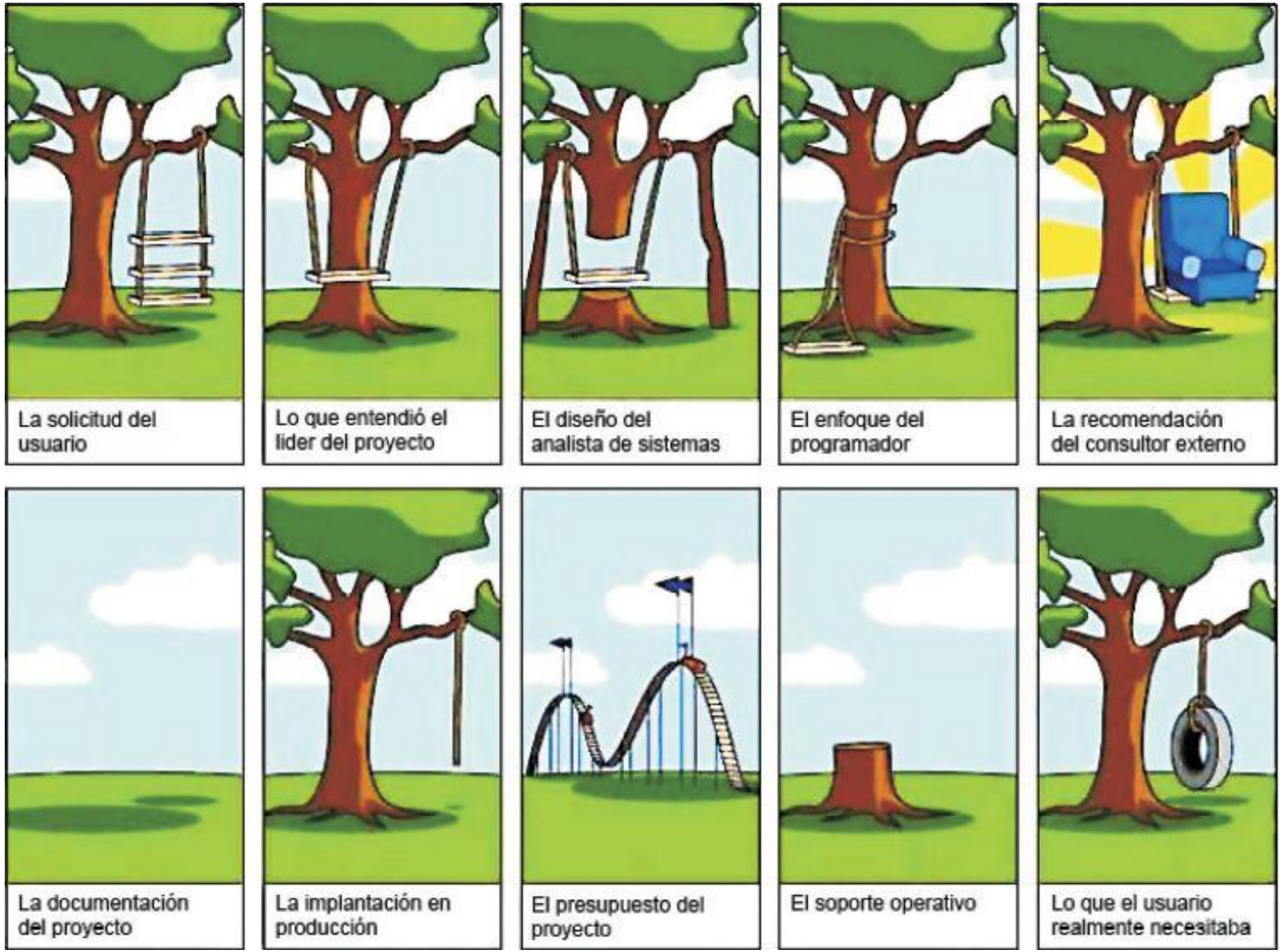
Ingeniería de Software

Introducción a la Ingeniería de Software

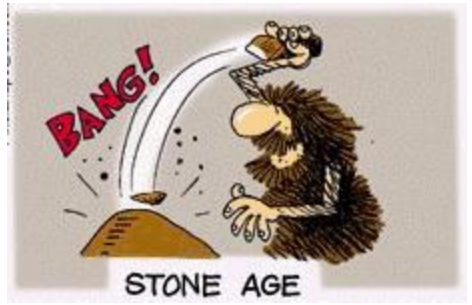

La solicitud del usuario
Lo que entendió el lider del proyecto
El diseño del analista de sistemas
El enfoque del programador
La recomendación del consultor externo
La documentación del proyecto
La implantación en producción
El presupuesto del proyecto
El soporte operativo
Lo que el usuario realmente necesitaba

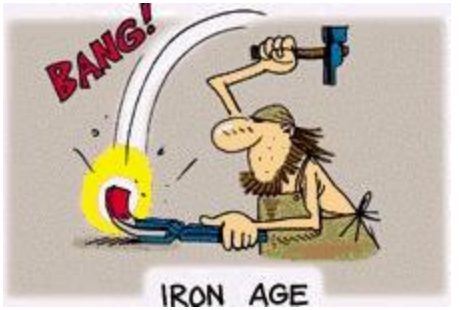
Software

- ¿Qué es el software?
  - **La suma total de** los programas de cómputo, procedimientos, reglas de documentación y datos asociados que forman parte de las operaciones de un sistema de cómputo [IEEE Computer Society Press, 1993].
  - **Es un producto** que diseñan y construyen los ingenieros de software. Esto abarca ***programas*** que se ejecutan dentro de una computadora de cualquier tamaño y arquitectura, ***documentos*** que comprenden formularios virtuales e impresos y ***datos*** que combinan números y texto y también incluyen representaciones de la información de audio, vídeo e imágenes [Pressman, 2002].

Evolución del software

- Primeros años (principios de los 50's a mediados de los 60's)
  - Lo más importante era el hardware, el software solo era un añadido a la medida.
  - El desarrollo del software era un proceso personalizado; planeado y diseñado en la mente de alguien.
  - Se utilizaba el procesamiento por lotes.
- La segunda era (mediados de los 60's a finales de los 70's)
  - El software se considera un producto que se distribuye para macro y mini computadoras.
  - Inicia la industria del software con la idea de desarrollar el mejor paquete y así ganar mucho dinero.
  - La multiprogramación y los sistemas multiusuario introdujeron nuevos conceptos de interacción hombre-máquina.
  - Surgen los primeros sistemas de gestión de bases de datos y también los sistemas de tiempo real.
  - El mantenimiento del software comenzó a ser algo crítico.



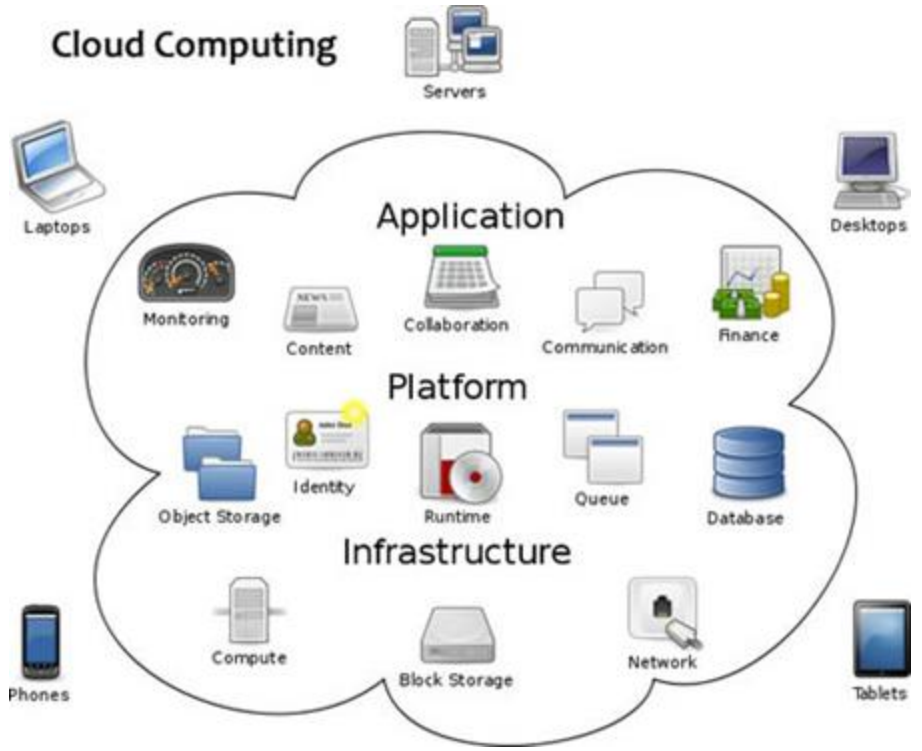


Evolución del software … (2)

- La tercera era (finales de los 70's principios de los 90's)
  - Crece considerablemente la presión sobre los desarrolladores de software.
  - Se incrementa notablemente la complejidad debido a los sistemas distribuidos.
  - Incrementa la demanda de acceso inmediato a los datos.
  - El uso personal del software aún no era común.
- La cuarta era (principios de los 90's … mediados de los 2000?)
  - La industria del software es considerada la cuna de la economía del mundo.
  - Dominan los sistemas cliente/servidor sobre los centralizados.
  - Tienen gran auge las tecnologías orientadas a objetos.
  - Irrumpe con fuerza el Internet y el comercio electrónico.
  - Sistemas de cómputo personales realmente potentes.
  - Las redes neuronales artificiales, cómputo paralelo, algoritmos genéticos y sistemas expertos salen de los laboratorios a aplicaciones prácticas.

