Orientado a Objetos)



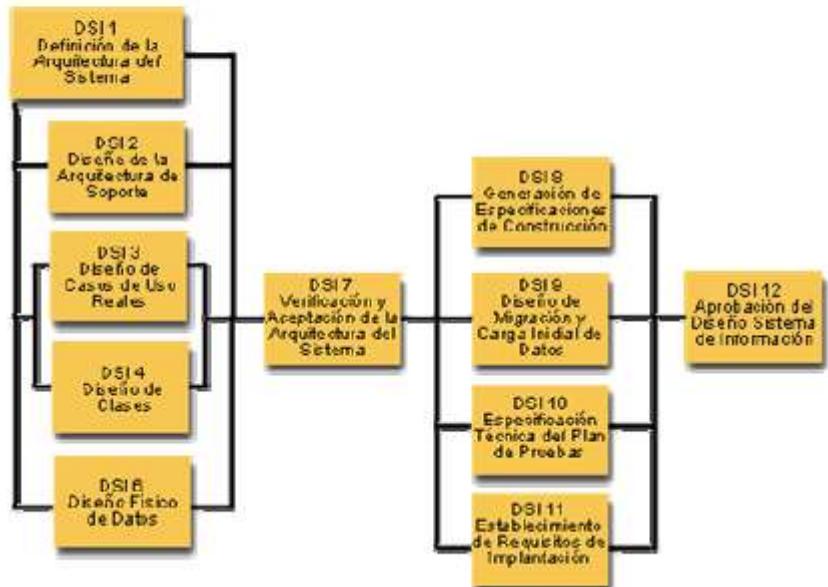
## Diseño de Sistemas de Información (DSI).

Esquema General de Actividades (Estructurado).



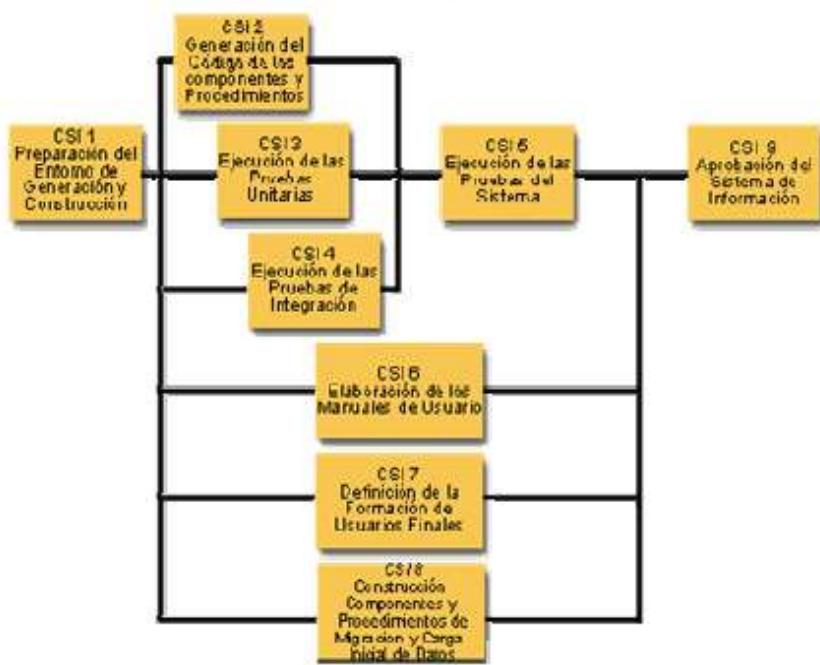
## Diseño de Sistemas de Información (DSI).

Esquema General de Actividades (Orientado a Objetos).



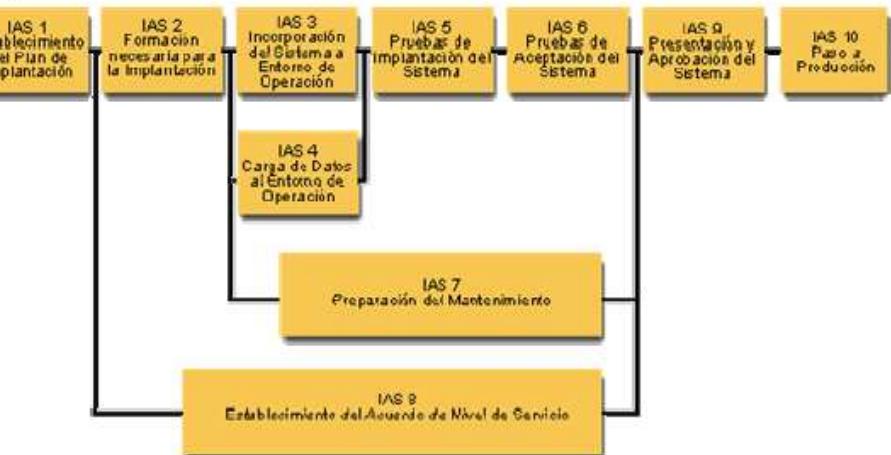
## Construcción de Sistemas de Información (CSI).

Esquema General de Actividades.



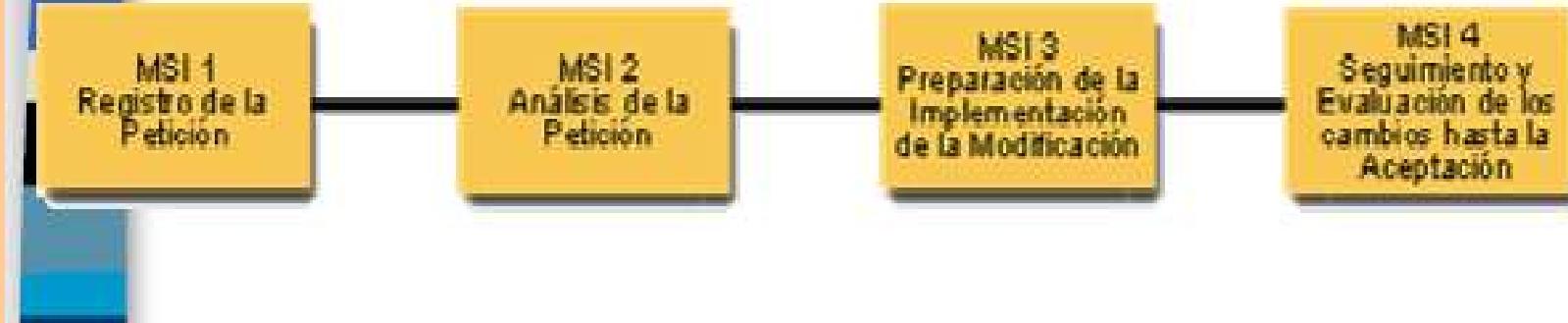
## Implantación y Aceptación del Sistema (IAS).

Esquema General de Actividades.



# *Mantenimiento de Sistemas de Información (MSI).*

## Esquema General de Actividades.



Las metodologías ágiles forman parte del movimiento de desarrollo ágil de software, que se basan en la adaptabilidad de cualquier cambio como medio para aumentar las posibilidades de éxito de un proyecto.

# XP



# ¿METODOLOGIAS AGILES?

Surge ante la necesidad de ofrecer una alternativa a la metodologías tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas.



# PRINCIPIOS DE LOS METODOS AGILES



Principio	Descripción
Participación del cliente	Los clientes deben estar fuertemente implicados en todo el proceso de desarrollo. Su papel es proporcionar y priorizar nuevos requerimientos del sistema y evaluar las iteraciones del sistema.
Entrega incremental	El software se desarrolla en incrementos, donde el cliente especifica los requerimientos a incluir en cada incremento.
Personas, no procesos	Se deben reconocer y explotar las habilidades del equipo de desarrollo. Se les debe dejar desarrollar sus propias formas de trabajar, sin procesos formales, a los miembros del equipo.
Aceptar el cambio	Se debe contar con que los requerimientos del sistema cambian, por lo que el sistema se diseña para dar cabida a estos cambios.
Mantener la simplicidad	Se deben centrar en la simplicidad tanto en el software a desarrollar como en el proceso de desarrollo. Donde sea posible, se trabaja activamente para eliminar la complejidad del sistema.

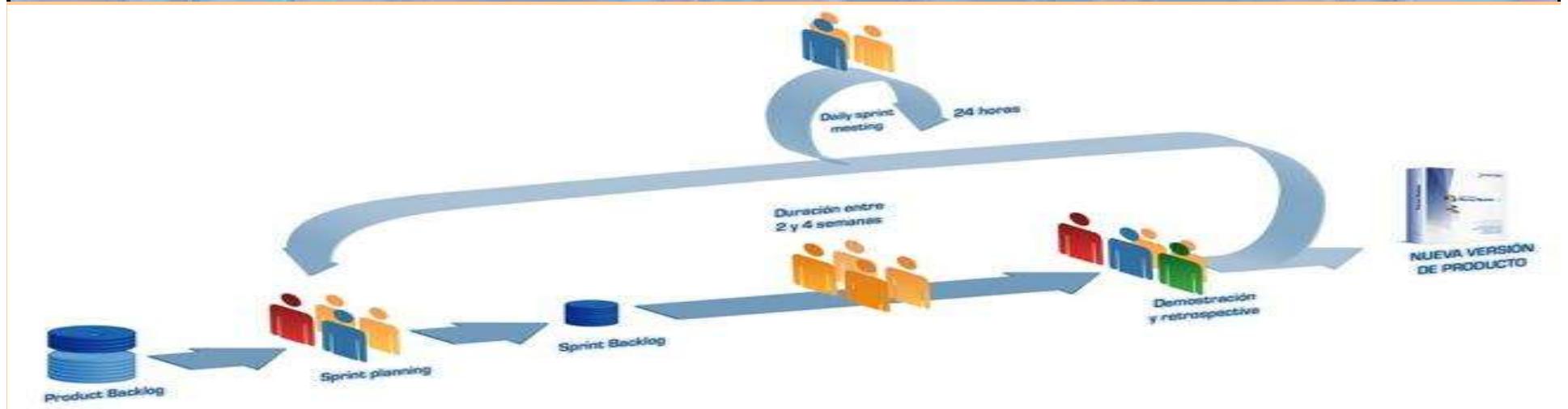
# DIFERENCIAS ENTRE METODOLOGIAS AGILES Y NO AGILES



Metodologías Ágiles	Metodologías Tradicionales
Basadas en heurísticas provenientes de prácticas de producción de código	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo
Especialmente preparados para cambios durante el proyecto	Cierta resistencia a los cambios
Impuestas internamente (por el equipo)	Impuestas externamente
Proceso menos controlado, con pocos principios	Proceso mucho más controlado, con numerosas políticas/normas
No existe contrato tradicional o al menos es bastante flexible	Existe un contrato prefijado
El cliente es parte del equipo de desarrollo	El cliente interactúa con el equipo de desarrollo mediante reuniones
Grupos pequeños (<10 integrantes) y trabajando en el mismo sitio	Grupos grandes y posiblemente distribuidos
Pocos artefactos	Más artefactos
Pocos roles	Más roles
Menos énfasis en la arquitectura del software	La arquitectura del software es esencial y se expresa mediante modelos

Tabla 1. Diferencias entre metodologías ágiles y no ágiles

# Modelos Ágiles: Scrum



Es una metodología ágil y flexible para gestionar el desarrollo de software, cuyo principal objetivo es maximizar el retorno de la inversión para su empresa (ROI). Se basa en construir primero la funcionalidad de mayor valor para el cliente y en los principios de inspección continua, adaptación, auto-gestión e innovación.



FLOWERS IN SPACE  
flowerinspace.com

## Breve introducción a Scrum

¿Que significa Scrum?

*Scrum significa melé*



Melé es un tipo de jugada del deporte Rugby.

Todos los jugadores de ambos equipo se agrupan en una formación en la cual lucharán por obtener el balón que se introduce por el centro.



## Breve introducción a Scrum

La complejidad de una melé hace que:

**Si un miembro del equipo se viene abajo, se cae toda la melé.**

En consecuencia, los jugadores deben estar bien coordinados, apoyarse en sus compañeros para empujar al mismo tiempo y con ello, avanzar a la misma velocidad.



# Historia

- Este modelo fue identificado y definido por Ikujiro Nonaka e Hirotaka Takeuchi a principios de los 80, al analizar cómo desarrollaban los nuevos productos las principales empresas de manufactura tecnológica: Fuji-Xerox, Canon, Honda, NEC, Epson, Brother, 3M y Hewlett-Packard (Nonaka & Takeuchi, *The New New Product Development Game*, 1986).
- En su estudio, Nonaka y Takeuchi compararon la nueva forma de trabajo en equipo, con el avance en formación de melé (scrum en inglés) de los jugadores de Rugby, a raíz de lo cual quedó acuñado el término “scrum” para referirse a ella.
- Aunque esta forma de trabajo surgió en empresas de productos tecnológicos, es apropiada para cualquier tipo de proyecto con requisitos inestables y para los que requieren rapidez y flexibilidad, situaciones frecuentes en el desarrollo de determinados sistemas de software.
- En 1995, Ken Schwaber presentó “Scrum Development Process” en OOPSLA 95 (Object-Oriented Programming Systems & Applications conference)(SCRUM Development Process), un marco de reglas para desarrollo de software, basado en los principios de Scrum, y que él había empleado en el desarrollo de Delphi, y Jeff Sutherland en su empresa Easel Corporation (compañía que en los macrojuegos de compras y fusiones, se integraría en VMARK, y luego en Informix y finalmente en Ascential Software Corporation).



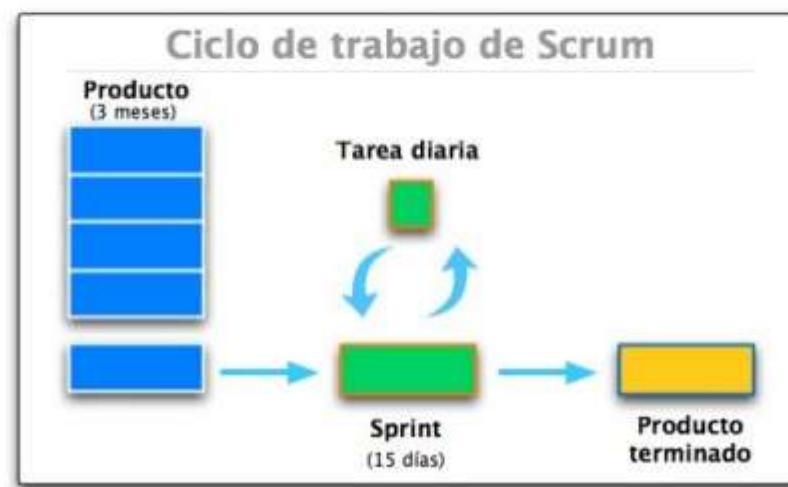


## Breve introducción a Scrum



¿Qué es Scrum?

Es un marco de trabajo ágil que se basa en la iteración y entrega incrementales de desarrollo de un producto o servicio.





## Breve introducción a Scrum



### Metodologías ágiles

Sus premisas principales son valorar:

- A los individuos y su interacción por encima de los procesos y las herramientas.
- Al software que funciona, por encima de la documentación exhaustiva.
- A la colaboración con el cliente, por encima de la negociación contractual.
- A la respuesta al cambio por encima del seguimiento de un plan.



## Breve introducción a Scrum



### Ciclo de trabajo Scrum

- 1º- Toma de requisitos al cliente. Para cada requisito principal se crea un bloque de trabajo, llamado **historia**
- 2º- El cliente ordena los bloques de trabajo en una **pila de producto** según su prioridad de entrega.
- 3º- El equipo de trabajo toma un grupo de historias, con el que trabajan durante una iteración o **sprint**.
- 4º- Una vez finalizado un sprint entregan al cliente el resultado del trabajo. Se vuelve al punto 2º hasta terminar la pila de producto.



## Breve introducción a Scrum



### Roles

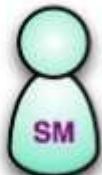
Cada persona que interviene en el proceso de creación de un producto o servicio tiene un rol específico en Scrum. En el ejemplo práctico veremos el papel que desarrolla cada uno y sus funciones.



