

Metodología de Sistemas I

Tecnicatura Universitaria en Programación

Profesor: Leandro Schmidt

Ayudante: Benjamin Wagner

PDF by: Raphael Nicaise

- Conocer las distintas etapas en el ciclo de vida de los proyectos.
- Descubrir los distintos modelos evolutivos: Incremental y Espiral.
- Incorporar habilidades para captura y clasificación de requerimientos funcionales.
- Conocer herramientas útiles para el modelado de datos y sus relaciones.
- Entender la importancia de los procesos de calidad asociados al desarrollo de SW.
- Explorar las diferentes metodologías de gestión de proyectos, identificando ventajas y desventajas de su uso en diferentes escenarios.

Objetivos del cursado de la materia

- Otorgar al alumno la habilidad, destreza, aptitudes y técnicas para analizar y diseñar soluciones computacionales, valiéndose de metodologías de trabajo que le permitirán gestionar proyectos de pequeña, mediana o gran escala.
- El alumno debe ser capaz de comprender ventajas y desventajas de cada metodología de análisis y diseño de sistemas, sabiendo interpretar los modelos resultantes.
- Elaborar estrategias para la programación de sistemas, aplicando las adecuadas arquitecturas y tecnologías para resolver cada problema.
- Incorporar el concepto de ciclo de vida de los proyectos.

Estrategia Metodológica

La metodología de dictado consiste en:

- Desarrollo de clases teórico – prácticas
- Aprendizaje basado en ejemplos de aplicación cotidiana.
- Resolución de ejercicios con guías de Trabajos Prácticos (no entregables) por unidad temática para su resolución tanto grupal como individual.

Criterios de Evaluación: Aprobación Directa

Condiciones para la aprobación directa del cursado:

- Aprobación del único parcial teórico-práctico con **nota igual o superior a 8 (ocho)**.
- Alternativamente, y siempre y cuando el estudiante haya cursado la materia, podrá entregar un **proyecto final** para acceder a la aprobación directa.

Cursado de la Materia

Condiciones para el cursado de la materia:

- Aprobación del único parcial (en instancia inicial o recuperatorio) con nota igual o superior a 6 (seis)

Guia de Fechas Importantes

- Jueves 20/03: inicio de clases
- Jueves 12/06: parcial teórico-práctico
- Jueves 19/06: recuperatorio primer parcial
- Jueves 26/06: fin de clases

Bibliografía (Opcional)

Título	Autor	Editorial	Año
Ingeniería del Software: Un Enfoque Práctico". 7º edición.	PRESSMAN, Roger S.	Mc Graw Hill.	2010
Análisis Estructurado Moderno	Edward Yourdon	Prentice Hall	
El Proceso Unificado de Desarrollo de Software	RUMBAUGH, James; JACOBSON, Ivar; BOCH, Grady	Addison Wesley	2000
El Lenguaje Unificado de Modelado.	RUMBAUGH, James; JACOBSON, Ivar; BOCH, Grady	Addison Wesley	2000

Unidad 1: Software e Ingeniería del Software

¿Qué es el Software y la Ingeniería del Software?

- El software de computadora es el producto que construyen y mantienen los programadores profesionales (Pressman).
- Incluye programas se ejecutan en un dispositivo de cualquier tamaño y arquitectura.
- La ingeniería de software está formada por un proceso, un conjunto de métodos (prácticas) y herramientas que permiten a los profesionales elaborar software de alta calidad.

¿Por qué es importante?

- El software es importante porque afecta a casi todos los aspectos de nuestras vidas.
- Ha invadido nuestro comercio, cultura y actividades cotidianas.
- La ingeniería de software es importante porque nos permite construir sistemas complejos en un tiempo razonable y con alta calidad.

Algunas preguntas que puede responder la Ingeniería del Software:

- ¿Por qué se requiere tanto tiempo para terminar el software?
- ¿Por qué son tan altos los costos de desarrollo?
- ¿Por qué no podemos detectar todos los errores antes de entregar el software a nuestros clientes?
- ¿Por qué dedicamos tanto tiempo y esfuerzo a mantener los programas existentes?
- ¿Por qué seguimos con dificultades para medir el avance mientras se desarrolla y mantiene el software?

¿Cuáles son los pasos?