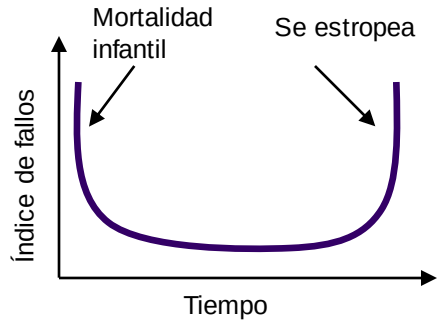
Evolución del software … (3)

- ¿En qué era se deberían ubicar los siguientes?
  - Cómputo ubicuo
  - Cómputo móvil
    - Teléfonos inteligentes
  - Cómputo en la nube
  - Cómputo GPU
  - Aplicaciones Web
  - Redes sociales

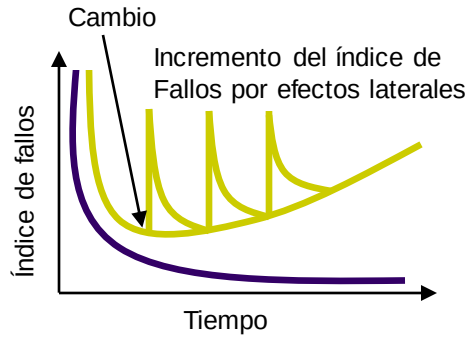


Características del software

- El software al ser un elemento lógico tiene ciertas características que lo diferencian claramente respecto al hardware [Pressman, 2002].
  - El software se desarrolla, no se fabrica en un sentido clásico.
    - El desarrollo y fabricación generan un producto pero desde enfoques diferentes.
  - El software no se estropea; pero se deteriora.
    - Los fallos del hardware se dan al principio y al final de su vida, mientras que en el software el mantenimiento dado a lo largo de su vida introduce nuevos fallos.
  - Aunque la industria tiende a ensamblar componentes, la mayoría del software se construye a la medida.
    - Esta situación esta cambiando con el uso más extendido de la programación orientada a objetos.



Curva de fallos del hardware



Curvas de fallos real e Idealizado del software

Dominios de aplicación del software

- Actualmente hay siete categorías de software [Pressman, 2010]
  1. Software de sistemas
     - Conjunto de programas para servir a otros programas.
     - En general tienen una fuerte interacción con el hardware, múltiples usuarios, operación concurrente, compartición de recursos, estructuras de datos complejas, entre otras.
     - Ejemplos: compiladores, editores, utilidades de gestión de archivos, controladores, software de redes, etc.
  2. Software de aplicación
     - Programas aislados que resuelven una necesidad específica de negocios.
     - Procesan datos comerciales o técnicos para facilitar las operaciones o toma de decisiones de negocios o técnicas.
     - Ejemplos: procesamiento de transacciones en puntos de venta, control de procesos de manufactura en tiempo real.
  3. Software de ingeniería y ciencias
     - Se ha caracterizado por “algoritmos devoradores de números”.
     - Las aplicaciones van desde la astronomía a la vulcanología, del análisis de tensiones en automóviles a la dinámica orbital de un transbordador espacial, de la biología molecular a la manufactura automatizada.

Dominios de aplicaciones del software … (2)

- Categorías del software …
  4. Software incrustado
     - Reside dentro de un producto o sistema y se usa para implementar y controlar funciones para el usuario final y para el sistema en sí.
     - Ejecuta funciones limitadas y particulares o provee una capacidad de funcionamiento y control.
     - Ejemplos: control del tablero de un microondas, funciones digitales en un automóvil como el control de combustible.
  5. Software de línea de productos
     - Es diseñado para proporcionar una capacidad específica para uso de muchos consumidores diferentes.
     - Se centra en un mercado particular o a mercados masivos.
     - Ejemplos: control de inventario de productos, procesadores de textos, hoja de cálculo, etc.
  6. Aplicaciones Web
     - Llamadas Webapps. Esta categoría de software centrado en redes agrupa una amplia gama de aplicaciones.
     - Van desde sencillas páginas dinámicas hasta ambientes de cómputo sofisticados con integración a bases de datos corporativas y aplicaciones de negocios.
  7. Software de inteligencia artificial
     - Hace uso de algoritmos no numéricos para resolver problemas complejos para los que no son adecuados los análisis directos.
     - Ejemplos: sistemas expertos o basados en conocimientos, reconocimiento de patrones, imágenes, voz, redes neuronales artificiales, etc.

La crisis del software

- La mayoría de los expertos están de acuerdo en que la manera más probable para que el mundo se destruya es por accidente. Ahí es donde nosotros entramos; somos profesionales de la informática, provocamos accidentes [Nathaniel Borenstein].
- Se trata más de una aflicción crónica que de una crisis.
  - Aflicción porque causa pena o desastre.
  - Crónica porque es duradero y reaparece con frecuencia.
  - No se le puede llamar del todo crisis porque no ha sido un punto decisivo en el curso de la “enfermedad”.
- Independientemente de que se le llame crisis o aflicción, se alude a un conjunto de problemas relacionados con el desarrollo de software.

Causas de la crisis del software

- Muchas de las causas de la crisis del software se pueden encontrar en una mitología que surge durante los primeros años del desarrollo del software.
- Se tienen diferentes tipos de mitos según los involucrados [Pressman, 2002]
  - De gestión
    - Los gestores de software normalmente están bajo la presión de cumplir los presupuestos, hacer que no se retrace el proyecto y mejorar la calidad; los mitos le crean la falsa ilusión de una menor presión.
  - Del cliente
    - En los clientes que solicitan una aplicación de software los mitos les crean falsas expectativas y finalmente quedan insatisfechos.
  - De los desarrolladores
    - A pesar de que algunos tiene 50 años, muchos de estos mitos aún están vigentes; hay quien sigue pensando que la programación es un arte.
  - De los ingenieros de software
    - Dentro del deseo de contar con prácticas más disciplinadas puede provocar querer definir los proyectos de construcción de software de manera similar a como se definen muchos proyectos de otras ingenierías [Rubby Casallas].

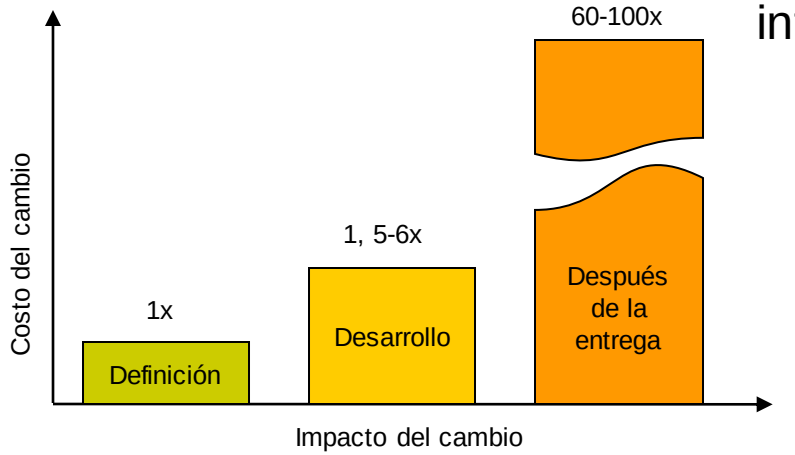
Causas de la crisis del software … (2)

- Mitos de gestión
  1. Tenemos ya un libro que está lleno de estándares y procedimientos para construir software.
  2. Mi gente dispone de las herramientas de desarrollo de software más avanzadas, después de todo, les compramos las computadoras más modernas.
  3. Si fallamos en la programación podemos añadir más programadores y adelantar el tiempo perdido
- Realidad
  1. Esta bien que el libro exista, pero ¿se usa? ¿se sabe de su existencia? ¿es completo?
  2. Más que computadoras actualizadas es más útil una herramienta CASE, aunque la mayoría de los desarrolladores no las utilizan eficazmente.
  3. El desarrollo de software no es un proceso mecánico y añadir más gente a un proyecto de software normalmente lo retraza aún más, sobretodo si no se ha planificado y coordinado correctamente.

Causas de la crisis del software … (3)

- Mitos del cliente
  1. Una declaración general de los objetivos es suficiente para comenzar a escribir los programas, los detalles los daremos más adelante.
  2. Los requisitos del software cambian constantemente, pero los cambios pueden adaptarse fácilmente, después de todo el software es flexible.

- Realidad
  1. Una mala definición inicial es la principal causa de trabajo infructuoso. Es esencial una descripción detallada y formal del sistema, que solo puede lograrse después de una exhaustiva comunicación con el cliente.
  2. Es verdad que los requisitos cambian, pero el impacto varía según el momento en que se introduzca



Causas de la crisis del software … (4)

- Mitos de los desarrolladores
  1. Una vez que escribimos el programa y hacemos que funcione, nuestro trabajo ha terminado.
  2. Hasta que no tengo el programa ejecutándose, realmente no tengo forma de comprobar su calidad.
  3. Lo único que se entrega al terminar el proyecto es el programa funcionando.

- Realidad
  1. Cuanto más pronto se comience a escribir código, más rápido tardará en terminarlo. Los datos industriales indican que entre el 60-80% del esfuerzo dedicado a un programa se realizará después de la primera entrega al cliente.
  2. Desde el principio del proyecto se pueden aplicar mecanismos para garantizar la calidad del software (revisión técnica formal) y detectar ciertos defectos.
  3. Un programa es solo una parte de una configuración de software que incluye muchos elementos. La documentación proporciona un buen fundamento para un buen desarrollo, además de guías para la importante tarea de mantenimiento.

Causas de la crisis del software … (5)

- Mitos de los ingenieros de software
  1. Se deben tener todos los requerimientos del sistema claramente definidos antes de empezar el proyecto para que éste sea exitoso.
  2. Diseñar completamente antes de empezar a programar para que sea más efectiva la labor de programación y más independiente.
  3. Para administrar en forma adecuada el proyecto, se debe tener un plan detallado de todo el proyecto desde el principio.

- Realidad
  1. Puede que se conozcan de manera global los objetivos que pretende el software, pero nunca se podrá tener el detalle de los requerimientos al inicio del proyecto. Es imposible que el cliente sepa detalladamente lo que quiere. Los desarrollos incrementales contribuyen en la solución del problema.
  2. Es imposible hacer un diseño completo y detallado para un conjunto de requerimientos incompletos y ambiguos. Cuando esto sucede se puede caer en el ciclo programar corregir. Las metodologías ágiles pueden contribuir a la solución del problema.
  3. Cuando no hay claridad en los requerimientos y no se puede tener completos los diseños, es una falacia pretender planificar en detalle el proyecto. El plan inicial debe contener los grandes hitos y estrategias y se debe detallar en permanentemente cada vez más en futuras iteraciones.