Cliente



Dueño de Producto



Scrum Master



Equipo

## Beneficios

- **Cumplimiento de expectativas:** El cliente establece sus expectativas indicando el valor que le aporta cada requisito /**historia** del proyecto, el equipo los estima y con esta información el **Product Owner** establece su prioridad. De manera regular, en las demos de Sprint el **Product Owner** comprueba que efectivamente los requisitos se han cumplido y transmite feedback al equipo.
- **Flexibilidad a cambios:** Alta capacidad de reacción ante los cambios de requerimientos generados por necesidades del cliente o evoluciones del mercado. La metodología está diseñada para adaptarse a los cambios de requerimientos que conllevan los proyectos complejos.
- **Reducción del Time to Market:** El cliente puede empezar a utilizar las funcionalidades más importantes del proyecto antes de que esté finalizado por completo.
- **Mayor calidad del software:** La metódica de trabajo y la necesidad de obtener una versión funcional después de cada iteración, ayuda a la obtención de un software de calidad superior.
- **Mayor productividad:** Se consigue entre otras razones, gracias a la eliminación de la burocracia y a la motivación del equipo que proporciona el hecho de que sean autónomos para organizarse.
- **Maximiza el retorno de la inversión (ROI):** Producción de software únicamente con las prestaciones que aportan mayor valor de negocio gracias a la priorización por retorno de inversión.
- **Predicciones de tiempos:** Mediante esta metodología se conoce la velocidad media del equipo por sprint (los llamados puntos historia), con lo que consecuentemente, es posible estimar fácilmente para cuando se dispondrá de una determinada funcionalidad que todavía está en el Backlog.
- **Reducción de riesgos:** El hecho de llevar a cabo las funcionalidades de más valor en primer lugar y de conocer la velocidad con que el equipo avanza en el proyecto, permite despejar riesgos eficazmente de manera anticipada.

# Programación Extrema



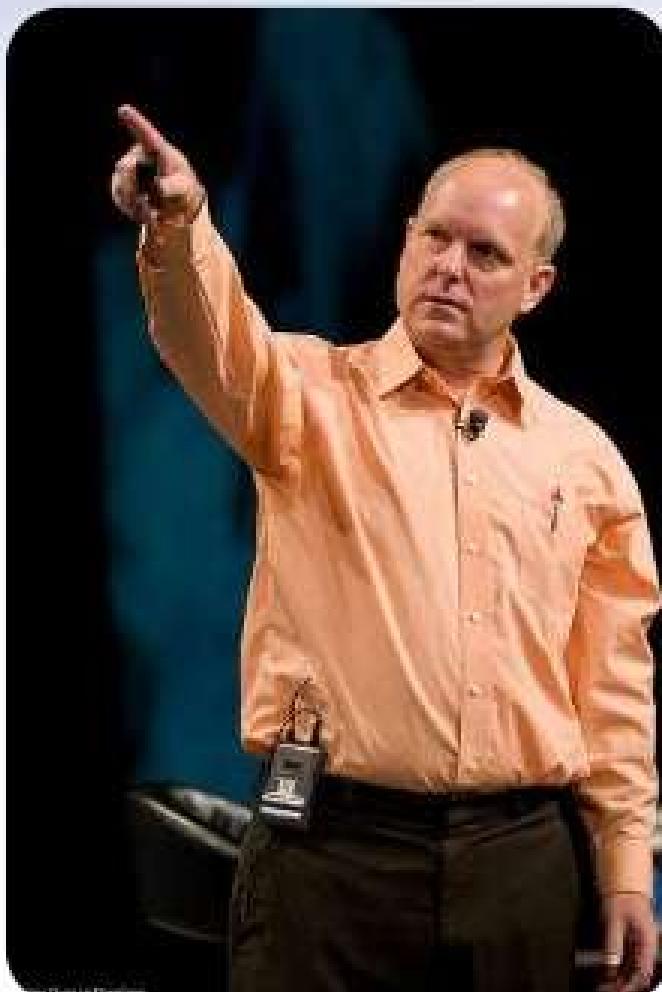
# ¿Programación Extrema o XP?

- La Programación Extrema es una metodología ligera de desarrollo de software que se basa en la simplicidad, la comunicación y la realimentación o reutilización del código desarrollado.
- Centrada en potenciar las relaciones interpersonales como clave para el éxito en el desarrollo de software.
- Su objetivo es aumentar la productividad al desarrollar software.



# ORIGEN DE LA METODOLOGIA XP

- Nace de la mano de Kent Beck en el verano de 1996, cuando trabajaba para Chrysler Corporation.
- El tenía varias ideas de metodologías para la realización de programas que eran cruciales para el buen desarrollo de cualquier sistema.
- Las ideas primordiales de su sistema las comunicó en la revista C++ Magazine en una entrevista que ésta le hizo el año 1999.



# OBJETIVOS DE XP

- La satisfacción del cliente.
- Potenciar el trabajo en grupo.
- Minimizar el riesgo actuando sobre las variables del proyecto:
  - Coste
  - Tiempo
  - Calidad
  - Alcance

© FRANCISCO J. GARCÍA

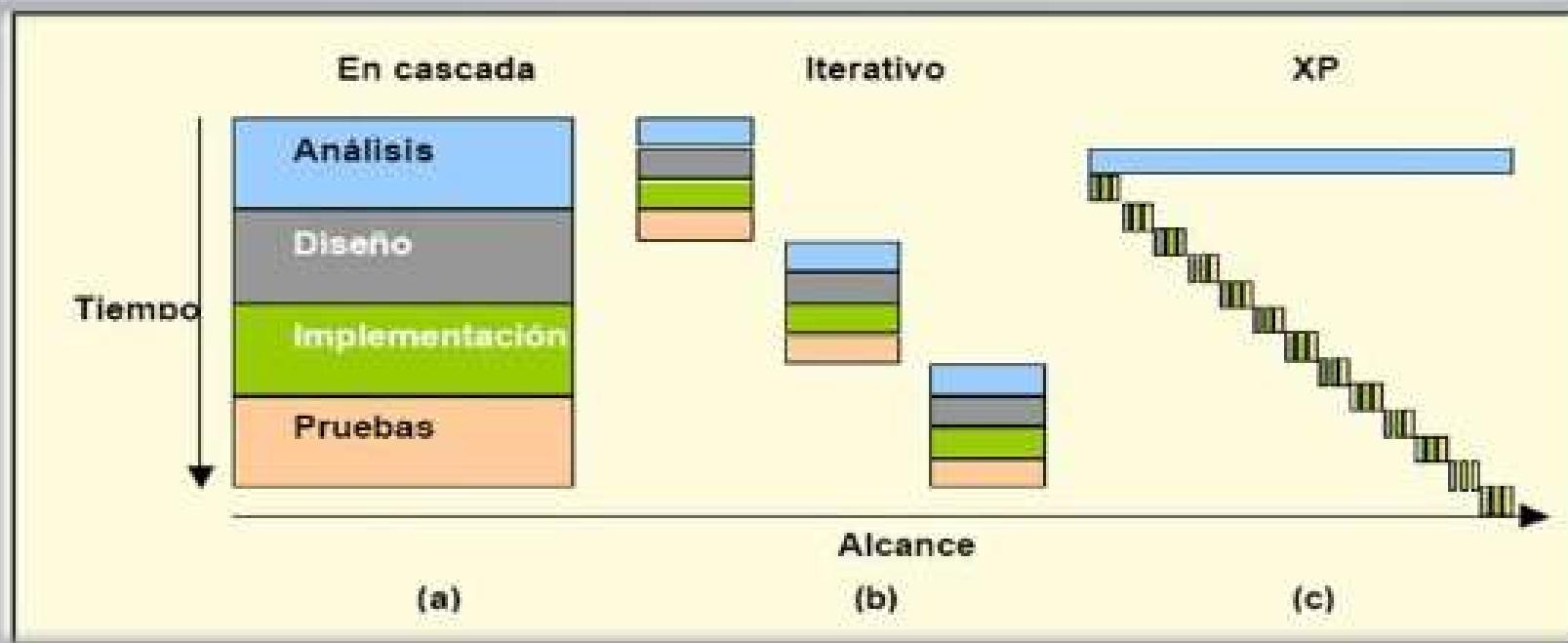


# VALORES QUE INSPIRAN XP

- Comunicación
- Sencillez
- Retroalimentación
- Valentía



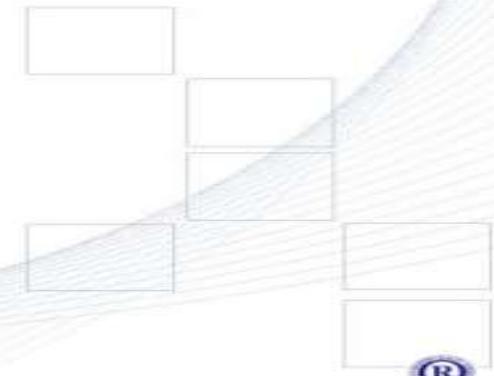
# EL COSTE DEL CAMBIO



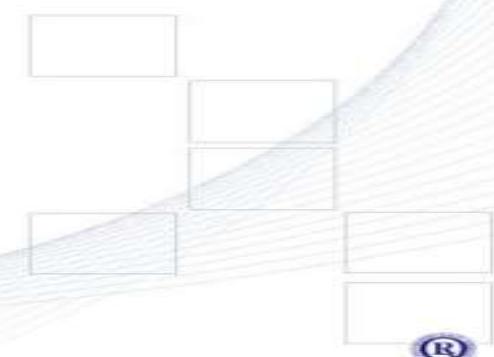
Evolución de los largos ciclos de desarrollo en cascada (a) a ciclos iterativos más cortos (b) y a la mezcla que hace XP

Las Prácticas son:

1. El juego de la planificación
2. Pequeñas entregas
3. Metáfora
4. Diseño simple
5. Pruebas
6. Refactorización



7. Programación por parejas
8. Propiedad colectiva
9. Integración continua
10. 40 horas semanales
11. Cliente en casa
12. Estándares de codificación.

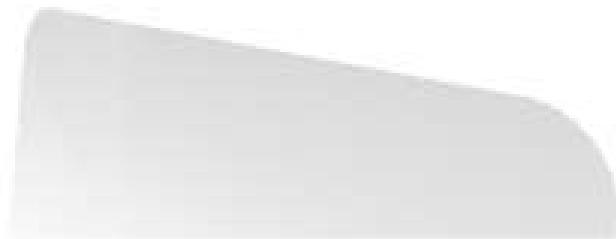


Son:

2. La satisfacción del cliente.

3. Potenciar el trabajo en grupo, todos están  
involucrados en el desarrollo del software.

# **ROLES DE XP**



# 1. PROGRAMADOR



- Pieza básica en desarrollos XP
- Más responsabilidad que en otros modos de desarrollo
- Responsable sobre el código
- Responsable sobre el diseño (refactorización, simplicidad)
- Responsable sobre la integridad del sistema (pruebas)
- Capacidad de comunicación
- Acepta críticas (código colectivo)



## 2. CLIENTE



- Pieza básica en desarrollos XP
- Define especificaciones
- Influye sin controlar
- Confía en el grupo de desarrollo
- Define pruebas funcionales



### 3. Encargado de pruebas (Tester)



- Apoya al cliente en la preparación/realización de las pruebas funcionales
- Ejecuta las pruebas funcionales y publica los resultados



## 4. Encargado de seguimiento (Tracker)



- Recoge, analiza y publica información sobre la marcha del proyecto sin afectar demasiado el proceso
- Supervisa el cumplimiento de la estimaciones en cada iteración
- Informa sobre la marcha de la iteración en curso
- Controla la marcha de las pruebas funcionales, de los errores reportados, de las responsabilidades aceptadas y de las prueba añadidas por los errores encontrados

## 5. Entrenador (Coach)



- Experto en XP
- Responsable del proceso en su conjunto
- Identifica las desviaciones y reclama atención sobre las mismas
- Guía al grupo de forma indirecta (sin dañar su seguridad ni confianza)
- Interviene directamente si es necesario
- Atajar rápidamente el problema



## 6. Gestor (Big boss)



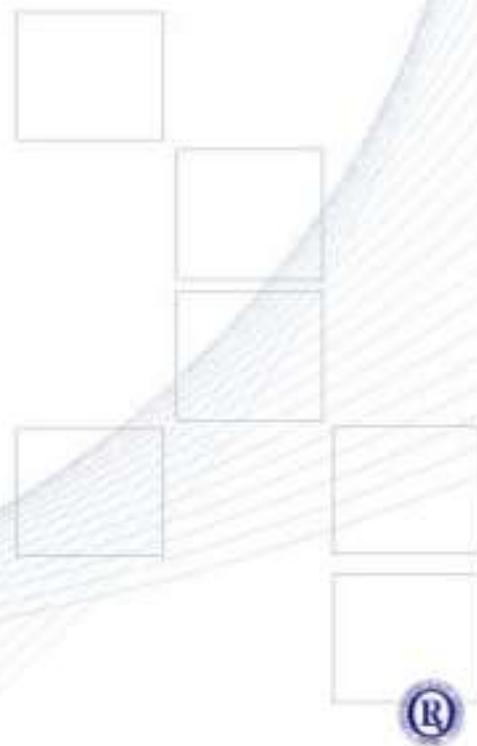
- Favorece la relación entre usuarios y desarrolladores
- Confía en el equipo XP
- Cubre las necesidades del equipo XP
- Asegura que alcanza sus objetivos



# Artefactos esenciales en XP



- + Historias del Usuario
- + Tareas de Ingeniería
- + Pruebas de Aceptación
  
- + Pruebas Unitarias y de Integración
- + Plan de la Entrega
- + Código



# Historia de Usuario



## Historia de Usuario

Número: 1 Nombre: Enviar artículo

Usuario: Autor

Modificación de Historia Número:

Iteración Asignada: 2

Prioridad en Negocio: Alta  
(Alta / Media / Baja)

Puntos Estimados:

Riesgo en Desarrollo:  
(Alto / Medio / Bajo)

Puntos Reales:

Descripción:

Se introducen los datos del artículo (título, fichero adjunto, resumen, tópicos) y de los autores (nombre, e-mail, afiliación). Uno de los autores debe indicarse como autor de contacto. El sistema confirma la correcta recepción del artículo enviando un e-mail al autor de contacto con un userid y password para que el autor pueda posteriormente acceder al artículo.