El software se construye del mismo modo que cualquier producto exitoso, pero siguiendo el enfoque de la ingeniería de software:

- Aplicación de un proceso ágil y adaptable
- Ciclos evolutivos del producto
- Búsqueda de un resultado de mucha calidad
- Se busca satisfacer las necesidades de las personas que usarán el producto.

¿Qué particularidades tiene el Software respecto a otros productos?

- El software es un elemento de un sistema lógico, no físico.
- Se desarrolla o modifica con intelecto; no se manufactura en el sentido clásico.
- Requiere de menor infraestructura para su “fabricación”.
- No se desgasta como el hardware. Pero con el tiempo se *deteriora, se ralentiza o simplemente se vuelve obsoleto*.

● ¿Cuál es el producto final?

Desde el punto de vista de un ingeniero de software, el producto final es el conjunto de programas, contenido (datos) y otros productos terminados que constituyen el software (el *entregable*).

- Desde la perspectiva del usuario, el producto final es la información resultante que de algún modo hace mejor al mundo en el que vive.

Unidad 1: Dominios de Aplicación del Software

Siete grandes categorías de software:

Software de sistemas: conjunto de programas escritos para dar servicio a otros programas. Se caracterizan por su gran interacción con el hardware de la computadora, uso intensivo por parte de usuarios múltiples, operación concurrente que requiere la secuenciación, recursos compartidos, estructuras complejas de datos e interfaces externas múltiples.

Ejemplos: compiladores, editores, herramientas para administración de archivos, componentes de sistemas operativos, *drivers*, software de redes, procesadores de telecomunicaciones, etc)

Software de aplicación: programas o aplicaciones que resuelven una necesidad específica de negocios, procesando datos comerciales o técnicos para facilitar las operaciones o la toma de decisiones administrativas o técnicas. Además, el software de aplicación se usa para controlar funciones de negocios en tiempo real (por ejemplo, procesamiento de transacciones en punto de venta, control de procesos de manufactura en tiempo real).

Ejemplos: sistema de facturación, control de acceso a edificio, etc.

Software de Ingeniería y Ciencias: históricamente relacionado a los algoritmos “devoradores de números”, las aplicaciones modernas dentro del área de la ingeniería y las ciencias están abandonando los algoritmos numéricos convencionales, dando paso al diseño asistido por computadora, la simulación de sistemas y otras aplicaciones interactivas.

Ejemplos: desde la astronomía, pasando por el análisis de tensiones en automóviles, a la dinámica orbital del transbordador espacial, y de la biología molecular a la manufactura automatizada.

Software Incrustado: reside dentro de un producto o sistema y se usa para implementar y controlar características y funciones para el usuario final y para el sistema en sí. El software incrustado ejecuta funciones limitadas y particulares (por ejemplo, control del tablero de un horno de microondas) o provee una capacidad significativa de funcionamiento y control (funciones digitales en un automóvil, como el control del combustible, del tablero de control y de los sistemas de frenado).

Ejemplos: software en electrodomésticos, sistemas de video-vigilancia, o los que utiliza la industria automotriz.

Software de Línea de Productos: está diseñado para proporcionar una capacidad específica para uso de muchos consumidores diferentes. El software de línea de productos se centra en algún mercado limitado y particular o se dirige a mercados masivos de consumidores (multimedios, entretenimiento, administración de base de datos y aplicaciones para finanzas personales o de negocios).

Se lo conoce también como “*software enlatado*”

Ejemplos: editor de imágenes, control del inventario de productos, hojas de cálculo, etc.

Aplicaciones Web / Mobile: esta categoría de software centrado en redes agrupa una amplia gama de aplicaciones, desde las webapps con contenido estático, pasando por otras que ofrecen funciones de cómputo y contenido para el usuario final, integradas con bases de datos corporativas y aplicaciones de negocios. La gran mayoría presenta los siguientes atributos: uso **intensivo de redes, concurrencia, carga impredecible, exigencias de alto rendimiento y disponibilidad**, están orientadas a datos, pueden contener información sensible y requerir de políticas específicas en cuanto a la seguridad. Su **estética es importante** y están pensadas dentro de un marco de evolución continua .