Causas de la crisis del software … (6)

- Y en la formación profesional de futuros desarrolladores de software?
  - También existen mitos y problemas a la hora de desarrollar proyectos:
    - Evaluaciones parciales o de fin de semestre
    - Estancias o servicio social
    - Tesis
  - Por lo regular se trata de problemas de exceso de confianza, planeación y poca disciplina
    - "Nada más encuentro como programar *esto* y ya prácticamente termine"
    - "Primero hago que *jale* y luego documento"
    - "Todavía tengo tiempo, luego desarrollo los otros módulos, mientras hago que este se vea bien *apantallador*"
    - "Si no encuentro cómo, tengo a mi cuate que puede *echarme* la mano"
    - "Buscando en la red seguro encuentro algo ya hecho que me sirva"
    - "Con que le *pegue* en las tardes pero bien y puede que termine antes de tiempo"
    - "No hay que preocuparse, en una noche sale"
    - "Con esto que desarrolle paso la materia, al fin y los demás tampoco van a terminar"
    - …. (la lista es vasta)

  - Agustín Cernuda del Río del Departamento de Informática de la Universidad de Ovideo cita algunos "errores clásicos" que influyen en la pérdida de control de un proyecto:
    - Expectativas poco realistas
    - Hazañas, ilusiones
    - Planificación excesivamente optimista
    - Gestión de riesgos insuficiente
    - Abandono de planeación bajo presión
    - Escatimar en las actividades iniciales y/o en el control de la calidad, a favor de la codificación
    - Programación a destajo
    - Exceso de requisitos
    - Control insuficiente

- Es importante y apropiado aprovechar los proyectos de evaluación parcial o de fin de semestre, los proyectos de estancia profesional o servicio social y los de tesis para que los futuros desarrolladores de software afronten los problemas y mitos relacionados con el desarrollo de software para fomentar una buena disciplina que les permita un desempeño profesional exitoso.

- Reflexión
  - Si no hacen análisis, hay tabla
  - Si no diseñan antes de programar, hay tabla
  - Si programan si hacer pruebas, hay tabla
  - Si no comentan el código, hay tabla
  - Si no emplean los estándares, hay tabla
  - Si cuelgan una aplicación por un ciclo infinito, hay tabla
  - Si reportan el mismo error dos veces, hay tabla
  - Si los casos de uso no están bien escritos, hay tabla
  - Si no planean, hay tabla
  - Si no gestionan proactivamente los riesgos, hay tabla
  - Si no versionan, hay tabla
  - Si preguntan por que todo esto, hay tabla

Consecuencias por fallas del software

- Se pueden clasificar en:
  - Consecuencias inmediatas y efectos directos
    - Son los perjuicios ocasionados mientras dura la caída de los sistemas.
    - En sistemas de misión crítica (sistema bancario) se generan perdidas realmente significativas.
    - Los costos de estos fallos son relativamente predecibles dado que dependen directamente del tiempo que dure la interrupción de la operación.
  - Consecuencias a mediano y largo plazo, y efectos indirectos
    - Son los perjuicios posteriores a la caída de los sistemas.
    - Las consecuencias varían, desde la restauración de los datos, propaganda negativa, pérdida de clientes hasta juicios en contra.
    - Es difícil de predecir el costo real a mediano y largo plazo.

Consecuencias por fallas del software … (2)

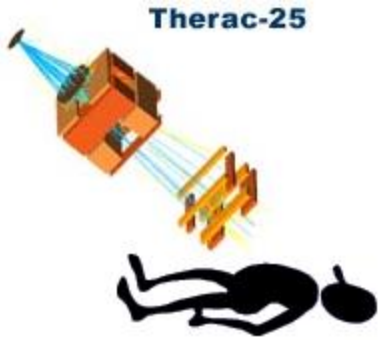
- Algunos casos de fallas en sistemas de software
  - Accidente de un F-18 (1986)
    - Giro descontrolado atribuido a un *if-then*, para la cual no había un *else*, por considerarlo innecesario.
  - Sobre costo en sistema de avión de carga C-17 (1989)
    - Costó más de 500 millones de dólares más de lo previsto debido a problemas de software. Se reportaron 19 computadoras abordo, 80 microprocesadores y seis lenguajes de programación diferentes.
  - Explosión del cohete Ariane 5 (1996)
    - Se salió de trayectoria por que el programa supuso que se había desviado al hacer la conversión de un número flotante de 64 bits a un entero de 16 bits.

Consecuencias por fallas del software … (3)

- Existen muchos casos documentados y no documentados de fallas provocadas por software o atribuidas a este

Error del Y2K afecto los datos climáticos en bruto de la NASA

1986 Sobregiro de 32 mil millones de dólares por un contador de 16 bits



1985-1987 El acelerador lineal médico diseñado emitió radiación sin control ocasionando la muerte de varios pacientes; la secuencia de comandos introducida por el operador provocaba un estado interno erróneo.

La caída del sistema en las elecciones de1988

Consecuencias por fallas del software … (4)

- Un caso más

Definiciones de ingeniería de software

- La ingeniería de 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 [Bauer, 1972].
- Ingeniería del software es la aplicación práctica del conocimiento científico en el diseño y construcción de programas de computadora y la documentación asociada requerida para desarrollar, operar y mantenerlos. Se conoce también como desarrollo de software o producción de software [Bohem, 1976].
- La ingeniería de software es el estudio de los principios y metodologías para desarrollo y mantenimiento de sistemas de software [Zelkovitz, 1978].
- Ingeniería de software: (1) La aplicación de un enfoque sistemático, disciplinado y cuantificable hacia el desarrollo, operación y mantenimiento del software; es decir, la aplicación la de la ingeniería al software. (2) El estudio de enfoques como en (1) [IEEE, 1993].
- La Ingeniería de Software es una disciplina de la Ingeniería que concierne a todos los aspectos de la producción de software [Sommerville, 1995].

Paradigmas de ciclos de vida de la ingeniería de software

- Cuando no se sigue un ciclo de vida y apenas se planea, se tiende a seguir el enfoque de “codificar y probar” lo que genera: una alta probabilidad de falla en el software, poca flexibilidad para modificaciones, no satisfacer plenamente los requisito y descontento de los clientes [Piattini].
- Qué es un ciclo de vida:
  - Un modelo de ciclo de vida es un marco de referencia que contiene los procesos, las actividades y las tareas involucradas en el desarrollo, la explotación y el mantenimiento de un producto de software, abarcando la vida del sistema desde la definición de los requisitos hasta la finalización de su uso [ISO/IEC 12207-1].
  - Ciclo de vida del software es una aproximación lógica a la adquisición, suministro, el desarrollo, la explotación y el mantenimiento del software [IEEE 1074].

Paradigmas de ciclos de vida de la ingeniería de software … (2)

- Algunas de las ventajas que aporta el enfoque de ciclo de vida [Piattini]:
  - En las primeras fases, aunque no haya líneas de código, pensar el diseño es avanzar en la construcción del sistema, pues posteriormente resulta más fácil la codificación.
  - Asegura un desarrollo progresivo, con controles sistemáticos, que permite detectar precozmente los defectos.
  - Se controla el sobrepasar los plazos de entrega y los costes excesivos mediante un adecuado seguimiento del progreso.
  - La documentación se realiza de manera formal y estandarizada simultáneamente al desarrollo, lo que facilita la comunicación interna entre el equipo de desarrollo y la de éste con los usuarios. También aumenta la visibilidad y posibilidad de control para la gestión del proyecto.
  - Supone una guía para el personal de desarrollo, marcando las tareas a realizar en cada momento.
  - Minimiza la necesidad de rehacer el trabajo y los problemas de puesta a punto.

Paradigmas de ciclos de vida de la ingeniería de software … (3)

- Actividades agrupadas en procesos que se pueden realizar durante el ciclo de vida del software [ISO 12207-1].

PROCESOS PRINCIPALES

- ADQUISICIÓN
- SUMINISTRO
- DESARROLLO
- EXPLOTACIÓN
- MANTENIMIENTO

PROCESOS DE SOPORTE

- DOCUMENTACIÓN
- GESTIÓN DE LA CONFIGURACIÓN
- ASEGURAMIENTO DE LA CALIDAD
- VERIFICACIÓN
- VALIDACIÓN
- REVISIÓN CONJUNTA
- AUDITORIA
- RESOLUCIÓN DE PROBLEMAS

PROCESOS DE LA ORGANIZACIÓN

- GESTIÓN
- INFRAESTRUCTURA
- MEJORA
- FORMACIÓN

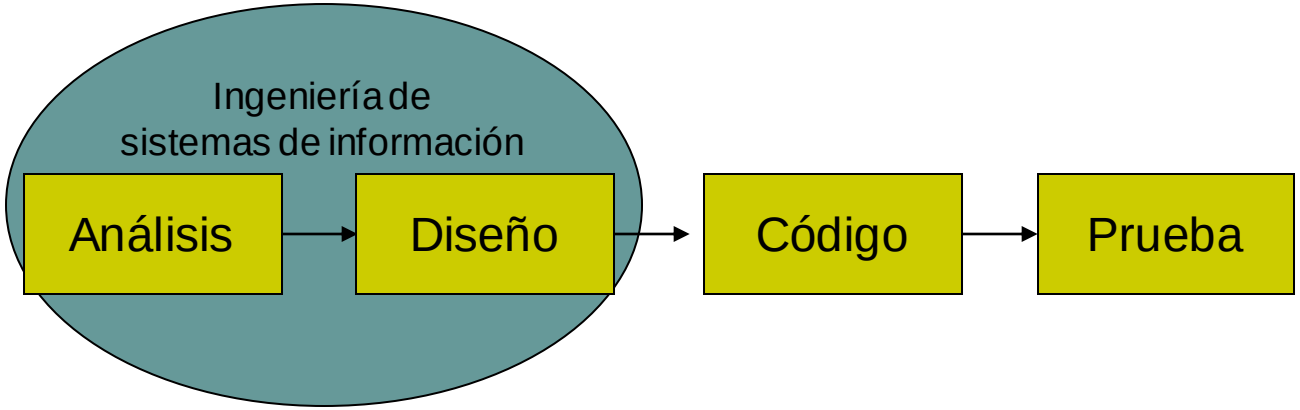
Investigar:

Fundamentos teóricos:

1. Ciclos de vida convencionales
    a) Modelo secuencial básico
    b) Modelo en Cascada.
    c) Modelo de construcción de prototipos
    d) Modelo Incremental
    e) Modelo espiral
2. Modelos Recientes
    a) Técnicas de cuarta generación (T4G)
    b) Modelo fuente.
    c) Desarrollo basado en componentes
    d) Proceso unificado (UP)
    e) Programación extrema XP