Observaciones:



# Tarea de Ingeniería



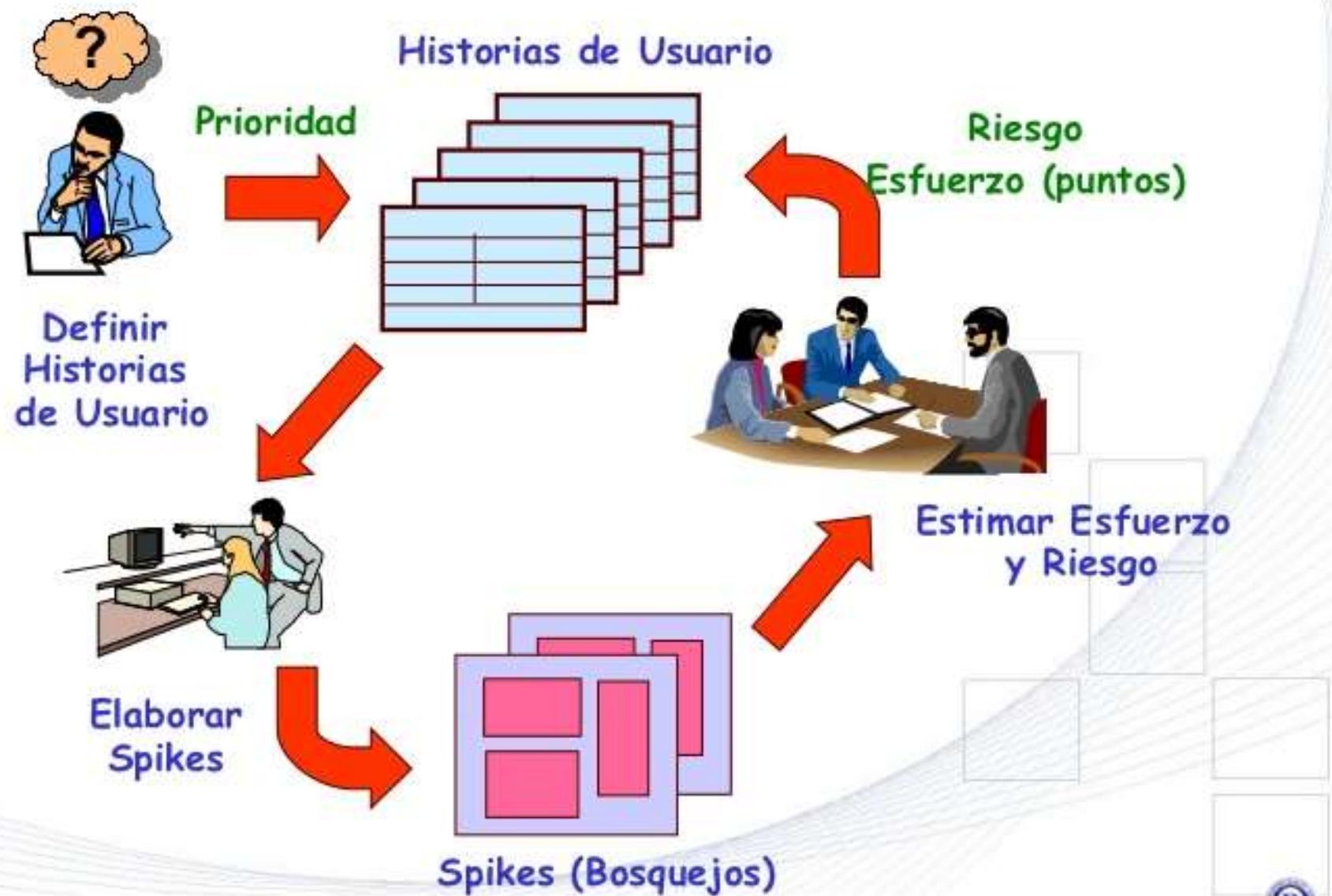
Tarea	
Número tarea:	Número historia:
Nombre tarea:	
Type de tarea: Desarrollo / Corrección / Mejora / Otra	Puntos estimados: <input type="text"/> <input type="text"/>
Fecha inicio:	Fecha fin: <input type="text"/> <input type="text"/>
Programador responsable: <input type="text"/>	
Descripción: <input type="text"/> <input type="text"/> <input type="text"/>	

# Prueba de Aceptación

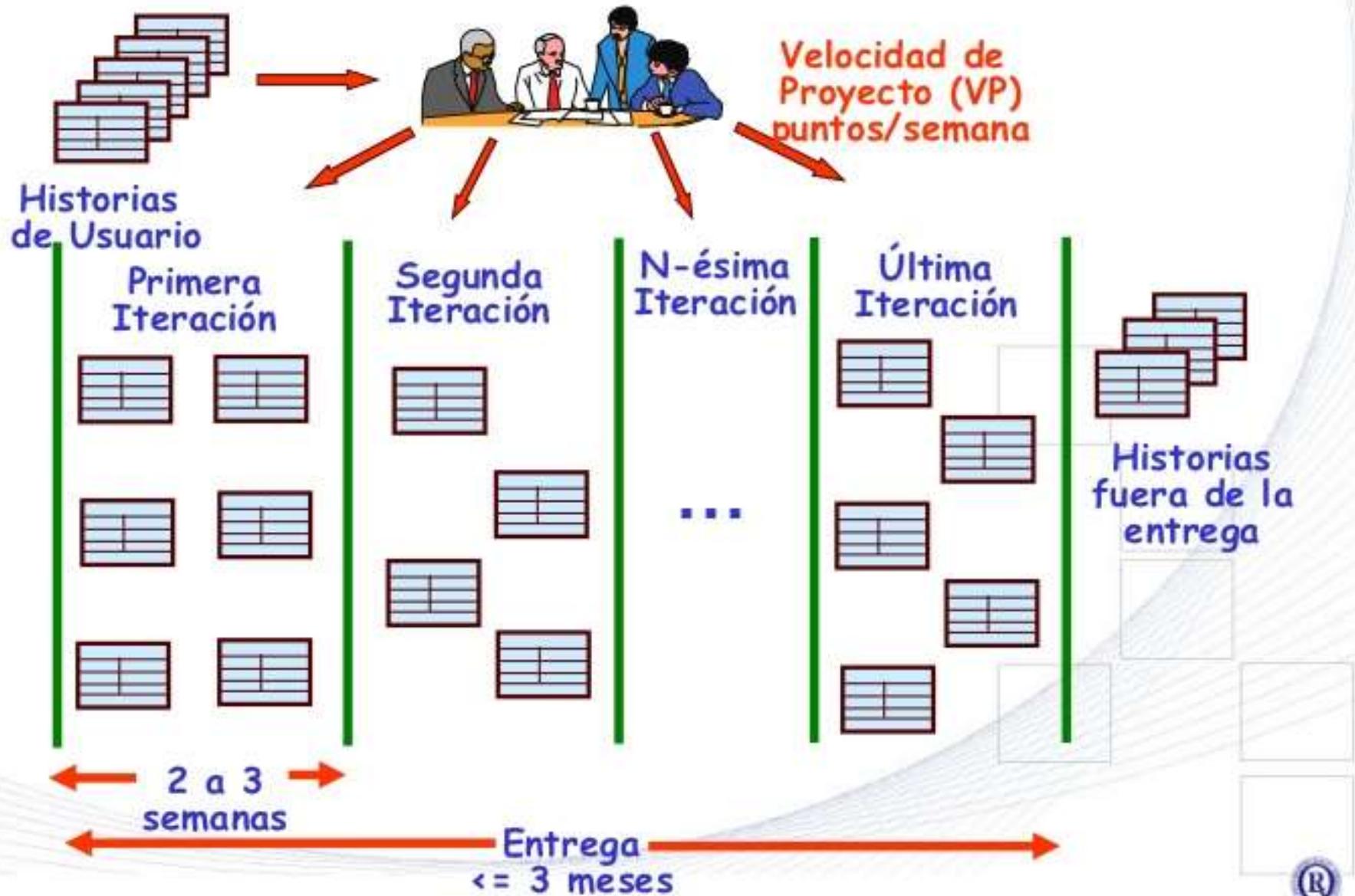


Caso de Prueba	
Número Caso de Prueba:	Número Historia de Usuario:
Nombre Caso de Prueba:	
Descripción:	
Condiciones de ejecución:	
Entradas:	
Resultado esperado:	
Evaluación:	

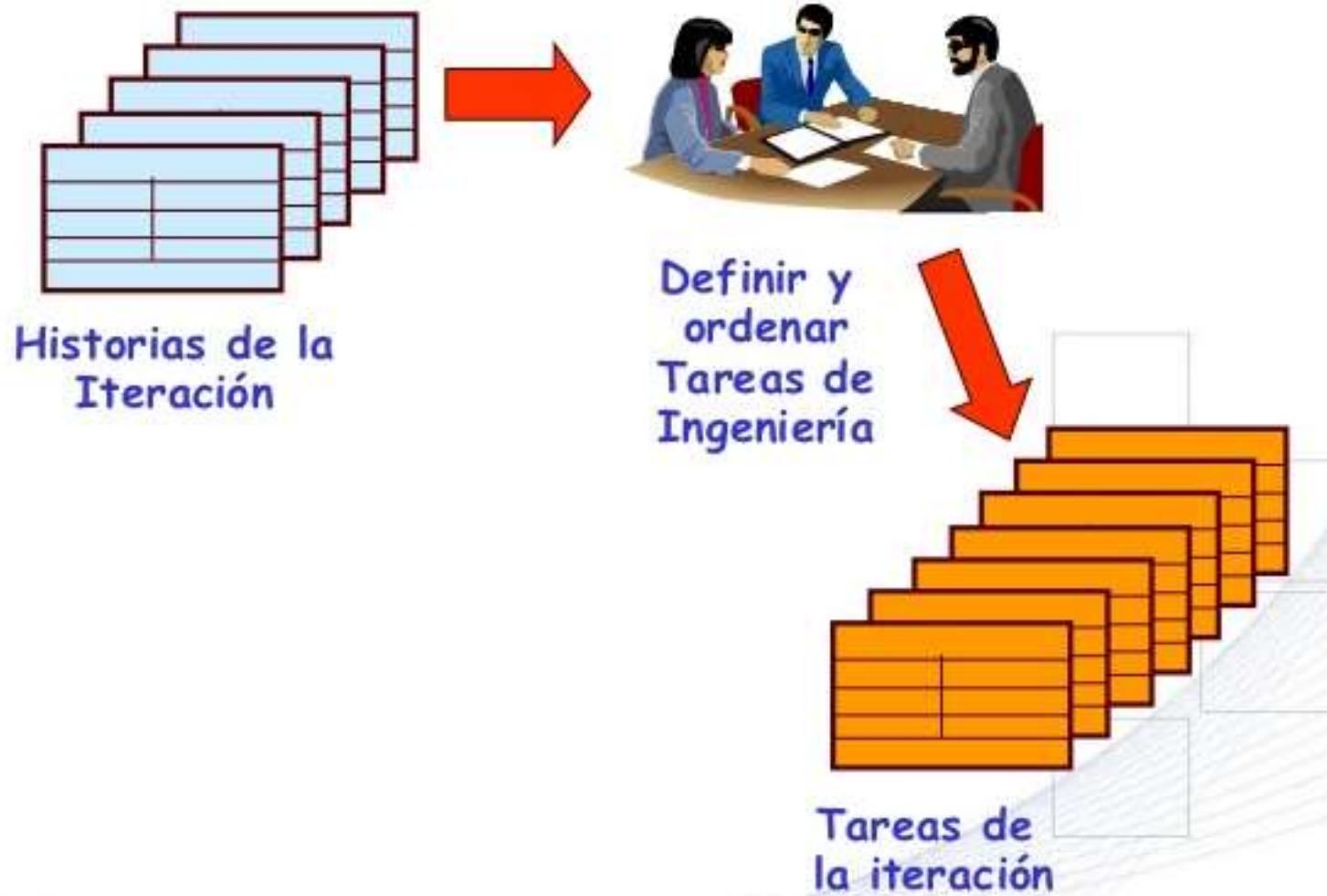
# Escenarios en XP : Exploración



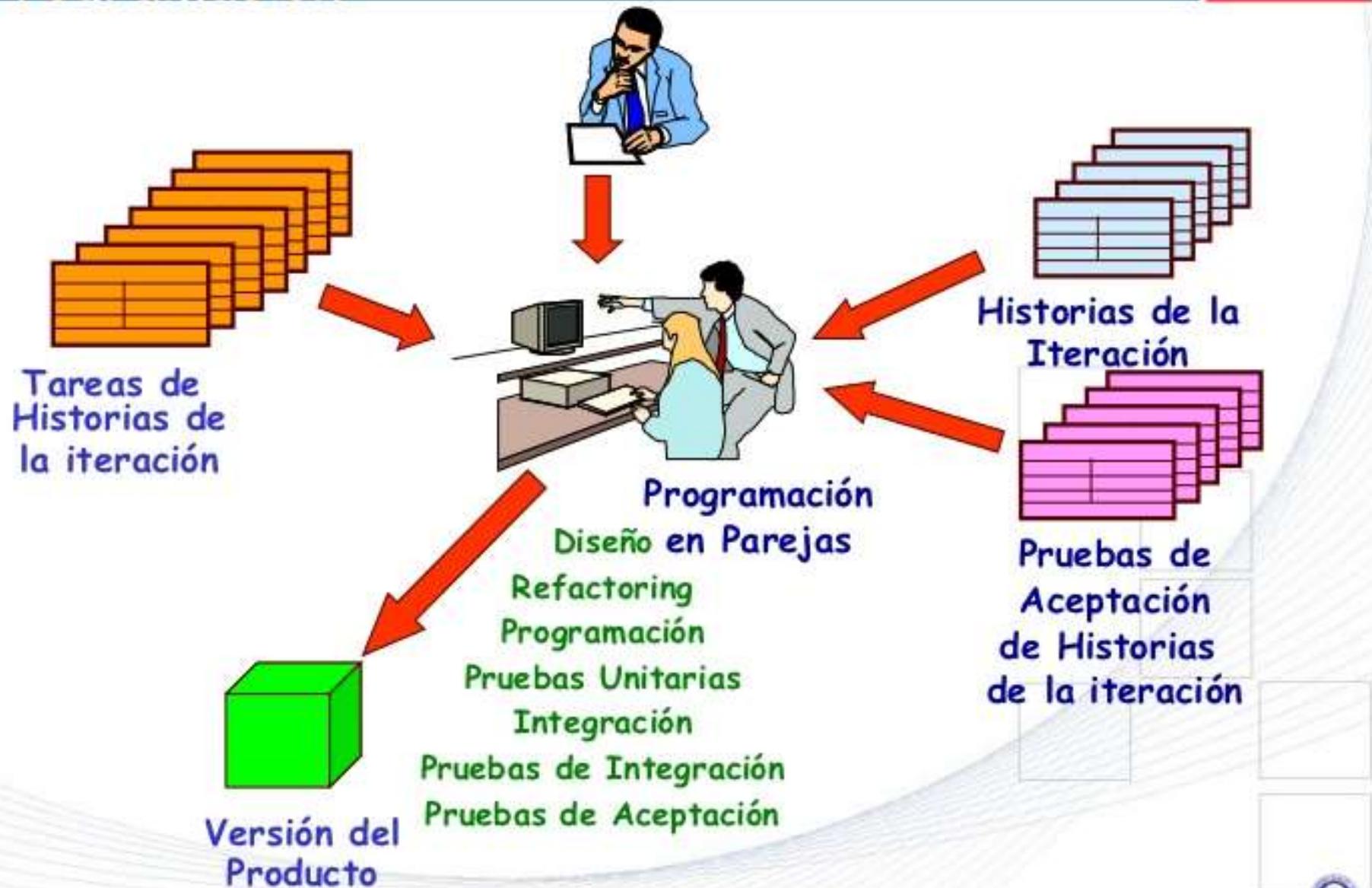
# Escenarios en XP: Planificación de la Entrega



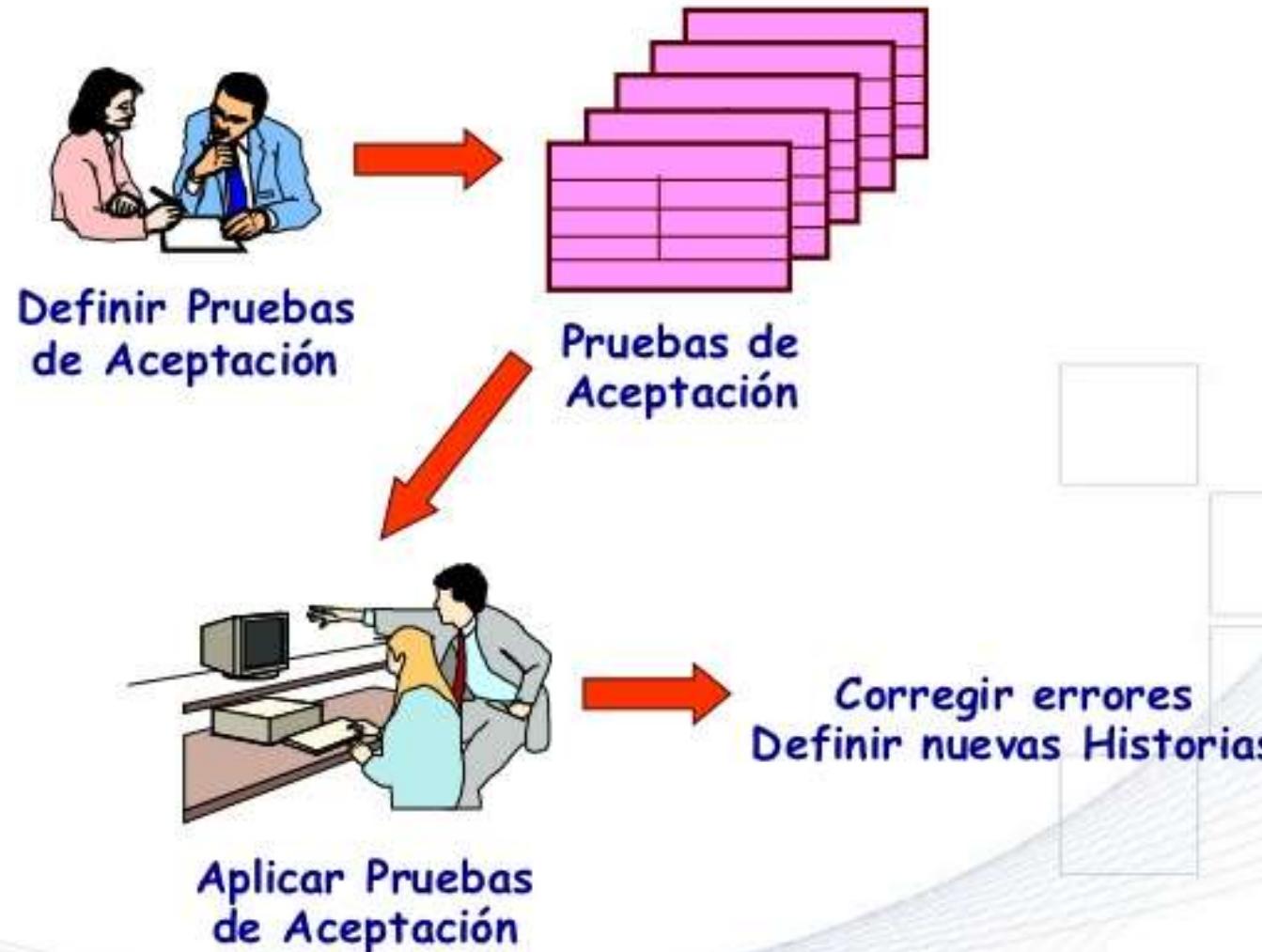
# Escenarios en XP : Comenzar Iteración



# Escenarios en XP : Programación



# Escenarios en XP : Pruebas de Aceptación



## Entorno y clima de trabajo Espacio de trabajo XP



- + Espacio abierto
- + Mesas centrales
- + Cubículos en el espacio exterior



Espacio de trabajo del proyecto C3 de DaimlerChrysler

## ... Entorno y clima de trabajo Reunión diaria XP

- + Reunión diaria: "Stand-up Meeting"