Software de Inteligencia Artificial: hace uso de algoritmos no numéricos para resolver problemas complejos que no son fáciles de tratar computacionalmente o con el análisis directo. Las aplicaciones en esta área son cada vez más comunes en la industria del software, y se vinculan con conceptos como *machine learning* y *data scientist*.

Ejemplos: robótica, sistemas expertos, reconocimiento de patrones (imagen y voz), redes neurales artificiales, demostración de teoremas y juegos.

A este grupo pertenecen las aplicaciones que actualmente vemos agrupadas en lo que se llama “IA Generativa”

Ejercicio mental: si intento automatizar el encendido y apagado de las luces de mi casa, con la posibilidad de forzar ese encendido / apagado incluso desde un App, utilizando algún hardware tipo Arduino y mis conocimientos de diseño y programación de interfaces, en cuál de los dominios anteriores podemos clasificar a nuestro “producto”? Y si el sistema pudiera ajustar el horario en base a mi rutina de retorno a casa?

- Inteligencia Artificial?
- Aplicación Mobile?
- Software Incrustado?

Unidad 1: El Proceso del Software

La ingeniería de software es una tecnología con varias capas:

- Herramientas: proporcionan apoyo a los métodos (automatizado o semi)
- Métodos: conjunto de tareas que proporcionan la experiencia técnica
- Proceso: propone el contexto para la aplicación de los métodos técnicos
- QA: es lo que permite pensar luego en mejora continua



- **Un proceso** es un conjunto de actividades, acciones y tareas que se ejecutan cuando va a crearse algún producto del trabajo.
- **Una actividad** busca lograr un objetivo amplio y se desarrolla sin importar el dominio de la aplicación, tamaño del proyecto, esfuerzo o grado de rigor con el que se usará la ingeniería de software. Ejemplo: comunicación entre los participantes
- **Una acción** es un conjunto de tareas que producen un producto importante del trabajo (por ejemplo, un modelo del diseño de la arquitectura).
- **Una tarea** se centra en un objetivo pequeño pero bien definido (por ejemplo, realizar una prueba unitaria) que produce un resultado tangible

Una estructura de proceso general para la ingeniería de software consta de cinco actividades estructurales:

1. Comunicación
2. Planeación
3. Modelado
4. Construcción
5. Despliegue



También existen *actividades sombrilla* que acompañan a las actividades estructurales:

- Seguimiento y control del proyecto de software.
- Administración del riesgo
- Aseguramiento de la calidad del software
- Revisiones técnicas
- Preparación y producción del producto del trabajo
- Mediciones
- Administración de la configuración del software
- Administración de la reutilización

Unidad 1: La Práctica de la Ingeniería del Software

La esencia de la práctica de la ingeniería de software podemos entenderla como:

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 (**probar y asegurar la calidad**).

Principios generales de la Ingeniería de Software:

- Primer principio: la razón de que exista todo (el valor que aporta)
- Segundo principio: mantenerlo simple
- Tercer principio: mantener la visión
- Cuarto principio: otro consumirá lo que usted produce
- Quinto principio: abrace al futuro
- Sexto principio: planee por anticipado la reutilización
- Séptimo principio: ¡piense!

Unidad 2: Modelos de Procesos

¿Qué es el proceso de Software?

- El desarrollo de software es un proceso de aprendizaje social. El proceso genera interacción entre usuarios y diseñadores, entre usuarios y herramientas cambiantes, y entre diseñadores y tecnología.
- El proceso del software se define como una estructura para las actividades, acciones y tareas que se requieren a fin de construir software de alta calidad.
- Es una forma organizada de desarrollar software. Es como una receta que indica qué pasos seguir para construir un programa de manera eficiente y sin caos.

¿Diferencia entre Proceso e Ingeniería?

- Un proceso del software define el enfoque adoptado mientras se hace ingeniería sobre el software, y ésta incluye tecnologías que pueblan el proceso, como métodos técnicos y herramientas automatizadas.
- La ingeniería de software intenta adaptar un proceso maduro de software a fin de que resulte apropiado para los productos que construyen y para las demandas de su mercado.