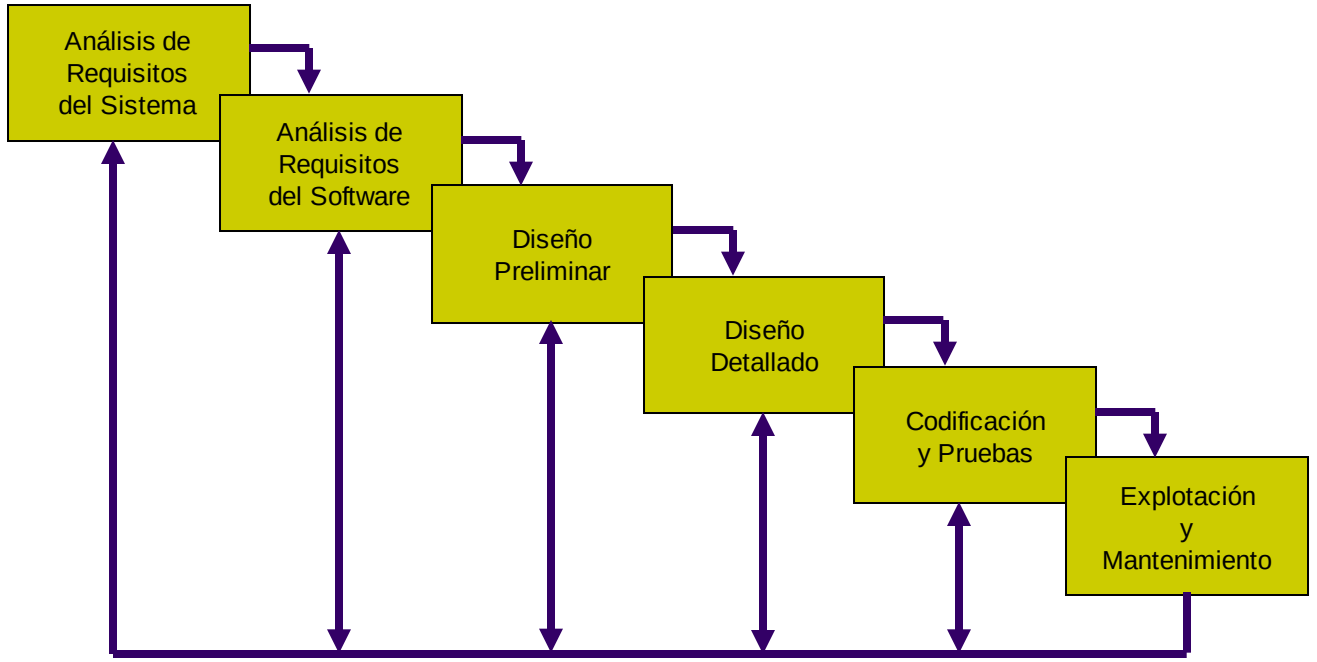
Ciclos de vida convencionales

- **Modelo secuencial (clásico)**
  - Análisis de los requisitos del software
    - Se debe comprender el dominio de la información, la función requerida, comportamiento, rendimiento e interconexión.
  - Diseño
    - Proceso de muchos pasos centrado en cuatro atributos: estructura de datos, arquitectura de software, representación de la interfaz y detalle procedimental.
  - Generación de código
    - El diseño es traducido a formato legible por la máquina.
  - Pruebas
    - Detección de errores y asegurar que una entrada definida produce resultados esperados.
  - Mantenimiento
    - Cambios en el software que implican aplicar todos los pasos precedentes en orden.



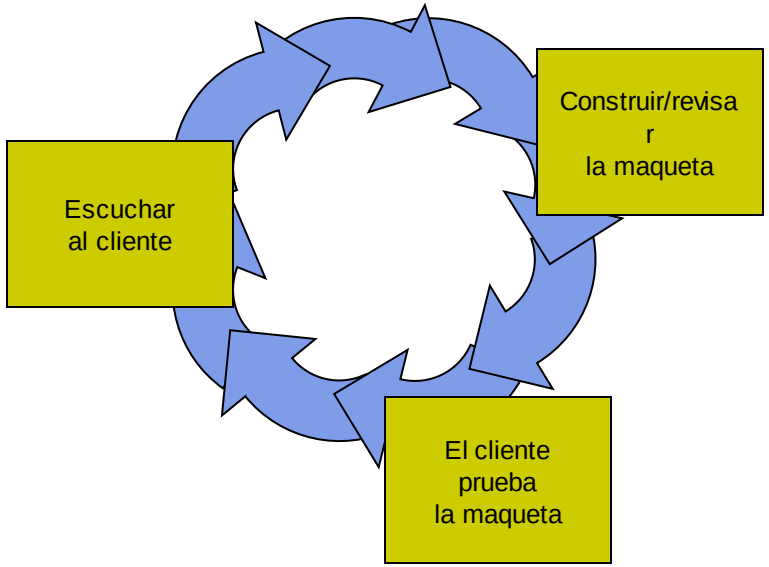
Ciclos de vida convencionales … (2)

- **Modelo en cascada**
  - Es una adaptación del modelo secuencial
  - Sin embargo el modelo secuencial y cascada han sido criticados en varios aspectos:
    - No refleja el proceso “real” de desarrollo de software, por ejemplo cuando hay redefinición de requisitos en la fase de codificación.
    - Se tarada mucho tiempo en pasar por todo el ciclo.
    - Acentúa los problemas con el usuario final



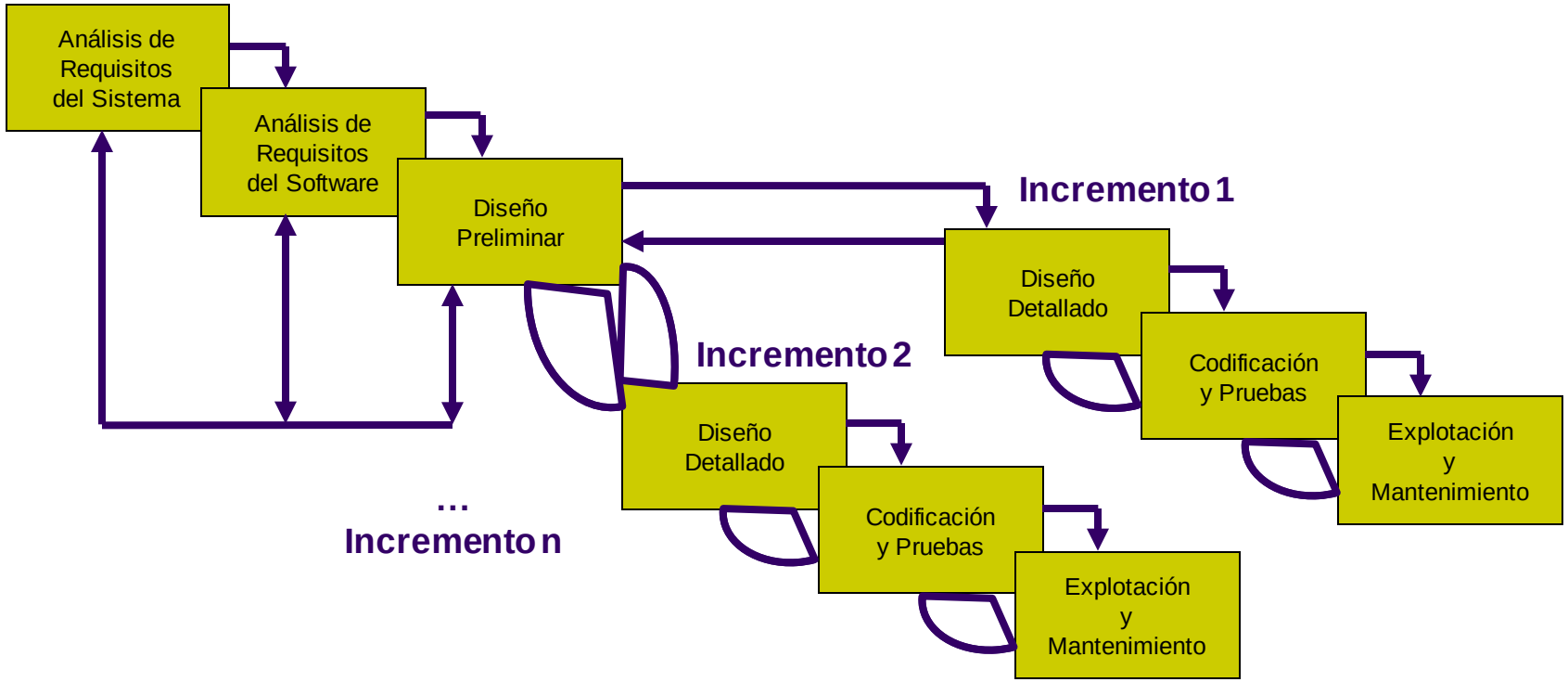
Ciclos de vida convencionales … (3)

- **Modelo de construcción de prototipos**
  - Paradigma:
    - Inicia con la recolección de requisitos
    - Se hace un diseño rápido de aspectos visibles para el cliente/usuario para generar un prototipo.
    - El prototipo lo evalúa el cliente/usuario para refinar los requisitos.
  - Pudiera presentar problemas debido a:
    - El cliente solo ve el sistema por “fuera” y no la calidad por “dentro”; el mantenimiento no es prioridad cuando se desarrolla rápido.
    - El desarrollador, con frecuencia, hace uso de las herramientas más a la mano no de las más apropiadas.