+ Todo el equipo

+ Problemas

+ Soluciones



+ De pie en un círculo

+ Evitar discusiones largas

+ Sin conversaciones separadas

®

## Entorno y clima de trabajo Gantt de Pared



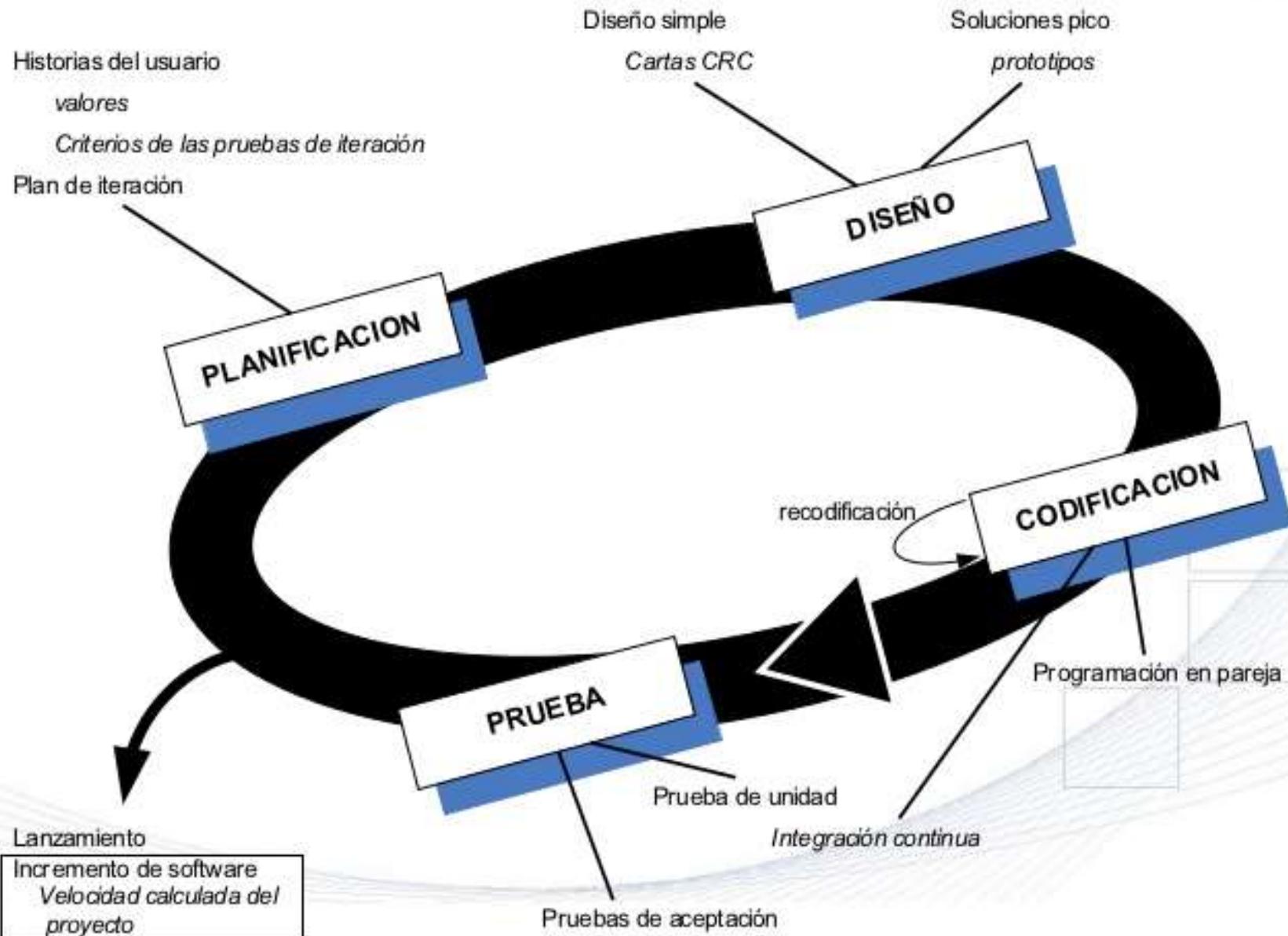
"Centro del universo del proyecto"

"Punto de reunión para la "Stand-up Meeting"

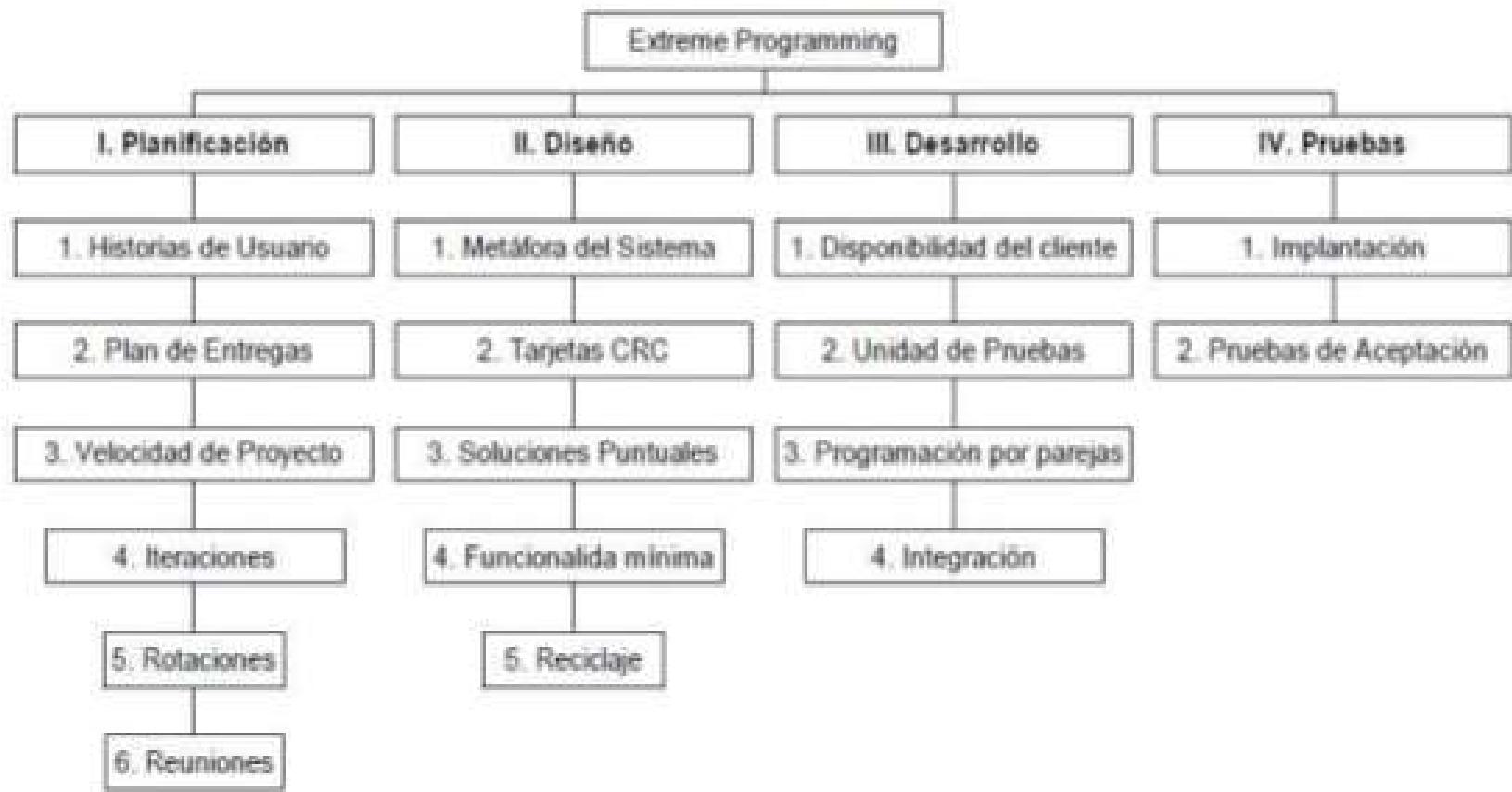
Obtenida de [www.agiletak.com](http://www.agiletak.com)

®

# Como hacemos funcionar la Metodología XP



# FASES DE LA METODOLOGIA XP



XP plantea la planificación como un permanente dialogo entre las partes la empresarial (deseable) y la técnica (possible).



deseable



possible

## .1 El Juego de la Planificación

### ▪ Negocio

- Ámbito *¿Qué debe resolver el software?*
- Prioridad *¿Qué debe ser echo en primer lugar?*
- Composición de versiones *¿Cuánto es necesario hacer para aportar valor?*
- Fechas de versiones *¿Fechas para presencia del software?*

## .1 El Juego de la Planificación

### ▪ Técnico

- Estimaciones *¿Cuánto lleva implementar una característica?*
- Consecuencias, informar sobre consecuencias de las decisiones que adopta el negocio.
- Procesos *¿Cómo se organiza el trabajo en el equipo?*
- Programación detallada: En una versión *¿Qué se resolverá primero?*

### .2 Pequeñas versiones.

Cada versión debe de ser tan pequeña como fuera posible, conteniendo los requisitos de negocios más importantes, las versiones tiene que tener sentido como un todo.

### .3 Metáfora.

Es una historia que todo el mundo puede contar a cerca de cómo funciona el sistema.

### .5 Recodificación.

Este proceso se le denomina recodificar o refactorizar (refactoring).y consiste en hacer el programa mas simple sin perder funcionalidad.

No debemos de recodificar ante especulaciones si no solo cuándo el sistema te lo pida.

### .4 Diseño simple.

El diseño adecuado para el software es aquel que:

- Funciona con todas las pruebas.
- No tiene lógica duplicada.
- Manifiesta cada intención importante para los programadores
- Tiene el menor número de clases y métodos.

### .6 Programación por parejas.

Todo el código de producción se escribe con dos personas mirando a una máquina, con un solo teclado y un solo ratón.

Cada miembro de la pareja juega su papel: uno codifica en el ordenador y piensa la mejor manera de hacerlo, el otro piensa mas estratégicamente, ¿Va a funcionar?, ¿Puede haber pruebas donde no funcione?, ¿Hay forma de simplificar el sistema global para que el problema desaparezca?.

#### .7 Propiedad Colectiva.

Cualquiera que crea que puede aportar valor al código en cualquier parcela puede hacerlo, ningún miembro del equipo es propietario del código.

#### .8 Integración continua.

El código se debe integrar como mínimo una vez al día, y realizar las pruebas sobre la totalidad del sistema.

#### .11 Estándares de Codificación.

Se debe establecer un estándar de codificación aceptado e implantado por todo el equipo.

#### .9 Cuarenta horas.

Si queremos estar frescos y motivados cada mañana y cansado y satisfecho cada noche del sistema.

#### .10 Cliente In Situ.

Un cliente real debe sentarse con el equipo de programadores, estar disponible para responder a sus preguntas, resolver discusiones y fijar las prioridades.

#### .11 Estándares de Codificación.

Se debe establecer un estándar de codificación aceptado e implantado por todo el equipo.

#### .12 Hacer pruebas.

Toda característica en el programa debe ser probada, los programadores escriben pruebas para chequear el correcto funcionamiento del programa, los clientes realizan pruebas funcionales. El resultado un programa mas seguro que soporte cambios en el tiempo.

# Prácticas XP



## PLANIFICACION

## DISEÑO

## CODIFICACION

## PRUEBAS

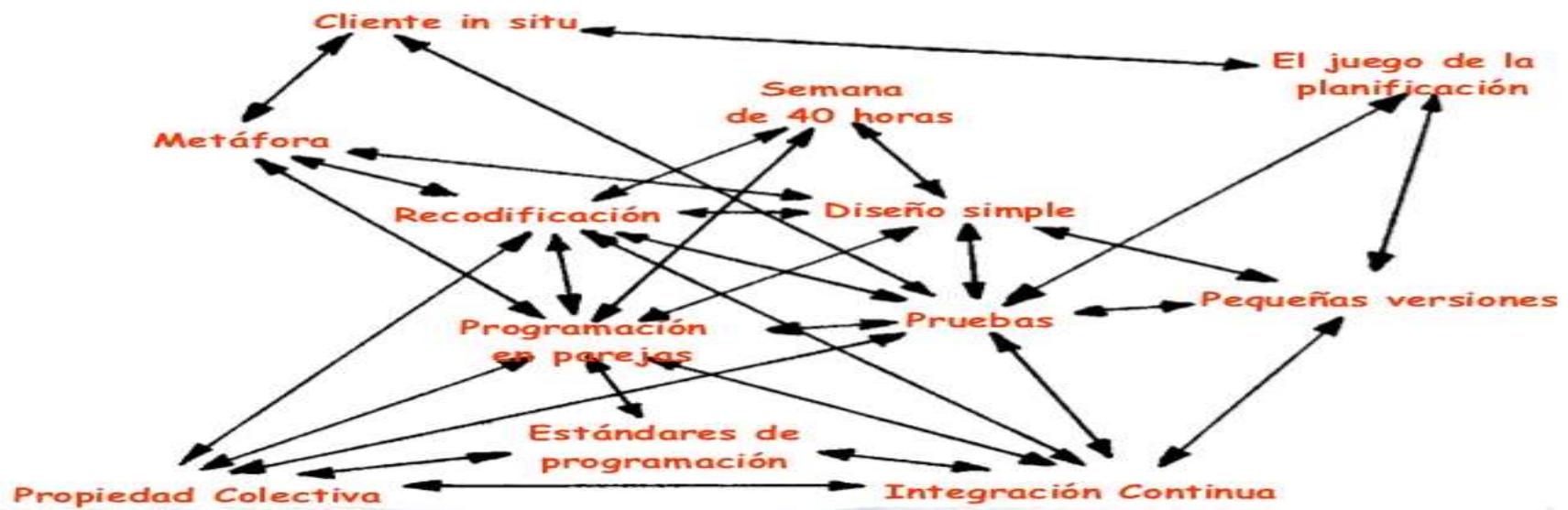
- 1. El juego de la planificación
- 2. Entregas pequeñas
- 3. Metáfora
- 4. Diseño simple
- 5. Recodificación
- 6. Programación en parejas
- 7. Propiedad colectiva
- 8. Integración continua
- 9. Semana de 40 horas
- 10. Cliente in situ
- 11. Estándares de programación



®



## ... Prácticas XP Interacción entre Prácticas



XP: Kent Beck

®

# VENTAJAS Y DESVENTAJAS DE XP



## Ventajas:

- Programación organizada.
- Menor taza de errores.
- Satisfacción del programador.