Ejemplo - Hagamos una pizza 🍕. Hay varias formas de prepararla:

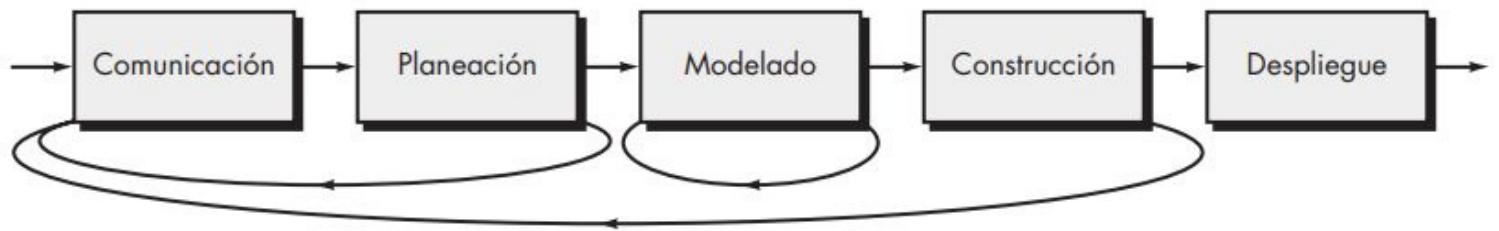
- Seguimos una receta clásica paso a paso (como el modelo en cascada).
- Probamos la masa, ajustando los ingredientes y mejorando (como el modelo ágil).
- Hacemos una versión rápida para probarla antes de la definitiva (como el modelo de prototipos).

Unidad 2: Flujos del Proceso:

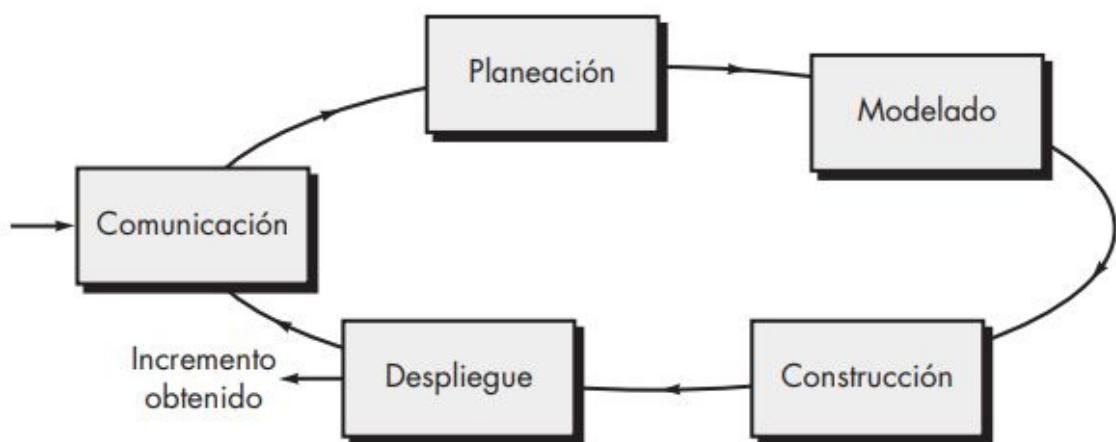
Flujo de Proceso Lineal



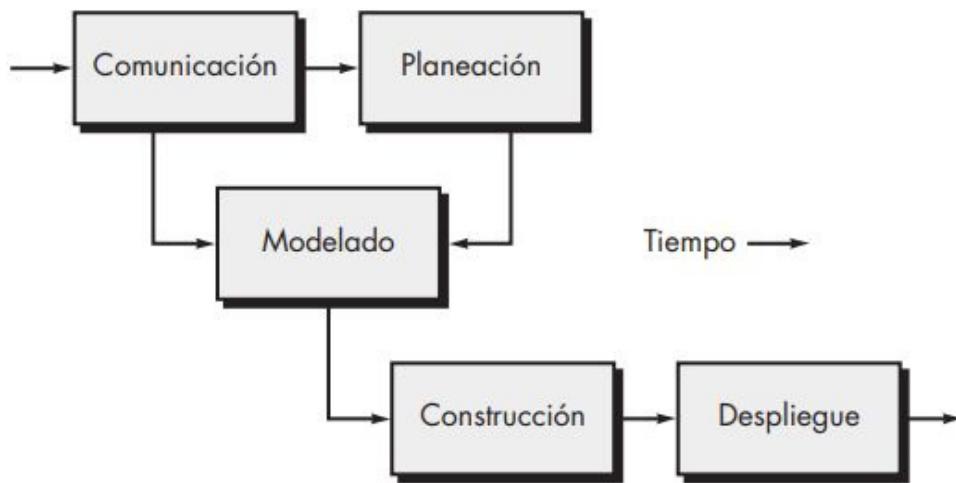
Flujo de Proceso Iterativo



Flujo de Proceso Evolutivo



Flujo de Proceso Paralelo



Unidad 2: Modelo Evolutivo vs Iterativo

Modelo Iterativo

- ✓ Se desarrolla en ciclos o iteraciones.
- ✓ Cada iteración es una versión mejorada del sistema, agregando más funcionalidades.
- ✓ Se parte de un diseño inicial y se va refinando en cada ciclo.
- ✓ Ejemplo: Modelo en espiral, RUP (Rational Unified Process).

Ejemplo simple:

Imagina que estás diseñando una aplicación de notas. En la primera iteración, solo puedes escribir y guardar notas. En la segunda, agregas colores. En la tercera, permites compartir notas.

Modelo Evolutivo

- ✓ Se desarrolla un prototipo inicial rápido que funciona, aunque sea básico.
- ✓ Se mejoran características basadas en la retroalimentación de los usuarios.
- ✓ No hay un diseño totalmente definido al inicio; el software cambia con el tiempo.
- ✓ Ejemplo: Modelo de Prototipos, Desarrollo Ágil.

Ejemplo simple:

Supongamos que creas una página web para una tienda online. Primero, solo tiene una lista de productos. Luego, te piden un carrito de compras y lo agregas. Más tarde, quieren pagos con tarjeta, y lo implementas.

Diferencia clave:

Iterativo: Se planifica desde el inicio, pero se construye en pasos.

Evolutivo: Se empieza con algo simple y se mejora en función de las necesidades reales.

En resumen, el modelo iterativo mejora un diseño predefinido en ciclos, mientras que el evolutivo parte de algo básico y lo transforma continuamente.

Unidad 2: Patrones de Procesos

¿Qué es un patrón de proceso?

Un patrón del proceso describe un problema relacionado con el proceso que se encuentra durante el trabajo de ingeniería de software, identifica el ambiente en el que surge el problema y sugiere una o más soluciones para el mismo.

Al combinar patrones, un equipo de software resuelve problemas y construye el proceso que mejor satisface las necesidades de un proyecto.

Un Patrón de Proceso es una solución **probada y reutilizable** para un problema común en el desarrollo de software. Así como en la construcción hay planos que guían la construcción, en software hay patrones de proceso que guían la forma en que se trabaja.

 **Ejemplo:** Imaginemos que soy un chef y tengo que cocinar un plato. Hay varias formas de organizarte:

- Hacer **todos los ingredientes primero** y luego cocinarlos.
- **Preparar y cocinar por partes**, en pequeñas tandas.
- **Probar el plato mientras lo haces** para ajustarlo.

Cada uno de estos enfoques es un **patrón de proceso**, porque establece una forma eficiente de trabajar en la cocina.

¿Por qué son útiles los Patrones de Proceso?

-  Ayudan a evitar errores comunes.
-  Mejoran la eficiencia del equipo.
-  Permiten reutilizar estrategias exitosas en distintos proyectos.

En resumen: Un patrón de proceso es como una receta para desarrollar software de manera organizada y efectiva. ¡Elegir el adecuado hace toda la diferencia! 😊

Un patrón se identifica por:

- Nombre
- Fuerzas que intervienen (ambiente / aspectos visibles del problema)
- Tipo (Etapa, Tarea o Fase)
- Contexto Inicial
- Problema a Resolver
- Solución Planteada
- Contexto Resultante
- Patrones Relacionados

EJEMPLO:

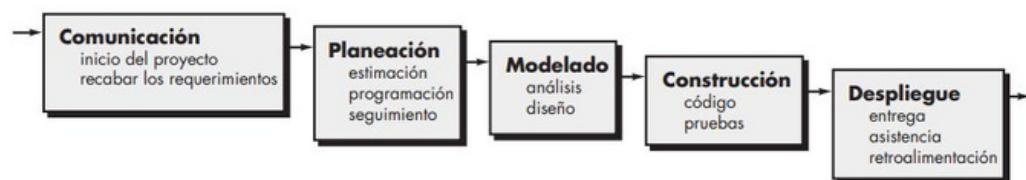
- ✓ Nombre del patrón
- ✓ Propósito
- ✓ Tipo
- ✓ Contexto inicial
- ✓ Problema
- ✓ Solución
- ✓ Contexto resultante
- ✓ Patrones relacionados
- ✓ Usos conocidos/ejemplos

Nombre: GalletaChocolateRelación
Contexto: Cocinando galletas de chocolate para tu familia y amigos
Considera estos patrones anteriormente: AzucarRelación, HarinaRelación, HuevoRelación
Problema: Determinar la relación óptima entre trozos de chocolate y masa de galleta
Solución: Observe que la mayoría de la gente considera los trozos de chocolate como lo mejor de la galleta. También observe que demasiado chocolate haría que la galleta no se mantuviese junta, disminuyendo su atractivo. Como se están cocinando pocas galletas, el coste no es un gran problema. Así, utilice tanto chocolate como pueda manteniendo la galleta compacta.
A considerar ahora: NuecesRelación o TiempoCocción o MétodoCongelación.

Unidad 2: Tipos de Patrones de Procesos

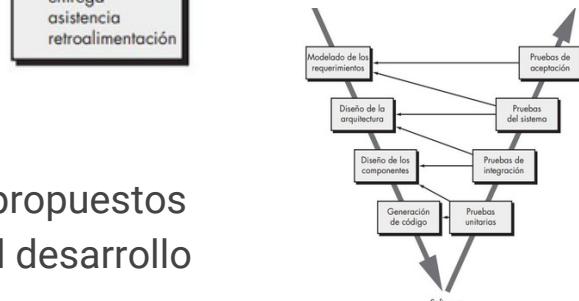
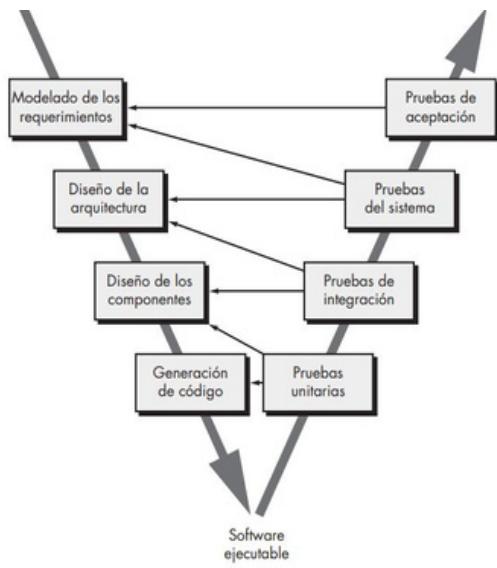
- **Patrón de Etapa:** define un problema asociado con una actividad estructural para el proceso. Ejemplo: *Establecer Comunicación*
- **Patrón de Tarea:** define un problema asociado con una acción o tarea de trabajo de la ingeniería de software y que es relevante para el éxito de la práctica de ingeniería de software. Ejemplo: *Recabar Requerimientos*
- **Patrón de Fase:** define la secuencia de las actividades estructurales que ocurren dentro del proceso, aun cuando el flujo general de las actividades sea de naturaleza iterativa. Ejemplo: *modelo espiral*