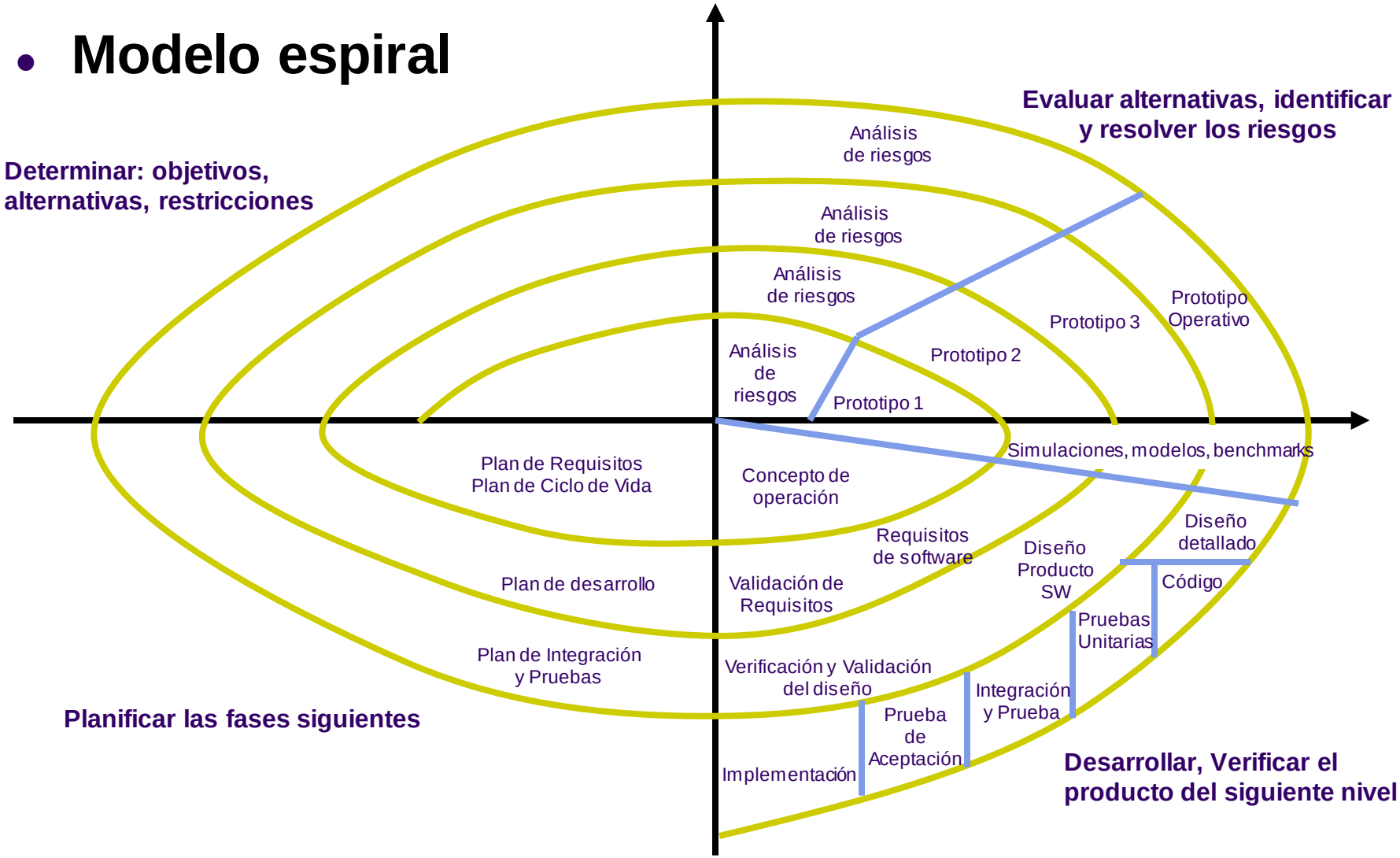
Ciclos de vida convencionales … (4)

- **Modelo incremental**
  - Combina elementos de los modelos secuencial, cascada y prototipos.
  - El sistema de software se crea añadiendo componentes funcionales, cada paso se actualiza el sistema hasta cumplir con los requisitos.
  - Este modelo se ajusta más a la incertidumbre del incremento de requisitos, sin embargo persiste el problema para determinar si los requisitos son válidos.



Ciclos de vida convencionales … (5)

- **Modelo espiral**

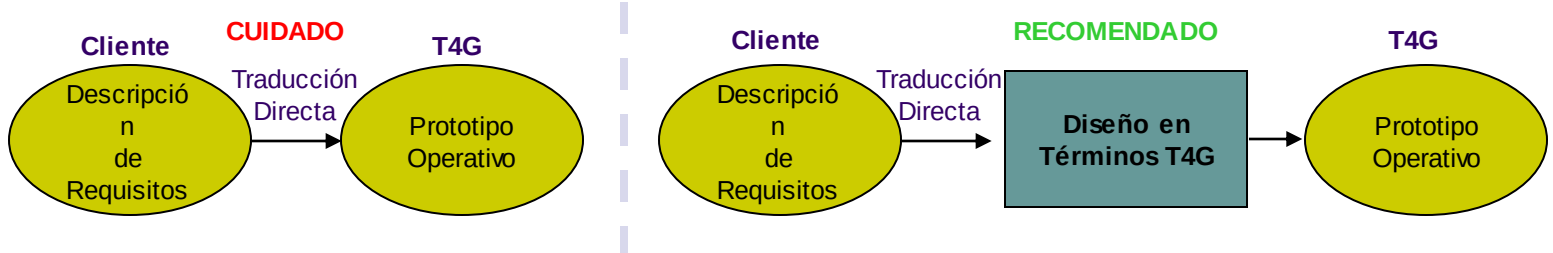


Ciclos de vida convencionales … (6)

- Modelo espiral
  - Paradigma
    - Combina los modelos secuencial y prototipos
    - Cada ciclo inicia con la identificación de objetivos (ej. Rendimientos, funcionalidad), alternativas principales de implementación de esa porción del producto (ej. Usar el diseño A, reutilizar el módulo X) y las restricciones para cada alternativa (ej. Interfaces, costo).
    - El siguiente paso es evaluar las diferentes alternativas en función de los objetivos y restricciones. Si existen riesgos se deben prevenir formulando una estrategia (prototipos, simulación).
    - Después se revisan los resultados del análisis de riesgos.
    - El siguiente paso consiste en planificar la fase posterior.
    - Una vez terminado el primer ciclo se volvería a empezar.
    - Cada ciclo se completa con una revisión de todos los productos generados durante el ciclo.
  - Este modelo también presenta ciertas dificultades:
    - Cuando se debe subcontratar no se tiene la misma flexibilidad para ajustarse a acuerdos por etapa, resolver caminos críticos, ajustar niveles de esfuerzo, etc.
    - Necesidad de contar con expertos en evaluación de riesgos para identificar y manejarlos.

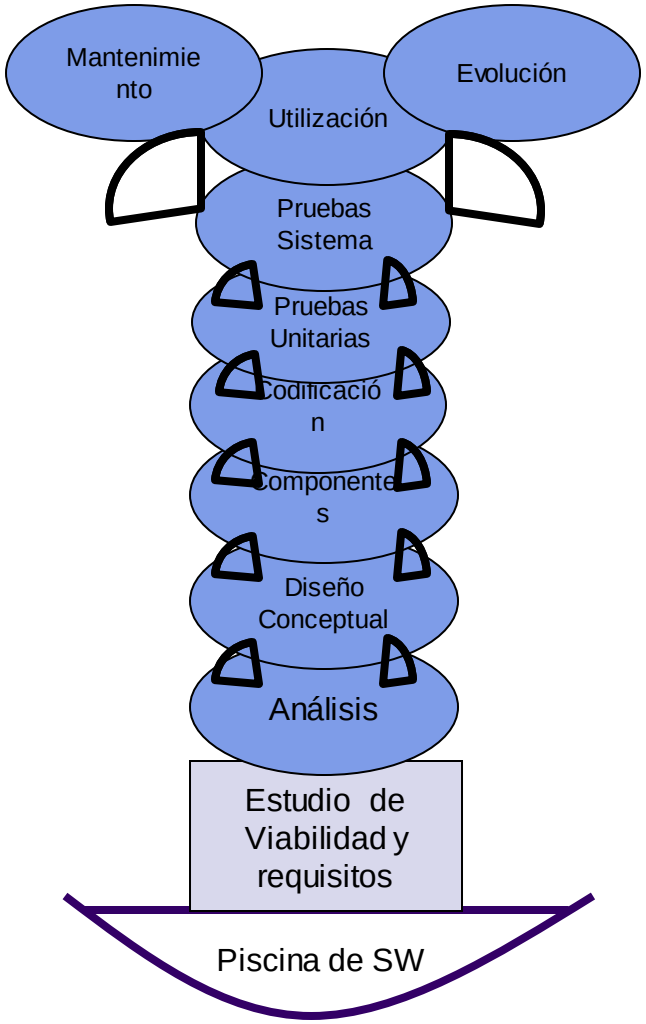
Modelos recientes

- **Técnicas de cuarta generación (T4G)**
  - Se orientan hacia la posibilidad de especificar el software utilizando formas de lenguaje especializado, o notaciones gráficas que describan el problema a resolver en los términos que el cliente entienda.
  - Las herramientas de 4G facilitan la especificación de algunas características de alto nivel del software; luego traducen directamente dichas especificaciones a código fuente.
  - Un entorno de desarrollo T4G debería tener herramientas como:
    - Lenguajes no procedimentales de consulta a bases de datos
    - Generación de informes
    - Manejo de datos
    - Interacción y definición de pantallas
    - Generación de códigos con capacidades gráficas de alto nivel
    - Capacidades de hoja de cálculo
  - Al igual que los modelos anteriores las T4G inician con la recolección de requisitos; idealmente la descripción de requisitos debería traducirse a un prototipo operativo. Sin embargo es importante hacer un mayor esfuerzo en la estrategia de diseño para evitar consecuencias como: poca calidad, mantenimiento pobre y poca aceptación por parte del cliente.
  - Se pueden identificar ventajas y desventajas de las T4G
    - Ventajas: (1) Reducción drástica del tiempo de desarrollo y (2) Mayor productividad en la gente que construye el software.
    - Desventajas: (1) Las T4G no son más fáciles de utilizar que los lenguajes de programación, (2) El código fuente producido generalmente es ineficiente y (3) El mantenimiento es cuestionable