## Desventajas:

- Es recomendable emplearlo solo en proyectos a corto plazo.
- Altas comisiones en caso de fallar.

## CONCLUSIONES



- + La programación extrema es una forma ligera, eficiente, flexible, predecible, científica y divertida de generar software.
- + La programación extrema se beneficia de la existencia de un gran número de herramientas de software libre que permiten aplicarla con gran productividad.
- + El software libre se inspira en algunas de las prácticas de la XP .

## CONCLUSIONES Cont.. (II)



- + Aprovecha el tiempo de los clientes y ayuda a que un cliente se sienta integrado, evitando que se desmoralice por no saber como preparar pruebas de aceptación.
- + El proceso de desarrollo de las pruebas ayuda al cliente a clarificar y concretar la funcionalidad de la historia de uso y favorece la comunicación entre el cliente y el equipo de desarrollo.
- + El desarrollo de pruebas ayuda identificar y corregir fallos u omisiones en las historias de uso.



## CONCLUSIONES Cont.. (III)



- + Permite corregir errores en las ideas del cliente, por ejemplo encontrando resultados que el cliente espera encontrar en la implementación.
- + Permite identificar historias adicionales que no fueran obvias para el cliente o en las que cliente no hubiese pensado de no enfrentarse a dicha situación.
- + Garantiza la cobertura de la funcionalidad de las pruebas de aceptación, garantizando que no se deja ningún punto importante de la funcionalidad de una historia de uso sin probar.

# METODOLOGÍA CRYSTAL



De los Ríos Liz

# *Un poco de historia...*

A principios de los 90's se comenzaba a estudiar las distintas metodologías. En esos momentos estaban surgiendo las nuevas metodologías ágiles:

- XP (Extreme Programming)
- Scrum
- Agile Alliance
- Crystal Metodologies

Nuestro estudio se centra en Crystal Methodologies, que fue impulsada por Alistair Cockburn.

# ¿QUÉ ES CRYSTAL CLEAR ?

- Crystal Clear no es una metodología en si misma sino una familia de metodologías con un "código genético" común.
- El nombre Crystal deriva de la caracterización de los proyectos según 2 dimensiones, tamaño y complejidad (como en los minerales, color y dureza).

# **¿EN QUE CONSISTE?**

- Crystal da vital importancia a las personas que componen el equipo de un proyecto, y por tanto sus puntos de estudio son:
  - Aspecto humano del equipo
  - Tamaño de un equipo (número de componentes)
  - Comunicación entre los componentes
  - Distintas políticas a seguir
  - Espacio físico de trabajo

# EL EQUIPO DE TRABAJO

- Crystal aconseja que el tamaño del equipo sea reducido (Pocos componentes).
- La mejora de la comunicación entre los miembros del equipo del proyecto:
  - ✓ Mismo lugar de trabajo → Disminuye el costo de la comunicación

# CRYSTAL METHODOLOGIES

La familia Crystal dispone un código de color para marcar la complejidad de una metodología:

- Cuanto más oscuro un color, más "pesado" es el método.
- Cuanto más crítico es un sistema, más rigor se requiere.
- Crystal es fácil de aprender en implementar.

# FASES DE LA METODOLOGÍA CRYSTAL

Las prácticas en las metodologías Crystal son:

- **Puesta en escena (staging).** Consiste en la planificación del siguiente incremento. La planificación debe finalizar con una planificación ejecutable cada tres o cuatro meses.  
El equipo selecciona los requerimientos que serán implementados en el incremento y planifican lo que harán.
- **Revisiones.** Cada incremento tiene varias iteraciones y cada iteración incluye las actividades de construcción, demostración y resumen de objetivos del incremento.
- **Monitoreo.** Los progresos son monitoreados a partir de las diferentes entregas. El proceso se mide con los hitos clave y la estabilidad de las fases.

# FASES DE LA METODOLOGÍA CRYSTAL

- **Parallelismo y flujo.** Cuando el monitoreo nos brinda un estado suficientemente estable es hora de pasar a la próxima etapa. En CO nos indica que los equipos pueden trabajar con la máxima eficiencia concurrente.
- **Estrategia de diversidad holística.** Se utiliza en CO para dividir grandes equipos funcionales en equipos multifuncionales.
- **Técnica de puesta a punto de la metodología.** Se basa en entrevistas y talleres para laborar una metodología específica para el proyecto. Sirve para modificar o fijar el proceso de desarrollo.
- **Puntos de vista del usuario.** En CC se recomienda la opinión de dos usuarios por cada versión del producto, en CO tres revisiones por parte del cliente en cada iteración.

# CODIFICACIÓN DE COLORES

Dado que el tamaño del proyecto indica el método a utilizar, se estableció una clasificación por colores.

- Crystal Clear (3 a 8 personas).
- Crystal Yellow (10 a 20 personas).
- Crystal Orange (25 a 50 personas).
- Crystal Red (50 a 100 personas).

3-8	10-20	25-50	50- 100	100- 200	200- 500	500+
-----	-------	-------	------------	-------------	-------------	------

# CODIFICACIÓN DE COLORES



# CODIFICACIÓN DE COLORES

Crystal sugiere que escoger un color de la metodología para un proyecto en función de su criticidad y tamaño. Los proyectos más grandes suelen necesitar una mayor coordinación y metodologías más complejas que los proyectos más pequeños.

Cuanto más crítico sea el sistema que queremos desarrollar, más rigurosidad necesitamos disponer en el desarrollo del proyecto. En la figura anterior aparecen unos caracteres (C, D, E y L) e indican las pérdidas potenciales por fallos del sistema, y lo hacen de la siguiente manera:

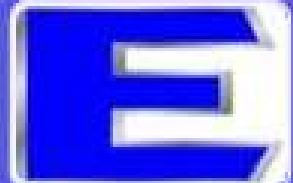
# CODIFICACIÓN DE COLORES



**Perdida de confort por fallas del sistema**



pérdida de dinero discrecional, es decir del que podemos disponer, generalmente nuestro



pérdida de dinero esencial, es decir dinero que probablemente no es nuestro y no podemos disponer de él libremente.



de Life en inglés, vida. Indica la pérdida de vidas por el fallo del sistema

## ROLES

Hay ocho roles nominados :

**1. Patrocinador.**

Produce la Declaración de Misión con Prioridades de Compromiso

**2. Usuario Experto.**

Junto con el Experto en Negocios produce la Lista de Actores Objetivos y el Archivo de Casos de Uso y Requerimientos.

**3. Diseñador Principal.**

Produce la Descripción Arquitectónica. Se supone que debe ser al menos un profesional de Nivel 3

## ROLES

**4. Diseñador Programador.**

Produce, junto con el Diseñador Principal, los Borradores de Pantallas

**5. Experto en Negocios.**

Junto con el Usuario Experto produce la Lista de Actores Objetivos.

**6. Coordinador.**

Con la ayuda del equipo, produce el Mapa de Proyecto, el Plan de Entrada, el Estado del Proyecto.

## ROLES

**7. Verificador.**

Produce los reportes.

**8. Escritor.**

Produce el Manual de Usuario.

# Herramientas y Técnicas

## HERRAMIENTAS:

- Catalogo Simple
- Caso de uso

## Colaboradora

- Requisito de diseño no funcional

## -Arquitectura

- Prueba de casos
- Diseño de Interfaz Usuario

## TÉCNICAS:

- Escribir casos de usos
- Tarjeta de Responsabilidad Clase

- Responsabilidad

- Programa de Derivación

# METODOLOGIAS DE CRYSTAL MAS CONOCIDAS

## Crystal Clear

- Crystal Clear se corresponde con el color Blanco en la codificación de colores de Crystal  
**3 – 8 personas**

## Crystal Orange

- Crystal Orange se corresponde con el color Naranja en la codificación de colores de Crystal  
**25 – 50 personas**

# Crystal Clear

- Es la menor de la familia de metodologías Crystal .
- Desarrollada por el investigador de IBM el Dr. Alistair Cockburn.
- Está diseñada para ser utilizada por equipos de hasta ocho integrantes y en el desarrollo de sistemas cuyos posibles errores puedan causar una pérdida prudencial de dinero o de confort.

# Crystal Clear

**"Es una metodología centrada en el factor humano, donde un diseñador líder y de dos a siete desarrolladores más se encuentran juntos en un local grande o en locales adyacentes con radiadores de información como pizarras y diagramas bien visibles en la pared, teniendo acceso fácil a usuarios claves; eliminando las distracciones; entregando código funcional, testeado y utilizable en intervalos de uno a tres meses; reflexionando periódicamente y ajustando continuamente su estilo de trabajo".**

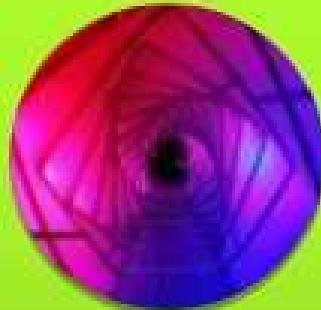
# Crystal Clear



Grupos de  
hasta 8  
personas



Reuniones en  
un mismo  
espacio



Proyectos  
pequeños



Proyectos no  
críticos



# Crystal Clear-Características

- 1. Entrega frecuente.** Consiste en entregar software a los clientes con frecuencia.
- 2. Retroalimentación continua.** El equipo entero se reúne constantemente para discutir las actividades del proyecto
- 3. Comunicación constante.** Se procura que cada uno de los miembros tengan acceso constante.
- 4. Seguridad.** Se reconoce la prioridad del software.

# Crystal Clear-Características

5. **Enfoque.** Saber lo que se está haciendo y tener la tranquilidad y el tiempo para hacerlo.
6. **Acceso a usuarios.** Acceso a uno o más usuarios del sistema que se están construyendo.
7. **Pruebas Automáticas e Integración.** Ambiente técnico con prueba automatizada, administración de configuración e integración frecuente.

# **CONCLUSIONES**

- ✓ Cuantas más personas estén implicadas, más grande debe ser la metodología.
- ✓ El aumento de tamaño o densidad añade un coste considerable al proyecto.
- ✓ Si el proyecto tiene mucha densidad, un error no detectado puede ser crítico.
- ✓ La forma más eficaz de comunicación es la interactiva (cara a cara).

# Metodología DSDM

**Dynamic Systems  
Development Method**



DSDM

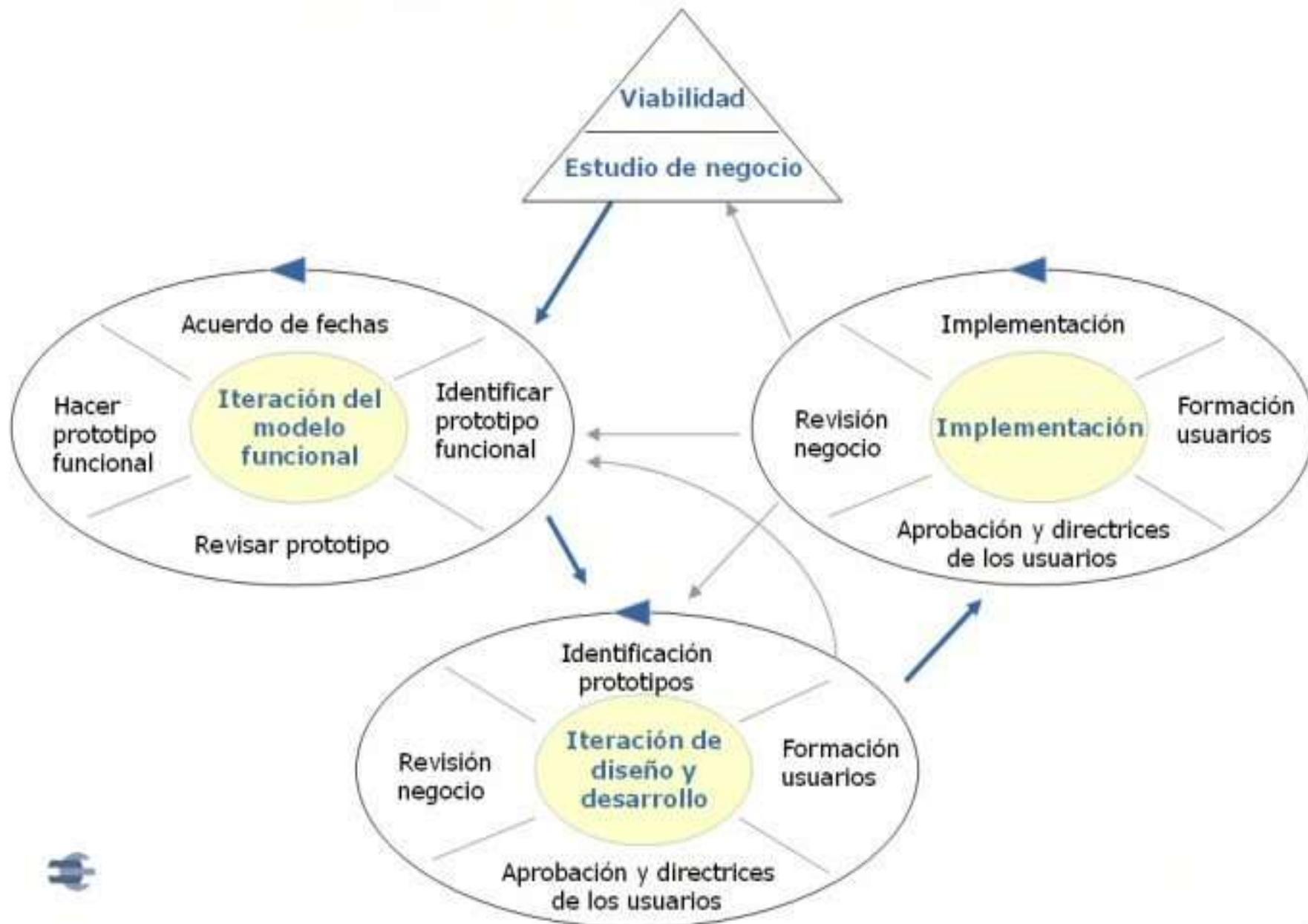
## ■ Principios

- 1. El involucramiento del usuario es imperativo.
- 2. Los equipos de DSDM deben tener el poder de tomar decisiones.
- 3. El foco está puesto en la entrega frecuente de productos.
- 4. La conformidad con los propósitos del negocio es el criterio esencial para la aceptación de los entregables.
- 5. El desarrollo iterativo e incremental es necesario para converger hacia una correcta solución del negocio.
- 6. Todos los cambios durante el desarrollo son reversibles.
- 7. Los requerimientos están especificados a un alto nivel.
- 8. El testing es integrado a través del ciclo de vida.
- 9. Un enfoque colaborativo y cooperativo entre todos los interesados es esencial.
-

DSDM define cinco fases en la construcción de un sistema :

- Estudio de factibilidad: si la metodología se ajusta al proyecto.
- Estudio del negocio: el alcance funcional a automatizar.
- Iteración del modelo funcional: diseño funcional
- Iteración del diseño y construcción, implantación: diseño técnico y desarrollo.

# Diagrama de procesos DSDM



- La metodología no tiene ninguna prescripción respecto a las técnicas a ser usadas en el proyecto, ni siquiera impone el desarrollo bajo un paradigma específico . Funciona tanto para el modelo de orientación a objetos como para el modelo estructurado.

## **Timebox :**

- Investigación: actividades que lo componen
- Refinamiento: desarrollo poniendo prioridades a las tareas
- Consolidación: completar los entregables verificando calidad

- **Roles:**
- Usuario embajador
- Coordinador técnico

- **Ventajas:**
- Todos los cambios durante el desarrollo son reversibles.
- Los requerimientos están especificados a un alto nivel
- El testing es integrado a través del ciclo de vida.

- **Desventajas:**
- De baja o media complejidad computacional
- Los equipos de DSDM deben tener el poder de tomar decisiones.
- Con flexibilidad en los requerimientos



Mgter. Oscar Adolfo Vallejos – FaCENA - UNNE

# QUE ES ?

Es un proceso de desarrollo de software diseñado por Peter Coad, Erich lefebvre y Jeff De Luca. Se basa en un proceso iterativo con iteraciones cortas que producen software funcional que el cliente y la dirección pueden ver y monitorizar.