Unidad 2: Modelos de Procesos Prescriptivos

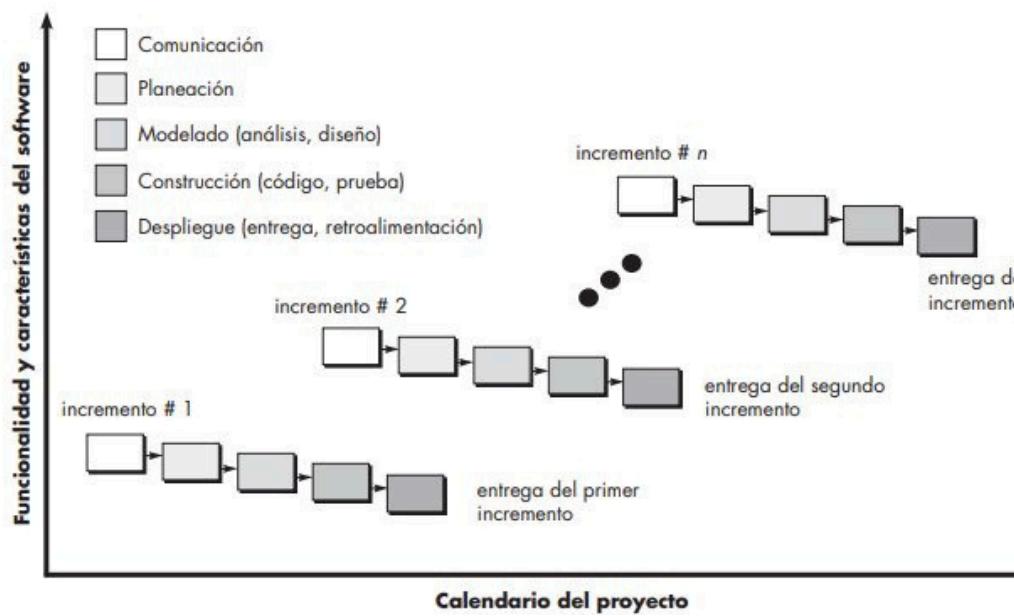


Los modelos de proceso prescriptivo fueron propuestos originalmente para poner orden en el caos del desarrollo de software

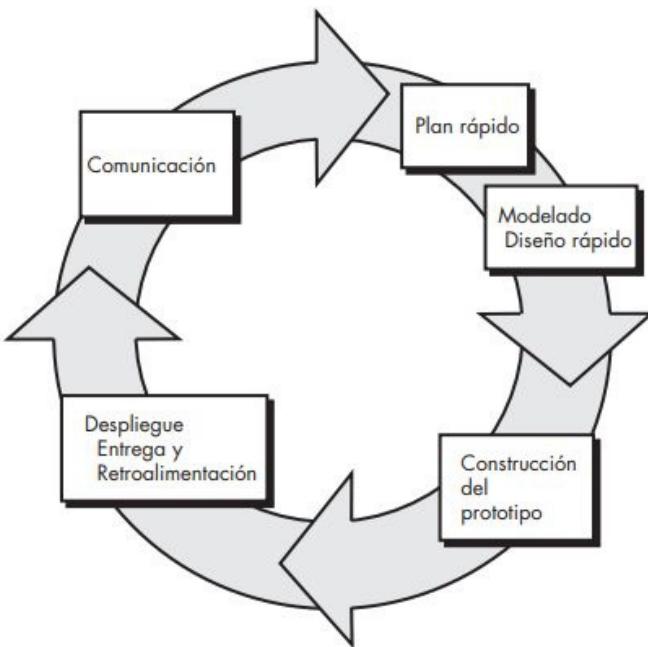
El modelo de la cascada es el paradigma más antiguo de la ingeniería de software



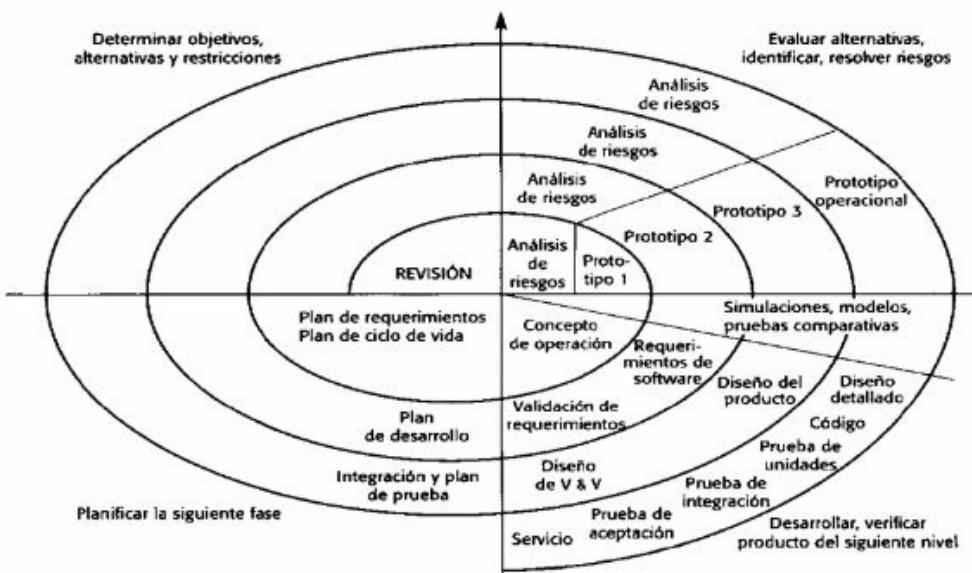
El modelo en V proporciona una forma de visualizar el modo de aplicación de las acciones de verificación y validación al trabajo de ingeniería inicial