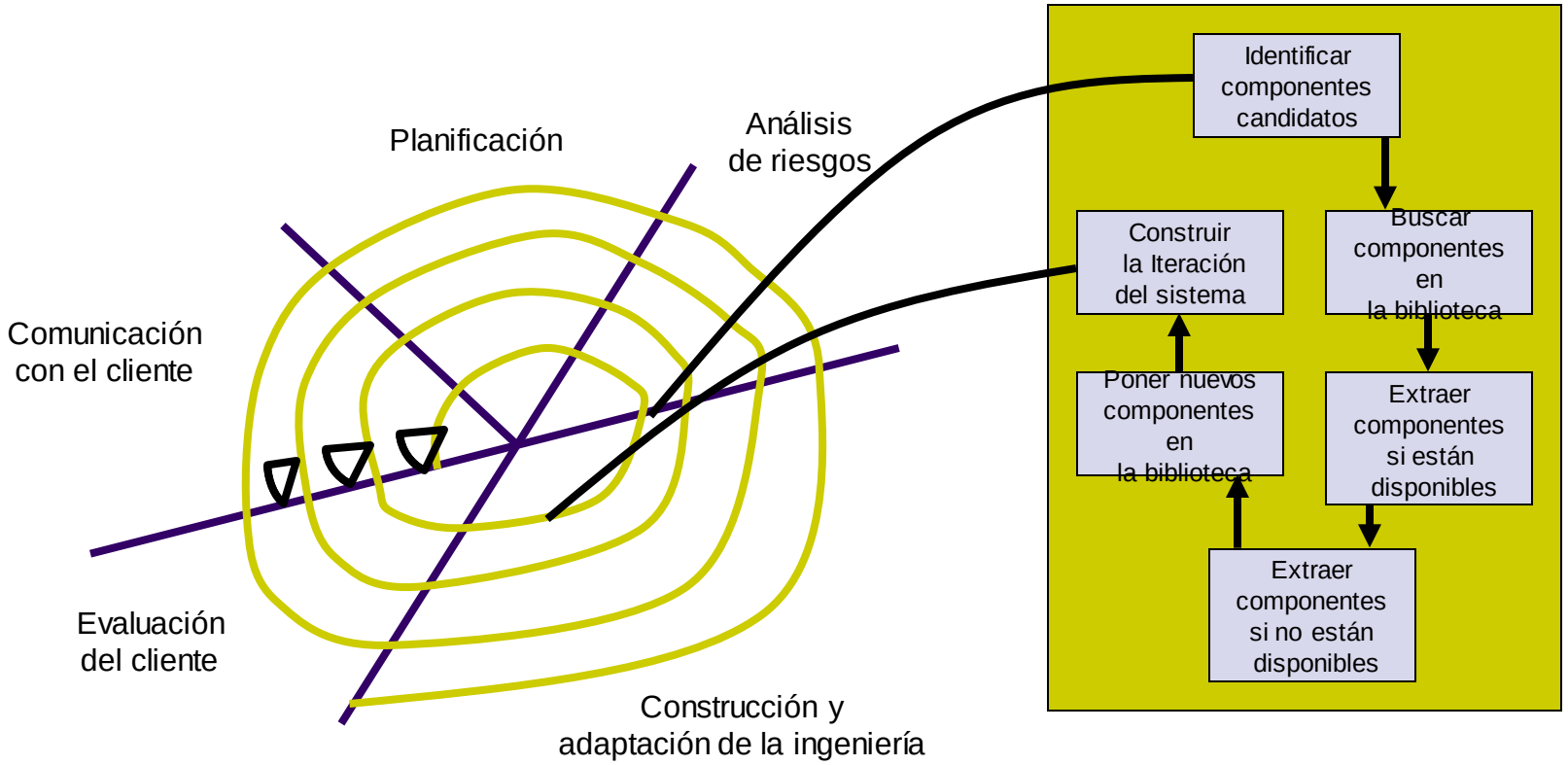
Modelos recientes … (1)

- **Modelo fuente**
  - Gráficamente representa un alto grado de iteración y solapamiento que hace posible la tecnología de objetos.
  - En la base está el análisis de requisitos, a partir del cual va creciendo el ciclo de vida, cayendo solo para el mantenimiento necesario.
  - Ese modelo también se propone para cada clase (o módulo), ya que cada una puede estar en una fase diferente del ciclo de vida durante el desarrollo del sistema



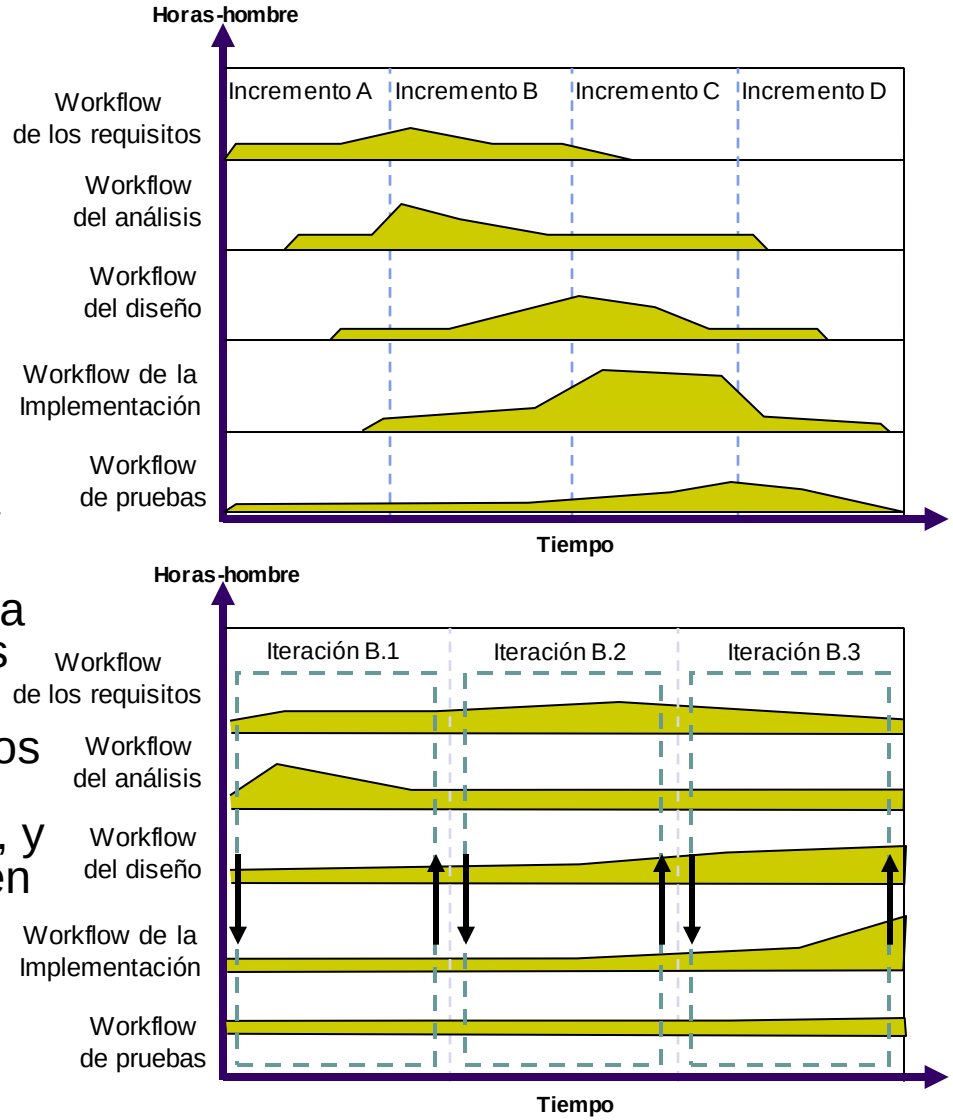
Modelos recientes … (2)

- **Desarrollo basado en componentes**
  - Incorpora muchas características del modelo espiral.
  - Configura aplicaciones desde componentes preparados de software (clases).
  - Conduce a la reutilización del software.



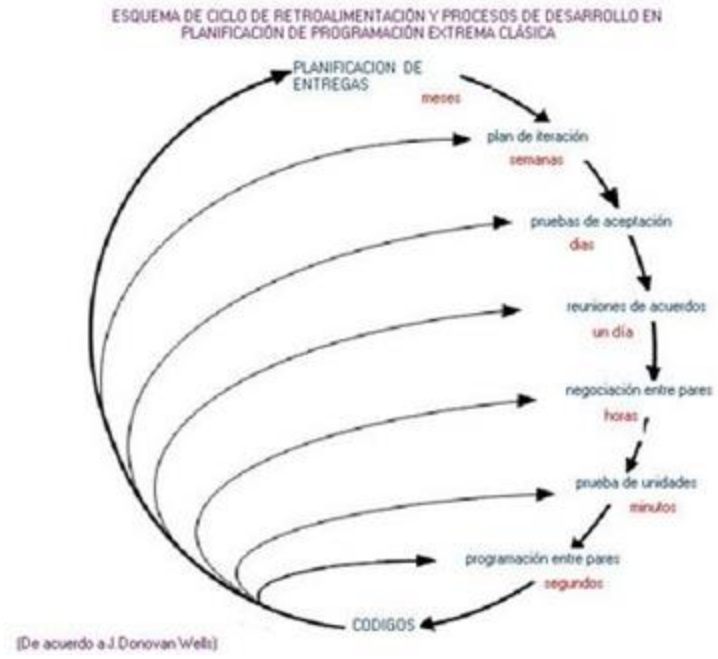
Modelos recientes… (3)

- **Proceso unificado (UP)**
  - Características principales:
    - Dirigido por casos de uso
    - Centrado en la arquitectura
    - Iterativo e incremental
  - Soporta el estándar UML
  - Se plantea un desarrollo por incrementos aplicando *workflows* (flujo de trabajo) de requisitos, análisis, diseño, implementación y pruebas, presentes a lo largo del todo el ciclo de vida, sin embargo a veces un *workflow* predomina más sobre los otros cuatro.
  - Las iteraciones se dan dentro de los incrementos, el número de estas varia dependiendo del incremento, y en cada iteración también se deben repetir los cinco *workflow*.



Modelos recientes… (4)

- **Programación extrema XP**
  - Disciplina de desarrollo de software creada por Kent Beck para proyectos cortos con requerimientos cambiantes o poco claros (respaldada por gran parte de la industria y rechazada por otro tanto).
  - Se basa en la simplicidad, la comunicación y el reciclado continúo de código.
  - Pretende:
    - La satisfacción del cliente
    - Potenciar al máximo el trabajo en grupo
    - Reducir el costo del cambio en las etapas de vida del sistema
    - Combinar las que han demostrado ser las mejores practicas de desarrollo de software y llevarlas al extremo.
  - Establece cuatro variables
    - Coste
    - Tiempo
    - Calidad
    - Ámbito
  - Plantea cuatro valores
    - Comunicación
    - Sencillez
    - Retroalimentación
    - Valentía
  - Define cuatro actividades básicas
    - Codificar
    - Hacer pruebas
    - Escuchar
    - Diseñar



Metodologías de desarrollo de software

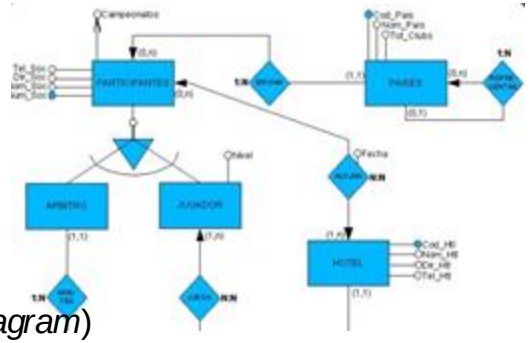
- Considerando una “metodología” como un conjunto de pasos y procedimientos que deben seguirse para desarrollar software, entre otras, se tienen:
  - Metodologías estructuradas
    - Proponen la creación de modelos del sistema que representen a los procesos, los flujos y las estructuras de los datos de una manera descendente.
  - Metodologías orientadas a objetos
    - Trata a los procesos y los datos de forma conjunta, agrupando así tanto la información como el procesamiento (objetos).
  - Metodologías para sistemas de tiempo real
    - La información se procesa, más orientada al control que a los datos, en función del tiempo.