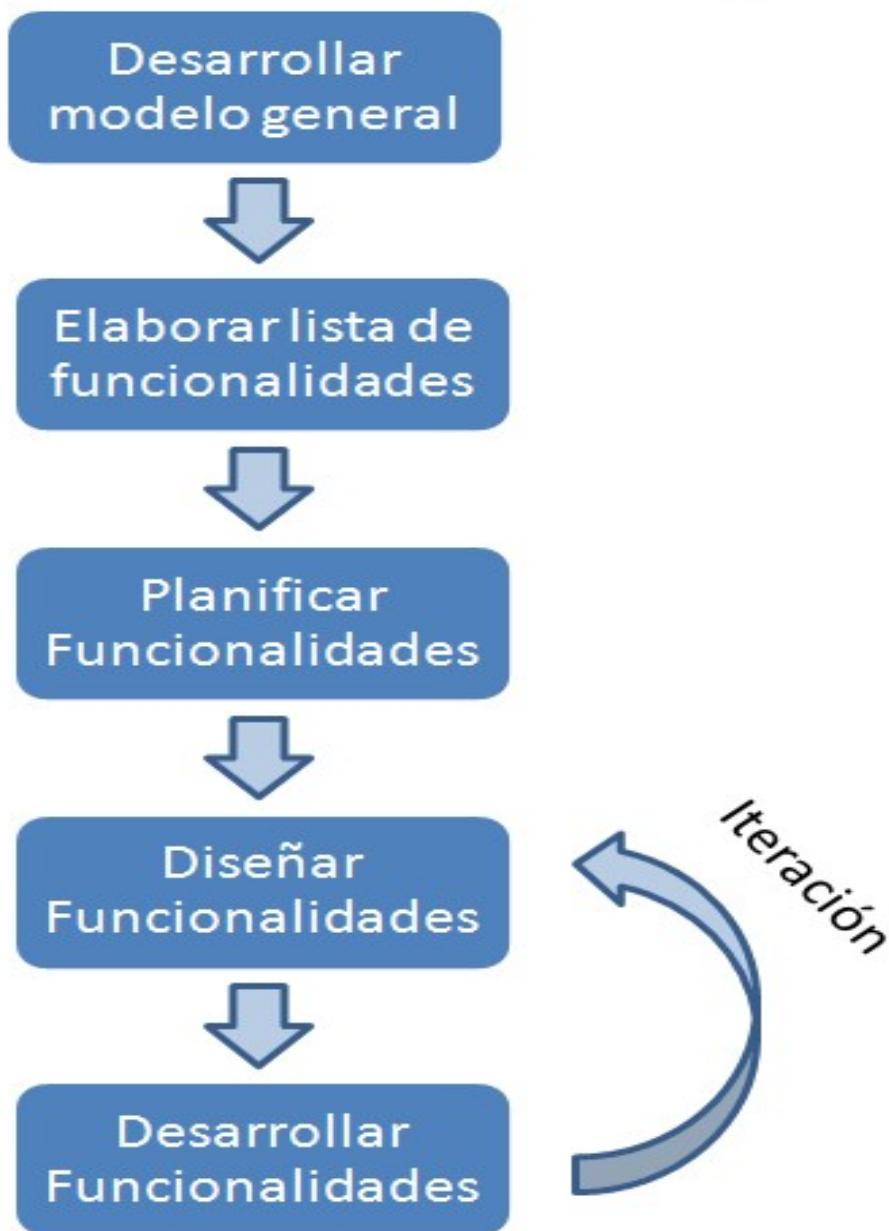


# CARACTERISTICAS

- Se preocupa por la calidad del software
- Ayuda a contrarrestar problemas
- Se obtienen resultados periódicos y tangibles
- No hace énfasis en la obtención de los requerimientos sino en como se realizan la fases de diseño y construcción



# Feature Driven Development (FDD)



# ROLES CLAVE

DIRECTOR DEL PROYECTO

ARQUITECTO JEFE

DIRECTOR DE DESARROLLO

EXPERTOS DE DOMINIO

PROPIETARIO DE CLASES



# ROLES DE SOPORTE

TESTER

HERRAMIENTISTA

IMPLEMENTADOR

RELEASE MANAGER

GURU DEL LENGUAJE

ADMINISTRADOR DE DOMINIO

INGENIERO DE CONSTRUCCION

ESCRITORES DE DOCUMENTOS TECNICOS





AGILE

# *Construir software no es como construir coches o casas*

## **La predictibilidad, ciclo de vida en cascada o desarrollo tradicional**

- En su nacimiento, la **gestión de proyectos software** intentó imitar la gestión de proyectos de otras disciplinas, como la arquitectura, las industria o la ingeniería civil, hasta el punto de heredar y adaptar al mundo del software muchos de sus roles.
- Hoy en día una de las prácticas más discutidas y polémicas de las que se han querido heredar desde otras disciplinas es la llamada **predictibilidad**, también conocida como **gestión de proyectos dirigida por la planificación, desarrollo tradicional o incluso también conocida como desarrollo pesado**.



- La **predictibilidad** se basa en dividir un proyecto en fases, por ejemplo, de manera simplificada, “requisitos”, “diseño” y “construcción”, y que cada una de estas fases no comience hasta que termine con éxito la anterior.
- **Se le llama predictibilidad porque cada fase intenta predecir lo que pasará en la siguiente;** por ejemplo, la fase de diseño intenta predecir qué pasará en la programación, y esos diseños intentarán ser muy precisos y detallados, para ser cumplidos sin variación por los programadores.
- Además, **en este tipo de gestión, cada una de estas fases se realiza una única vez** (no hay dos fases de requisitos). Y las fases están claramente diferenciadas (en teoría, está claro cuándo termina el diseño y comienza la programación), hasta el punto de **tener profesionales claramente diferenciados y especializados en cada una de ellas**: “analistas de requisitos”, “arquitectos de diseño software”, programadores, personas para pruebas, etc.
- Normalmente **cada fase concluye con un entregable documental que sirve de entrada a la siguiente fase**, la “especificación de requisitos software” es la entrada al diseño, el “documento de diseño” la entrada a la construcción, etc.

# Ágil vs. Tradicional

- **En software, la experiencia nos dice que es muy difícil especificar los requisitos en una única y primera fase.**
- **También es muy difícil documentar de una única vez**, a la primera, antes de la codificación, un diseño que especifique de manera realista y sin variación todas las cuestiones a implementar en la programación.
- **Las ingenierías clásicas o la arquitectura** necesitan seguir este tipo de **ciclos de vida en cascada o predictivos** porque precisan mucho de un diseño previo a la construcción, exhaustivo e inamovible.
- Además, **los planos para construir son precisos y pocas veces varían**, ya que la mayoría de los diseños de las ingenierías clásicas, arquitecturas, etc., pueden hacer un mayor uso de las matemáticas o la física. **En software no es así**. Y aunque se pretenda emular ese modo de fabricación, en software no funciona bien, y debemos tener muy claro que **es casi imposible cerrar un diseño a la primera para pasarlo a programación sin tener que modificarlo posteriormente**.

- **Realizar un cambio en el producto final que construyen las ingenierías clásicas o la arquitectura es muy costoso.** Y además, normalmente, en la arquitectura o en las ingenierías clásicas los costes de construir son muy elevados en comparación con los de diseñar. El coste del equipo de diseñadores es sustancialmente inferior al de la realización de la obra, del puente, edificio, etc.
- **La anterior relación de costes no se comporta igual en el caso del software.** Por un lado, el software, por su naturaleza (y si se construye mínimamente bien), es más fácil de modificar.
- **En software no existe esa división tan clara entre los costes del diseño y los de la construcción.**
- **También en las ingenierías clásicas o la arquitectura los roles y especialización necesaria en cada fase son diferentes.** Los planos o diseños los realizan arquitectos que no suelen participar en la fase de construcción. La construcción tiene poco componente intelectual y mucho manual, al contrario que el diseño. Y todo apoya a que existan dos actividades claramente diferenciadas: el diseño y la construcción.
- **Se han intentado encontrar metodologías que imitasen y replicasen los procesos** de construcción tradicional al software. (diseño como UML, o metodologías como RUP [2] o Métrica v3).
- Sin embargo en muchas ocasiones, estos intentos de emular la construcción de software a productos físicos **han creado importantes problemas** y algunos de los mayores errores a la hora de gestionar proyectos software.
- Diferenciar el cómo se construye software del cómo se construyen los productos físicos es uno de los pilares de las metodologías ágiles.
- Y es que en software, es frecuente que diseño y construcción muchas veces se solapen, y por ello se recomienda construir por iteraciones, por partes, y el uso de prototipos incrementales.

## Cascada

- Las fases del ciclo de vida (requisitos, análisis, diseño, etc.) se realizan (en teoría) de manera lineal, una única vez, y el inicio de una fase no comienza hasta que termina la fase anterior.
- Su naturaleza es lineal, típica de la construcción de productos físicos y su principal problema viene de que no deja claro cómo responder cuándo el resultado de una fase no es el esperado.
- El ciclo de vida más criticado en los últimos años. En muchos proyectos su implantación ha sido un fracaso, mientras que hay otros proyectos que trabajan perfectamente de esta manera.

## El ciclo de vida incremental

Cada iteración (una iteración es un periodo de tiempo, no confundir con el ciclo de vida iterativo, que veremos luego, siendo este punto confuso, por las definiciones) contiene las fases del cascada estándar, pero cada iteración trabaja sobre un subconjunto de funcionalidad.

La entrega total del proyecto se divide en subsistemas priorizados.

Desarrollar por partes el producto software, para después integrarlas a medida que se completan. Un ejemplo de un desarrollo puramente incremental puede ser la agregación de módulos en diferentes fases. El agregar cada vez más funcionalidad al sistema.

## El ciclo de vida iterativo

En cada ciclo, iteración, se revisa y mejora el producto. Un ejemplo de desarrollo iterativo es aquel basado en refactorizaciones, en el que cada ciclo mejora más la calidad del producto. Es importante señalar que este ciclo no implica añadir funcionalidades en el producto, pero si la revisión y la mejora.

### Iterativo e incremental

Incremental = añadir, iterativo = retrabajo

Se va liberando partes del producto (prototipos) periódicamente, en cada iteración, y cada nueva versión, normalmente, aumenta la funcionalidad y mejora en calidad respecto a la anterior. Aquí hay [un post](#) con más información.

Además, el ciclo de vida iterativo e incremental es una de las bases de un proyecto ágil, más concretamente, con iteraciones cortas en tiempo, de pocas semanas, normalmente un mes y raramente más de dos

# Ciclo de vida ágil

- Es **ciclo de vida iterativo e incremental**, con **iteraciones cortas (semanas)** y sin que dentro de cada iteración tenga porque haber **fases lineales**.
- Quizá el caso más popular es el de **Scrum**. (en ell 85, primera presentación oficial de Scrum, Ken Schwaber).
- Según comenta Ken Schwaber, **el ciclo de vida en Cascada y el Espiral cierran el contexto y la entrega al inicio de un proyecto**. La principal diferencia entre **cascada, espiral e iterativo y los ciclos de vida ágiles**, concretamente en Scrum, es que estos últimos asumen que el análisis, diseño, etc., de cada iteración o Sprint son impredecibles. **Los Sprints, o iteraciones cortas, no son (o a priori no tienen porqué) lineales y son flexibles**.
- **Cada metodología de las llamadas ágiles**, FDD, Crystal, DSDM, XP, Scrum etc., matizará su **ciclo de vida**.

# El proyecto ágil

- Un **proyecto ágil** se podría definir como una manera de **enfocar el desarrollo software** mediante un **ciclo iterativo e incremental**, con **equipos que trabajan de manera altamente colaborativa y autoorganizados**.
- En un **proyecto ágil** cada **iteración** no es un “**mini cascada**”. Cuanto menor es el **tiempo de iteración** más **se solapan las tareas**. Hasta el punto que implicará que de manera no secuencial, muchas veces solapada, y repetitivamente, durante una iteración se esté casi a la vez diseñando, programando y probando.
- **Implica máxima colaboración e interacción de los miembros del equipo.** **Equipos multidisciplinares.** E **implica auto-organización.**

# **El proyecto ágil**

**Un proyecto ágil lleva la iteración al extremo:**

- 1. Se busca **dividir las tareas del proyecto software en incrementos de una corta duración.**
- 2. **Cada iteración suele concluir con un prototipo operativo. Al final de cada incremento se obtiene un producto entregable que es revisado junto con el cliente.**

# **El Manifiesto Ágil**

El 12 de febrero de 2001, 17 destacados y conocidos profesionales de la ingeniería del software escribían en Utah el **Manifiesto Ágil**.

- **Valorar a los individuos y las interacciones del equipo de desarrollo sobre el proceso y las herramientas.**
- **Desarrollar software que funciona más que conseguir una documentación exhaustiva.**
- **La colaboración con el cliente más que la negociación de un contrato.**
- **Responder a los cambios más que seguir estrictamente un plan.**

## **LO QUE NO DICE**

- **Ausencia total de documentación; Ausencia total de planificación: planificar y ser flexible es diferente a improvisar; El cliente debe hacer todo el trabajo y será el Jefe de Proyecto; El equipo puede modificar la metodología sin justificación.**

# Los Principios Ágiles

- La prioridad es **satisfacer al cliente** mediante entregas tempranas y continuas de software que le aporten valor.
- **Dar la bienvenida a los cambios.** Se capturan los cambios para que el cliente tenga una ventaja competitiva.
- **Entregar frecuentemente software que funcione** desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.
- **La gente del negocio y los desarrolladores deben trabajar juntos** a lo largo del proyecto.
- **Construir el proyecto en torno a individuos motivados.** Darles el entorno y el apoyo que necesitan y confiar en ellos para conseguir finalizar el trabajo.
- **El diálogo cara a cara es el método más eficiente** y efectivo para comunicar información dentro de un equipo de desarrollo.
- **El software que funciona es la medida fundamental de progreso.**
- Los procesos ágiles **promueven un desarrollo sostenible**. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.
- **La atención continua a la calidad técnica y al buen diseño** mejora la agilidad.
- **La simplicidad** es esencial.
- Las mejores arquitecturas, requisitos y diseños surgen de los **equipos organizados por sí mismos**.
- En intervalos regulares, el equipo reflexiona respecto a cómo llegar a **ser más efectivo**, y según esto ajusta su comportamiento.

# Gestión de equipos

- 1. Buscar a los miembros más adecuados y retenerlos:** es esencial el talento, hay que buscar la persona más adecuada para el tipo de tecnología y proyecto que se está desarrollando, y una vez que se tiene a la persona idónea conseguir retenerla.
- 2. Trabajar en un entorno de productividad, sin interrupciones:** los entornos físicos tienen un impacto altísimo en la productividad. Uno de los principales impactos vienen de las interrupciones, como solución encontramos la técnica **pomodoro** (técnica de gestión personal).
- 3. Conocer el impacto de la NO calidad:** si un software está mal hecho, la mantenibilidad baja, y conlleva a que la productividad baje (como una mala solución a ello meten más gente a trabajar; una buena solución es refactorizar) y como consecuencia se disparan los costes. La calidad afecta mucho a la productividad.
- 4. Equipos pequeños:** el tamaño de los equipos es una de las claves más importantes en la gestión de equipos. Añadir gente a un proyecto retrasado hace que te retrase más.
- 5. Equipos multifuncionales (sin héroes ni apagafuegos):** un buen equipo no tiene apaga fuegos. Un equipo multifuncional es un equipo en el que hay un equilibrio entre sus miembros, es decir, cada uno tiene su tarea, pero en momentos específicos puede realizar otras.

## **Trabajar en más de un proyecto a la vez genera pérdida de tiempo y disminuye la productividad**

- Weinberg en 1992, fue famoso principalmente por un estudio en el que explicaba el **desperdicio de tiempo** que suponía tener a las personas trabajando en varios proyectos a la vez.

Número de proyectos simultáneos	% de disponibilidad para el proyecto	Pérdida debido al cambio de contexto
1	100 %	0 %
2	40 %	20 %
3	20 %	40 %
4	10 %	60 %
5	5 %	75 %

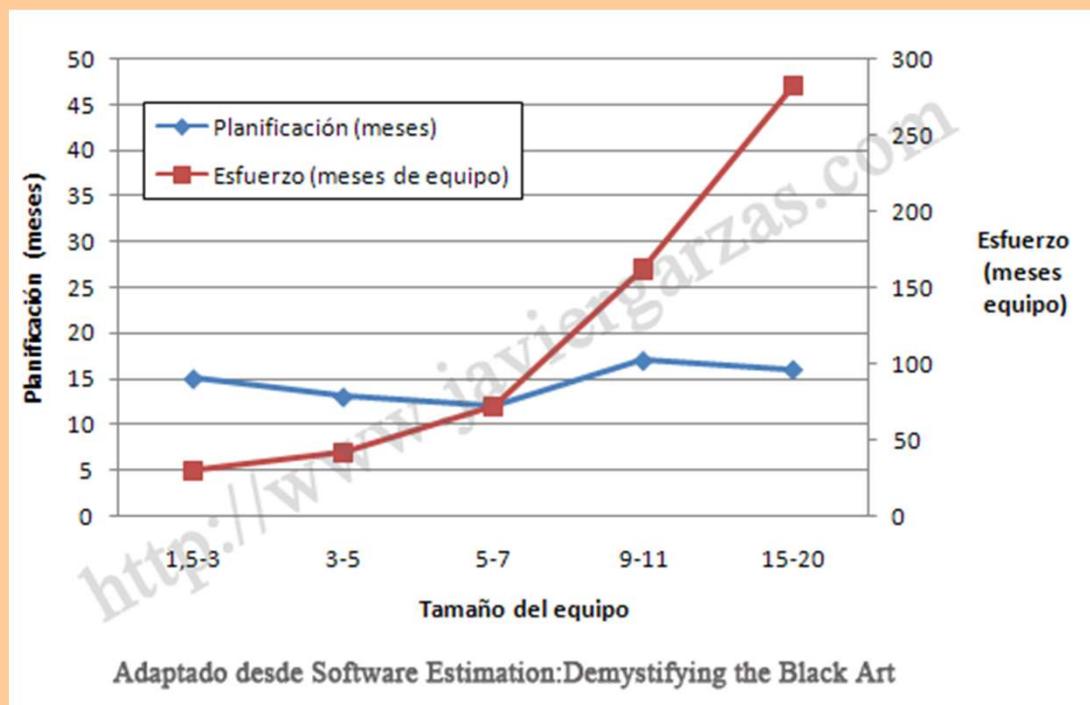
# **Interrumpir a quién programa, o al que realiza cualquier actividad intelectual, hace que su productividad caiga**

- En la actividades de programar, post, diseñar, etc., las interrupciones son un mal que afecta enormemente a la productividad.
- Parnin y Rugaber hicieron un estudio sobre las interrupciones en el año 2010, siendo su conclusión más destacada la siguiente:
  1. **Lo normal es que a un programador le lleve de 10 a 15 minutos volver al estado de concentración previo al haber sido interrumpido.**
  2. La **interrupción** pueden ser:
    1. **externa**
    2. **Interna (procastinación)**

Dos consejos son:

- Utilizar la técnica “pomodoro”.
- Para luchar contra interrupciones externas. Aíslate, en la medida de lo posible, mañanas, horas o días, a entornos sin interrupciones.

# Los equipos con mucha gente son menos productivos



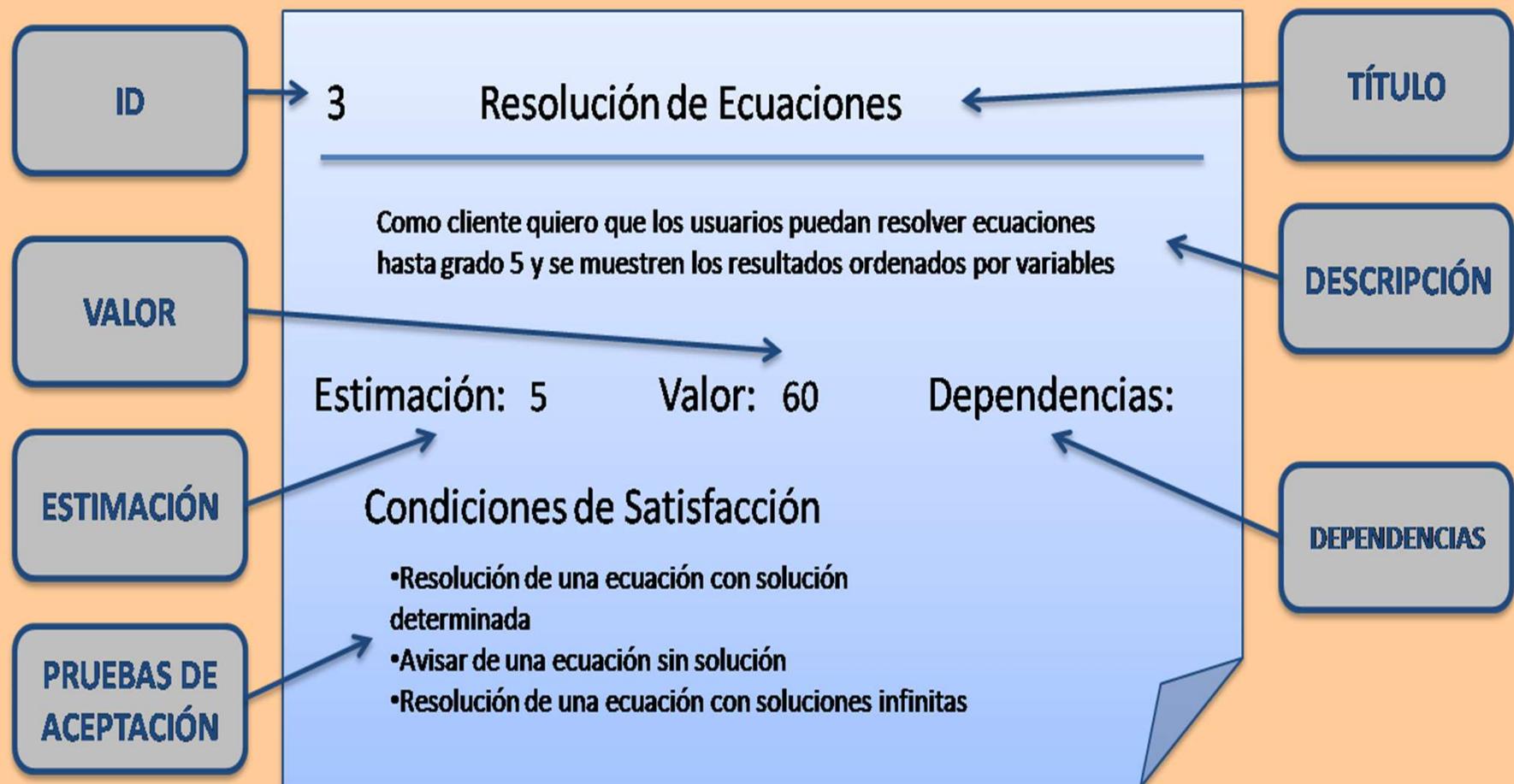
# El “Product Owner” y las historias de usuario

- El “product owner” (o propietario del producto) es aquella persona con una visión muy clara del producto que se quiere desarrollar, que es capaz de transmitir esa visión al equipo de desarrollo y, además, está altamente disponible para transmitirla.
- La figura del product owner es clave en un proyecto ágil, en su planificación y seguimiento.
- Desgraciadamente, es frecuente encontrar implantaciones erróneas alrededor de este rol. En unas ocasiones sus tareas se minimizan, en otras suelen pasarse por alto muchas de sus importantes responsabilidades, etc.
- La mayoría de las ocasiones, el negocio, los usuarios, etc., van proporcionando una cantidad de ideas a implementar, que se van convirtiendo en historias de usuario, muy superiores en número. La función del product owner es vital, debe ser quien decida qué historias de usuario entran en el product backlog, cuáles no, y además la prioridad de las historias del product backlog.

# Historias de usuario

- En las metodologías ágiles la descripción de las necesidades se realiza a partir de las **historias de usuario (user story)** que son, principalmente, lo que el cliente o el usuario quiere que se implemente; es decir, **son una descripción breve, de una funcionalidad software tal y como la percibe el usuario.**
- El concepto de **historia de usuario** tiene sus raíces en la **metodología “eXtremeProgramming”** o programación extrema.
- No obstante, las **historias de usuario** se adaptan de manera apropiada a la mayoría de las metodologías ágiles, teniendo por ejemplo, un papel muy importante en la **metodología Scrum.**

# Qué información contiene una historia de usuario



- Cohn en (M. Cohn, 2004) [1] comenta que **si bien las historias de usuario son lo suficientemente flexibles como para describir la funcionalidad de la mayoría de los sistemas, no son apropiadas para todo. Si por cualquier razón, se necesita expresar alguna necesidad de una manera diferente a una historia de usuario, recomienda que se haga.** Por ejemplo, las interfaces de usuario se suelen describir en documentos con muchos pantallazos. Al igual puede ocurrir con documentos especificaciones de seguridad, normativas, etc.
- No obstante, **lo que sí es importante con esta documentación adicional es mantener la trazabilidad con las historias de usuario.** Por ejemplo, a través de hojas de cálculo donde se lleve el control de a qué historia pertenece cada documento adicional, o especificando el identificador de la historia en algún apartado del documento, etc.

# ¿Las historias de usuario equivalen a requisitos funcionales?

Popularmente se asocia el concepto de historia de usuario con el de la especificación de un requisito funcional.

- **Una historia de usuario describe funcionalidad que será útil para el usuario y/o cliente de un sistema software.** Y aunque normalmente las historias de usuario asociadas a las metodologías ágiles, suelen escribirse en pósit o tarjetas, son mucho más que eso. Como se ha comentado anteriormente, una historia no es sólo una descripción de una funcionalidad, sino también es de vital importancia la conversación que conllevan.
- **Las historias de usuario, frente a mostrar el “cómo”, sólo dicen el “qué”. Es decir, muestran funcionalidad que será desarrollada, pero no cómo se desarrollará.**
- De esta manera, **equiparar las historias de usuario con las especificaciones de requisitos no es demasiado correcto** ya que por definición, las historias de usuario no deben tener el nivel de detalle que suele tener la especificación de un requisito
- **Una historia de usuario debería ser pequeña, memorizable, y que pudiera ser desarrollada por un par de programadores en una semana.**
- Para resolver el anterior problema hay que entender que **el objetivo de las historias de usuario es, entre otros, lograr la interacción entre el equipo y el cliente o el usuario por encima de documentar** (manifesto ágil) **por lo que no se deben sobrecargar de información.** Sin embargo, la realidad de los proyectos y de los negocios es otra y hace que la teoría se deba ajustar a la práctica, por lo que se pueden dar varias soluciones para reflejar toda esa información que en un primer momento parece que no cuadra en las historias de usuario: (los tradicionales requisitos en ciclos de vida ágil; Usar casos de uso; Utilizar técnicas de trazabilidad para relacionar las historias de usuario con otros documentos: de diseño, normativas, etc.).

# ¿Las historias de usuario equivalen a casos de uso?

- Básicamente, **si decíamos que una historia de usuario es el “qué” quiere el usuario, el caso de uso es un “cómo” lo quiere.**
- En un proyecto de una metodología ágil, se deberían olvidar completamente los casos de uso y el equipo debería centrarse en la realización de historias de usuario, esto puede producir los siguientes problemas:
  - Las historias de usuario no proporcionan a los diseñadores un contexto desde el que trabajar. Pueden no tener claro cuál es el objetivo en cada momento. ¿Cuándo le surgiría al cliente o usuario esta necesidad?
  - Las historias de usuario no proporcionan al equipo de trabajo ningún sentido de completitud. Se puede dar el caso que el número de historias de usuario no deje de aumentar, lo que puede provocar desmotivación en el equipo. Realmente, ¿cómo de grande es el proyecto?
  - Las historias de usuario no son un buen mecanismo para evaluar la dificultad del trabajo que está aún por llegar.
- Por tanto, si en un proyecto ocurre alguno de estos problemas se puede barajar la posibilidad (relajando la agilidad) de complementar las necesidades descritas en las historias de usuario con casos de uso donde quede reflejado el comportamiento necesario para cumplir dichas necesidades.
- **En el caso de que se usen las historias de usuario y los casos de uso de manera complementaria, una historia de usuario suele dar lugar a la especificación de varios casos de uso.**

# Creando buenas historias de usuario

- Bill Wake describió en 2003 un método llamado **INVEST** (Wake, 2003) [1]. El método se usa para comprobar la calidad de una historia de usuario revisando que cumpla las características descritas en la Tabla.

Independent (independiente)	Es importante que cada historia de usuario pueda ser planificada e implementada en cualquier orden. Para ello deberían ser totalmente independientes (lo cual facilita el trabajo posterior del equipo). Las dependencias entre historias de usuario pueden reducirse combinándolas en una o dividiéndolas de manera diferente.
Negotiable (negociable)	Una historia de usuario es una descripción corta de una necesidad que no incluye detalles. Las historias deben ser negociables ya que sus detalles serán acordados por el cliente/usuario y el equipo durante la fase de "conversación". Por tanto, se debe evitar una historia de usuario con demasiados detalles porque limitaría la conversación acerca de la misma.
Valuable (valiosa)	Una historia de usuario tiene que ser valiosa para el cliente o el usuario. Una manera de hacer una historia valiosa para el cliente o el usuario es que la escriba el mismo.
Estimable (estimable)	Una buena historia de usuario debe ser estimada con la precisión suficiente para ayudar al cliente o usuario a priorizar y planificar su implementación. La estimación generalmente será realizada por el equipo de trabajo y está directamente relacionada con el tamaño de la historia de usuario (una historia de usuario de gran tamaño es más difícil de estimar) y con el conocimiento del equipo de la necesidad expresada (en el caso de falta de conocimiento, serán necesarias mas fases de conversación acerca de la misma).
Small (pequeña)	Las historias de usuario deberían englobar como mucho unas pocas semanas/persona de trabajo. Incluso hay equipos que las restringen a días/persona. Una descripción corta ayuda a disminuir el tamaño de una historia de usuario, facilitando su estimación.
Testable (comprobable)	La historia de usuario debería ser capaz de ser probada (fase "confirmación" de la historia de usuario). Si el cliente o usuario no sabe como probar la historia de usuario significa que no es del todo clara o que no es valiosa. Si el equipo no puede probar una historia de usuario nunca sabrá si la ha terminado o no.

# **Asignar valor a una historia de usuario**

**Dicho valor es asignado por el Product Owner, y pondera básicamente las siguientes variables:**

- Beneficios de implementar una funcionalidad
- Pérdida o coste que demande posponer la implementación de una funcionalidad
- Riesgos de implementarla
- Coherencia con los intereses del negocio
- Valor diferencial con respecto a productos de la competencia

# Asignar valor a una historia de usuario

- Es muy recomendable incluir algún tipo de escala cualitativa.
- Una manera rápida de empezar a asignar valor a las historias es dividirlas en 3 grupos, según sean imperativas, importantes o prescindibles.
- técnica MoSCoW. (2004). Su fin es obtener el entendimiento común entre cliente y el equipo del proyecto sobre la importancia de cada requisito o historia de usuario. La clasificación es la siguiente:
  - **M - MUST**. Se debe tener la funcionalidad. En caso de que no exista la solución a construir fallará.
  - **S - SHOULD** Se debería tener la funcionalidad. La funcionalidad es importante pero la solución no fallará si no existe.
  - **C - COULD** Sería conveniente tener esta funcionalidad. Es en realidad un deseo.
  - **W - WON'T** No está en los planes tener esta funcionalidad en este momento. Posteriormente puede pasar a alguno de los estados anteriores.

# Herramienta

- Las *herramientas de la Ingeniería del software proporcionan* un enfoque automático o semi-automático para el proceso y para los métodos. Cuando se integran herramientas para que la información creada por una herramienta la pueda utilizar otra, se establece un sistema de soporte para el desarrollo del software llamado *ingeniería del software asistida por computadora (CASE)*.