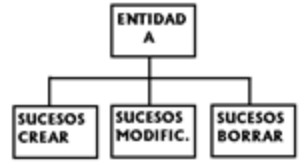
Metodologías estructuradas

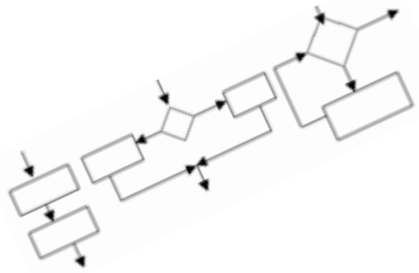
- Pasan de una visión general del problema (abstracción cercana a las personas) hasta llegar a un nivel de abstracción más sencillo (abstracción cercana al hardware).
- Esta visión se puede enfocar en las funciones del sistema, estructura de los datos o ambos, lo que da lugar a las siguientes metodologías:
  - Orientadas a los procesos
    - Se centra en la transformación de los datos de entrada para generar la salida esperada.
  - Orientadas a los datos
    - Estructuras de datos jerárquicas
      - Se centran en las entradas y salidas; primero se definen las estructuras de datos y, a partir de éstas, se derivan los componentes procedimentales.
    - Estructuras de datos no jerárquicas
      - Los tipos de datos son el corazón del sistema ya que son más estables que los procesos.
  - Mixtas
    - Se enfocan tanto en el proceso como en los datos tomando desde diversos puntos de vista.

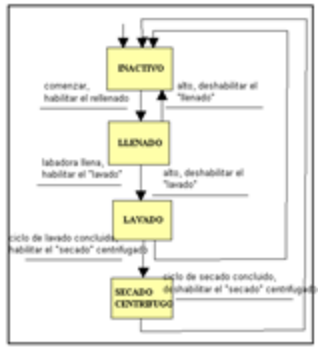
Metodologías estructuradas … (2)

- Existen diversas metodologías estructuradas:
  - Orientadas a procesos:
    - De Marco.
    - Gane y Sarson
    - Yourdon/Constantine
  - Orientadas a datos jerárquicos
    - JSP y JSD
    - LCP
  - Orientadas a datos no jerárquicos
    - IE
  - Metodologías mixtas
    - Merise
    - SSADM
    - Métrica
- Las especificaciones estructuradas utilizan:
  - DFD (Diagramas de flujo de datos, *Dataflow Diagram*)
  - Diagramas E-R (Entidad-Relación), o alternativamente DED (Diagramas de Estructura de Datos)
  - Diagramas HVE (Historia de vida de las entidades)
  - Diagramas de transición de estados (STD, *State Transition Diagram*)
  - Diccionario de datos
  - Especificación de procesos
    - Lenguaje estructurado
    - Pre y post condiciones
    - Tablas y árboles de decisión
  - Diagramas de estructura



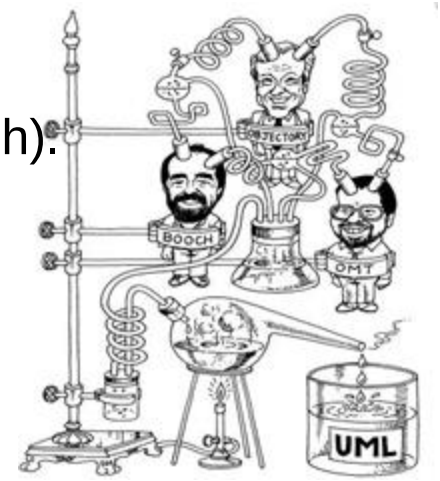






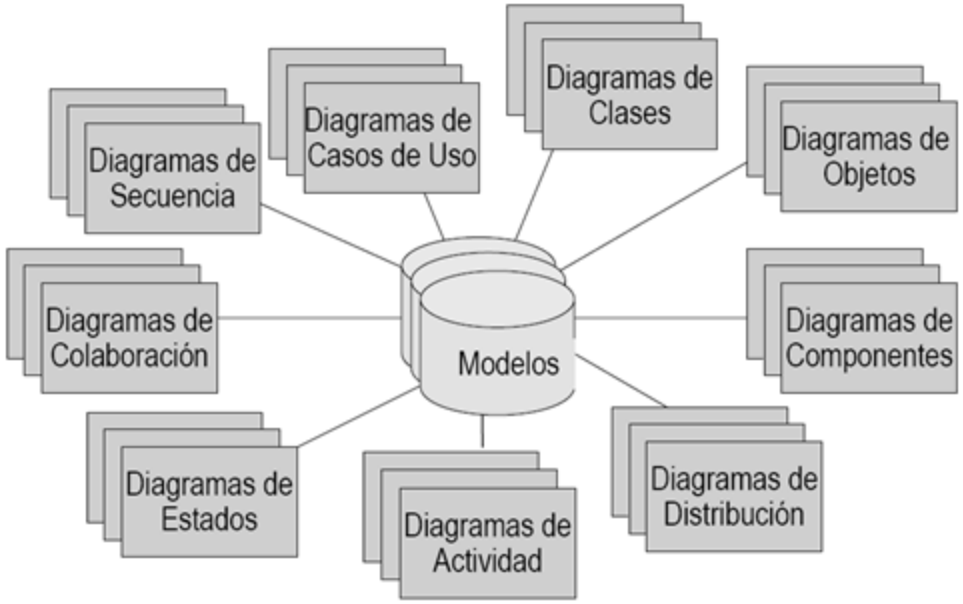
Metodologías orientadas a objetos

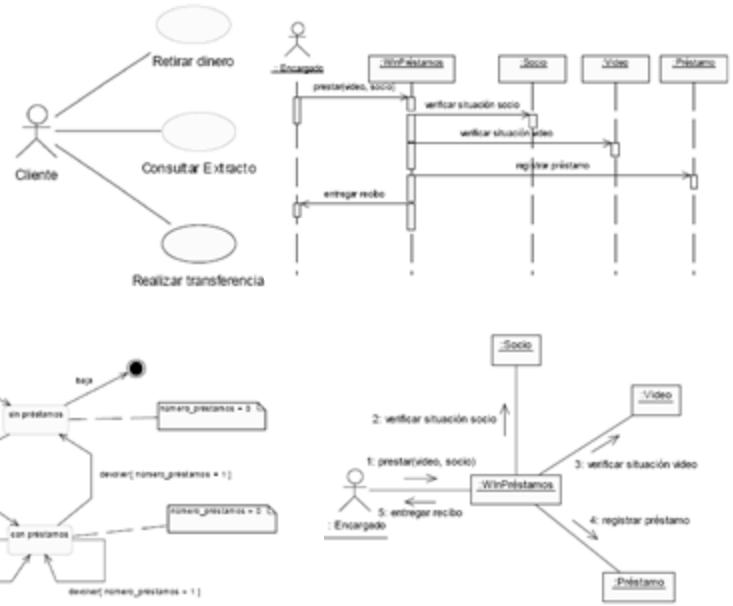
- De forma general:
  - El dominio del problema se caracteriza mediante un conjunto de objetos con atributos y comportamientos específicos.
  - Los objetos son manipulados mediante una colección de métodos y se comunican mediante un protocolo de mensaje.
  - Los objetos son clasificados en clases y subclases.
- Se retoman muchas de las ideas de las metodologías estructuradas pero con el apoyo de lenguajes orientados a objetos.
- En los 90's había diversos enfoques orientados a objetos:
  - Booch
  - Rumbaugh
  - Jacobson
  - Otros más (Shaler y Mellor, Coleman)
- En el 95 comienza el método unificado (Booch, Rumbaugh):
  - El mismo año se une Jacobson
  - Nace Rational Rose
  - De ahí surge UML aceptado por el OMG (*Object Management Group*) en el 97



Metodologías orientadas a objetos … (2)

- Actualmente las especificaciones orientadas a objetos utilizan el lenguaje estándar predominante **UML** (*Unified Modeling Lenguage*) el cual combina notaciones provenientes desde:
  - Modelado orientado a objetos
  - Modelado de datos
  - Modelado de componentes
  - Modelado de flujos de trabajo (*workflows*)
- Los diagramas que expresan gráficamente las partes de un modelo son:





Herramientas CASE

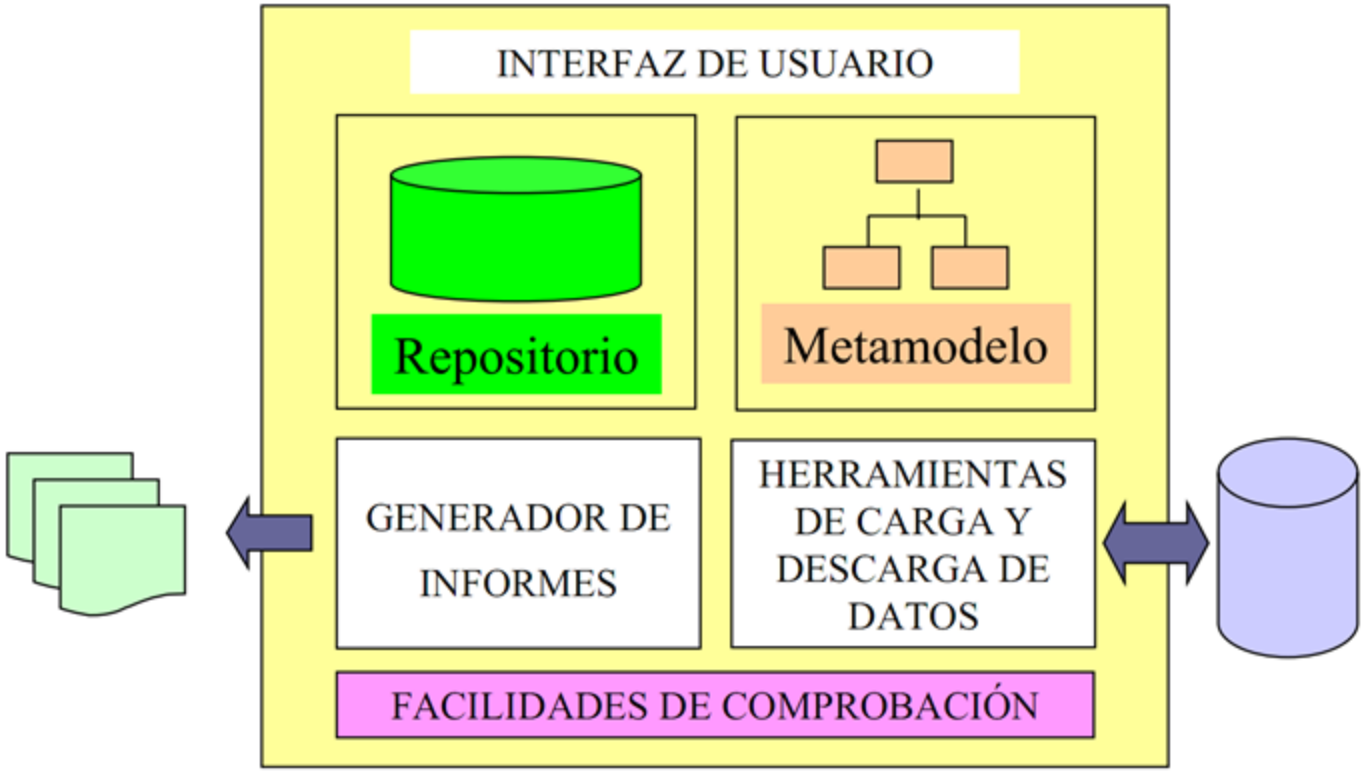
- CASE
  - ***C**omputer **A**ided **S**oftware **E**ngineering*
  - Ingeniería de software asistida por computadora
- Sistema de software que intenta proporcionar ayuda automatizada a las actividades del proceso de software en el marco de un ciclo de vida en particular [Wikipedia].
- Proporcionan la posibilidad de automatizar actividades manuales de la ingeniería de software ayudando a garantizar que la calidad sea diseñada antes de construir el producto [Pressman].