# Herramientas CASE-I

***Las herramientas Case son ayudas automatizadas para el desarrollo del software***

En los últimos años la utilización y uso de estas herramientas ha experimentado un fuerte crecimiento debido a:

- Dificultad para poner en funcionamiento ciertos procedimientos de las metodologías actuales de desarrollo si no es con la ayuda de herramientas informáticas (diseños gráficos complejos, tareas repetitivas, modelización o prototipado de sistemas, etc.)
- Rigor en todo el proceso de análisis. (Control de Proyectos).
- Facilitan la representación de flujos de datos y procesos haciendo más fácil el diálogo técnico-usuario.
- Necesidad de contar con herramientas capaces de diseñar a nivel lógico y físico la aplicación a partir del análisis.

Las herramientas Case se pueden clasificar de manera general en dos categorías:

1. *Herramientas de Alto Nivel (Front End)*: Automatizan las fases de análisis y diseño lógico. Se pueden citar los productos comerciales Rational, Designer 2000 de Oracle, Visible Analyst y EasyCase, Enterprise Architect, entre otros muchos.
2. *Herramientas de Bajo Nivel (Back End)*: Automatización de la codificación y posterior mantenimiento. Como ejemplo se pueden incluir todas las herramientas de RAD, tales como Delphi o Visual Basic de Microsoft.

# Herramientas

**Las herramientas (CASE-I)** son un soporte automático o semiautomático para el proceso y los métodos.

- o Microsoft Project (Planificación).
- o UML (Modelado).
- o RationalRose, visio (Modelado soportan UML).
- o V.A. 7.0 (AS/DS con posibilidad ADOO)
- o E.A. (UML)
- o Designer 2000.
- o Erwin (Bases de datos).
- o MAGERIT (Seguridad).

# Un caso práctico sencillo ...

**Breve introducción a Scrum** Caso práctico

FLOWERS IN SPACE  
flowersinspace.com

A small gray robot icon is positioned on the right side of the slide.

Un cliente se pone en contacto con una empresa que fabrica robots.

El cliente les realiza el pedido.

Quiero un robot que me sirva de escolta



FLOWERS IN SPACE  
flowersinspace.com

## Breve introducción a Scrum Caso práctico



El Cliente se reune con el Dueño de producto, que toma nota de lo que tiene en su cabeza.





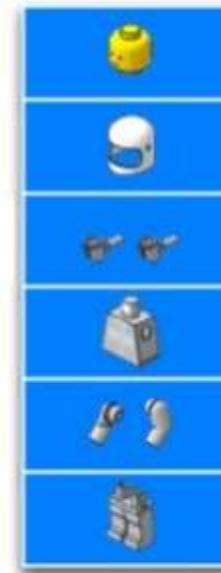
## Breve introducción a Scrum Caso práctico



El Dueño de Producto divide el proyecto en historias que son las que componen la pila de producto.



**Dueño de Producto**

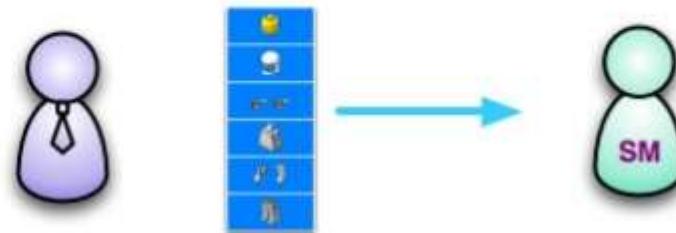


**Pila de Producto**



El Scrum Master es un miembro del equipo que tiene el papel de comunicar y gestionar las necesidades del Dueño de Producto y la pila de Sprint.

El Dueño de Producto le entrega la pila de producto para que estimen el coste de creación del producto.



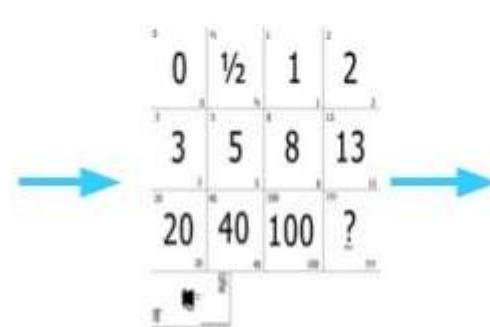
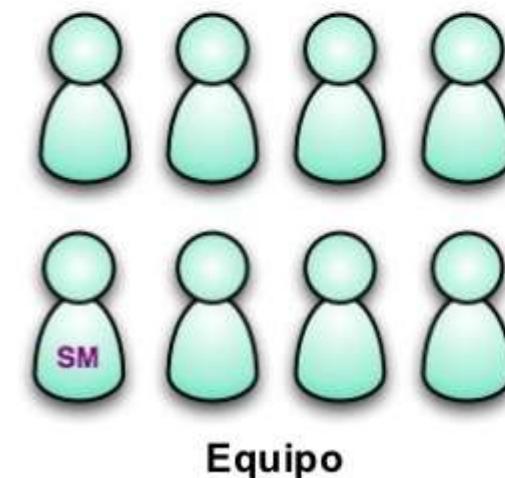
**Dueño de Producto**

**Scrum Manager**



El equipo se reúne para estimar el coste de cada historia de la pila de producto.

En este caso utilizan Planning Poker.



A 2x4 grid of numbers for Planning Poker estimation, ranging from 0 to 100+.

0	½	1	2
3	5	8	13
20	40	100	?



A table of estimated values for each item, with icons next to the numbers.

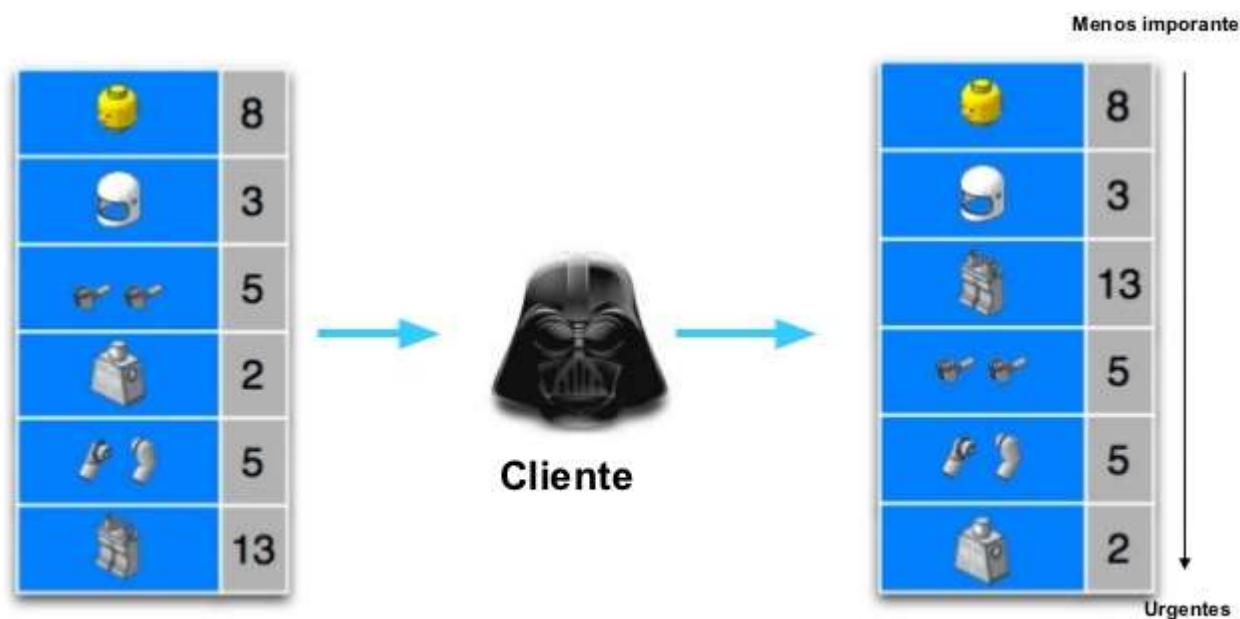
8
3
5
2
5
13



## Breve introducción a Scrum Caso práctico

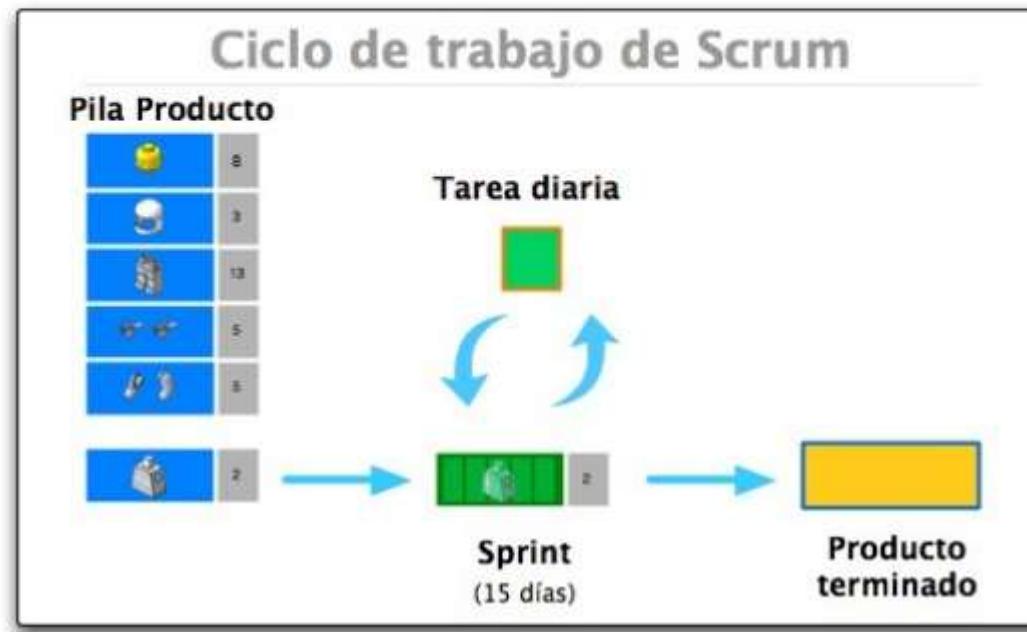


El cliente, una vez aprobado el presupuesto, reordena la pila de producto para que el equipo vaya trabajando según la prioridad del cliente.





El equipo comienza su trabajo desglosando la primera historia de la pila de producto, la cual subdividen en tareas menores para crear la pila de sprint.





## Breve introducción a Scrum

Caso práctico



La pila de sprint tiene como utilidad fraccionar el trabajo de un periodo de 15 días en tareas mas pequeñas, que tarden como mucho dos días.





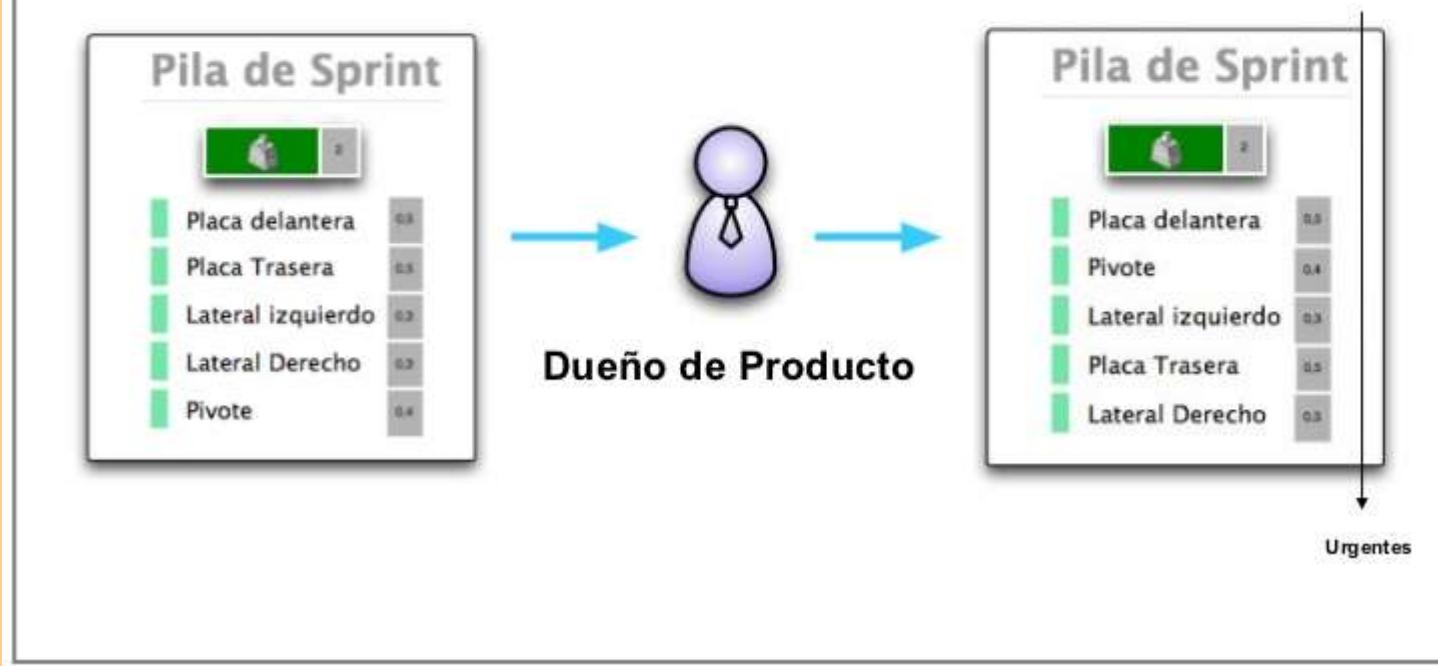
## Breve introducción a Scrum

Caso práctico



Estas tareas se colocan en una pila, la cual prioriza el Dueño de Producto, que ha consultado con el cliente, antes de comenzar el sprint.

Menos importantes





## Breve introducción a Scrum Caso práctico



El equipo comienza el sprint tomando las tareas priorizadas. Una vez concluida una se toma la siguiente de la lista. Se convoca todos los días una reunión del equipo donde se cuenta las tareas realizadas el día anterior y cuales se van a realizar ese día.





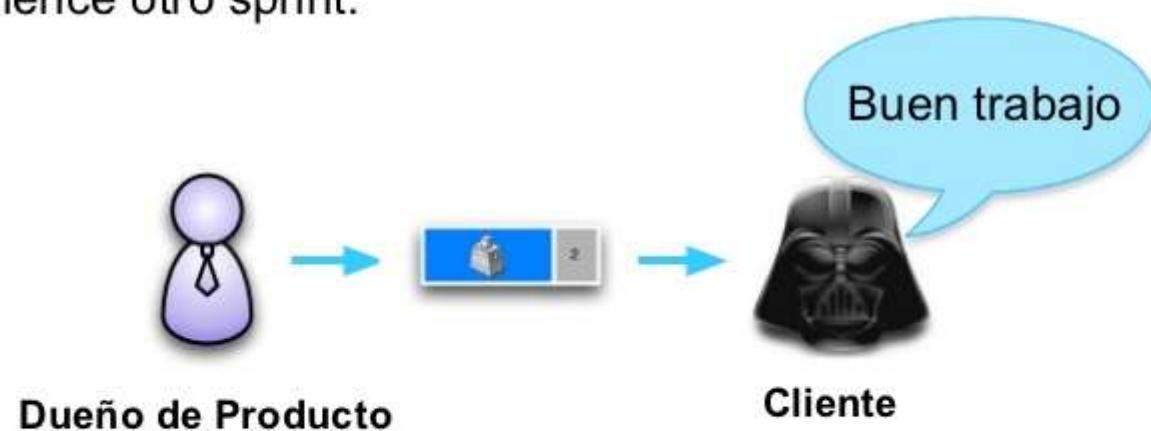
## Breve introducción a Scrum

Caso práctico



Una vez finalizado el sprint, el Dueño de Producto le muestra al cliente el resultado del trabajo realizado.

El cliente ya tiene el primer contacto con su encargo y además puede volver a priorizar la pila de producto antes de que comience otro sprint.



# Bibliografia

- **Plantillas de la Clases de Teoria. Mgter. Vallejos, Oscar A.**
- **Capitulo IV de: INGENIERÍA DEL SOFTWARE. Séptima edición Ian Sommerville. ISBN: 84-7829-074-5. PEARSON ADDISON WESLEY.**
- **Capitulo II de: INGENIERÍA DEL SOFTWARE. Un enfoque practico. Séptima edición. Roger S. Pressman. ISBN: 978-607-15-0314-5. MC GRAW HILL**
- **La Programación Extrema en la practica. James Robert. Iberoamericana España S.A. 2002.**

## Link

- <http://www.software-engin.com>