Herramientas CASE … (2)

- Este enfoque de construir software persigue mejorar la calidad y la productividad planteando los siguientes objetivos [Piattini]:
  - Aplicación práctica de metodologías de desarrollo de software
  - Facilitar la realización de prototipos y y el desarrollo conjunto de aplicaciones
  - Simplificar el mantenimiento de los programas
  - Mejorar y estandarizar la documentación
  - Aumentar la portabilidad de las aplicaciones
  - Facilitar la reutilización de componentes software
  - Desarrollo y refinamiento visual utilizando gráficos

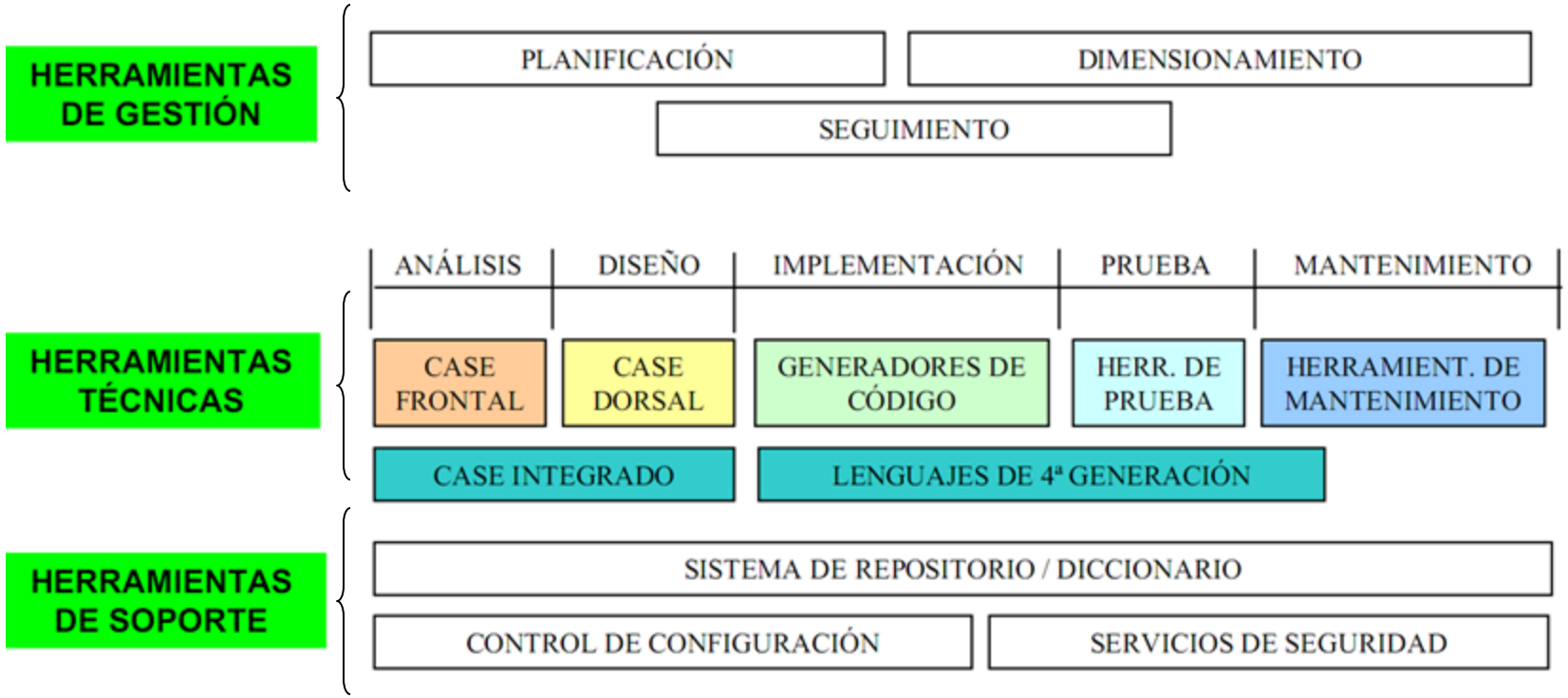
Herramientas CASE … (3)

- Componentes de una herramienta CASE



Herramientas CASE … (4)

- Categorías de herramientas CASE
  - Existen numerosas clasificaciones, Piattini propone la siguiente:

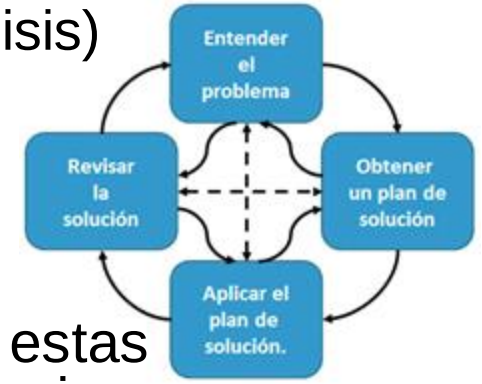


Herramientas CASE … (5)

- Inconvenientes por los cuales se han abandonado algunas herramientas CASE:
  - Deficiencias de la propia tecnología
    - Soporte parcial del ciclo de vida
    - Incompatibilidad entre herramientas, inclusive entre versiones propias
    - Escasa integración entre diferentes herramientas
    - Poca confianza entre vendedor/distribuidor
    - Escasa e inadecuado documentación generada
    - Gran abundancia de herramientas
    - Funcionamiento deficiente en entornos multiusuario
    - Poca capacidad de adaptación de los usuarios
    - Son costosas
  - Incorrecta implantación
    - Solo soportan una metodología (ej. Sistemas de gestión o de tiempo real)
    - No siempre soportan la técnica más adecuada
    - Funcionan o para proyectos pequeños o grandes
    - Se centran mucho en aspectos técnicos dejando de lado aspectos de gestión
  - Deficiencias de la propia organización
    - Actitud (falsas expectativas)
    - Infravaloración del esfuerzo requerido
    - Inconsistencia de herramientas respecto al nivel de madurez de la organización
    - Inadecuada capacitación de los usuarios
    - No medir la productividad ni la rentabilidad que resulten de la aplicación de la tecnología.

La práctica de la ingeniería de software

- How to Solve It [Polya, 1945]
  - Simple sentido común para resolver un problema. Sin embargo, el problema está en que el sentido común es poco común en el mundo del software [Pressman, 2010].
  - La esencia de la solución de problemas:
    1. Entender el problema (comunicación y análisis)
    2. Planear la solución (modelado y diseño del software)
    3. Ejecutar el plan (generación del código)
    4. Examinar la exactitud del resultado
  - En el contexto de la ingeniería de software, estas etapas de sentido común conducen a una serie de preguntas esenciales.



La práctica de la ingeniería de software … (2)

- Entender el problema
  - Escuchamos por unos segundos y después pensamos: claro, sí, entiendo, resolvamos esto. ¿Será esto correcto?, o valdrá la pena tomarse un poco más de tiempo a resolver estas preguntas:
    - ¿Quiénes tienen que ver con la solución del problema?, es decir, ¿Quiénes son los participantes?
    - ¿Cuáles son las incógnitas?, ¿Cuáles son las funciones y características que se requieren para resolver el problema en forma apropiada?
    - ¿Puede fraccionarse el problema?, ¿Es posible representarlo con problemas más pequeños que sean más fáciles de entender?
    - ¿Es posible representar gráficamente el problema?, ¿Puede crearse un modelo del análisis?

La práctica de la ingeniería de software … (3)

- Planear la solución
  - Después de entender el problema (o al menos eso se supone) y ya queremos escribir código. “La cosa es calmada”, antes habrá que hacer un diseño.
    - ¿Ha visto antes problemas similares?, ¿Hay patrones reconocibles en una solución potencial?, ¿Hay algún software existente que implemente los datos, las funciones y características que se requieren?
    - ¿Ha resuelto un problema similar? Si es así, ¿son reutilizables los elementos de la solución?
    - ¿Pueden definirse problemas más pequeños? Si así fuera, ¿hay soluciones evidentes para éstos?
    - ¿Es capaz de representar una solución en una forma que lleve a su implementación eficaz?, ¿Es posible crear un modelo del diseño?

La práctica de la ingeniería de software … (4)

- Ejecutar el plan
  - El diseño creado sirve como un mapa de carreteras para el sistema que se quiere construir. Puede haber desviaciones inesperadas y puede que encontremos un camino mejor a medida que avanza, pero el “plan” nos permitirá proceder sin perdernos.
    - ¿Se ajusta la solución al plan?, ¿El código fuente puede apegarse al modelo del diseño?
    - ¿Es posible que cada parte componente sea una solución correcta?, ¿El diseño y código se han revisado, o mejor aún, se han hecho pruebas respecto a la corrección del algoritmo?

La práctica de la ingeniería de software … (5)

- Examinar el resultado
  - No se puede estar seguro de que la solución sea perfecta, pero si de que se ha diseñado un número suficiente de pruebas para descubrir tantos errores como sea posible.
    - ¿Puede probarse cada parte componente de la solución?, ¿Se ha implementado una estrategia razonable para hacer pruebas?
    - ¿La solución produce resultados que se apegan a los datos, funciones y características que se requieren?, ¿El software se ha validado contra todos los requerimientos participantes?