Modelo Incremental: es útil en particular cuando no se dispone de personal para la implementación completa del proyecto



Modelo Evolutivo: es iterativo y se caracteriza por la manera en la que permite desarrollar versiones cada vez más completas del software - *los prototipos*



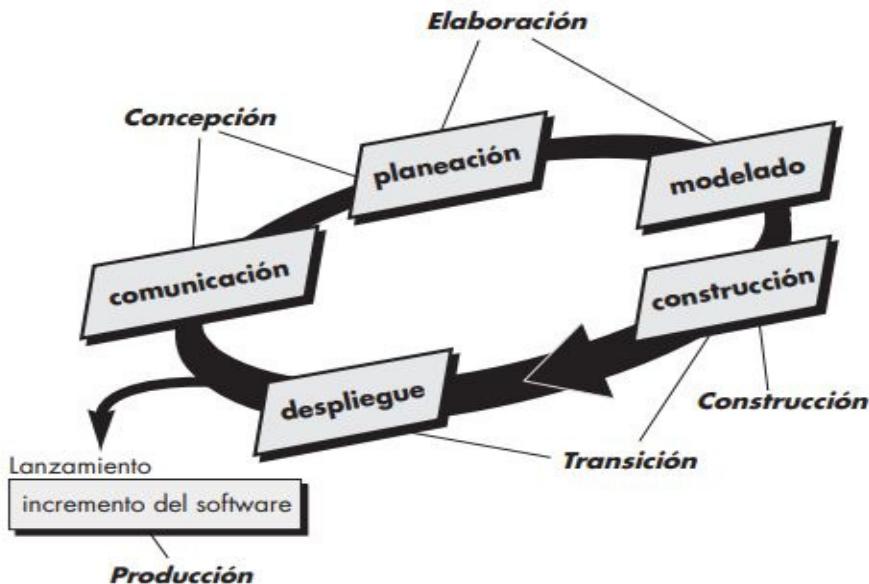
Modelo Espiral: modelo basado en riesgo, se aplica a todo el proceso del SW. Cada ciclo representa una fase del desarrollo de SW

Figura 4.5 Modelo en espiral de Boehm para el proceso del software (© IEEE, 1988).

Unidad 2: El Proceso Unificado

El proceso unificado es un intento por obtener los mejores rasgos y características de los modelos tradicionales del proceso del software, pero en forma que implemente muchos de los mejores principios del desarrollo ágil de software.

- Reconoce la importancia de la comunicación con el cliente y los métodos directos para describir su punto de vista (el caso de uso).
- Hace énfasis en la importancia de la arquitectura del software.
- Sugiere un flujo del proceso iterativo e incremental



Fases del PU:

- Concepción
- Elaboración
- Construcción
- Transición
- Producción

El Proceso Unificado - Características

- **Dirigido por casos de uso:** Los casos de uso reflejan lo que los usuarios futuros necesitan y guían el proceso de desarrollo (flujos de trabajo)
- **Centrado en la arquitectura:** La arquitectura muestra la visión común del sistema completo. El equipo de proyecto y los usuarios acuerdan los cimientos del sistema, mediante iteraciones, comenzando por los CU relevantes desde el punto de vista de la arquitectura. Se incluyen los diagramas de UML.
- **Iterativo e Incremental:** Una iteración involucra actividades de todos los flujos de trabajo, aunque desarrolla fundamentalmente algunos más que otros. Cada mini proyecto es una iteración que resulta en un incremento.

El Proceso Unificado -> Elementos

- **Trabajadores (“quién”):** Define las responsabilidades (rol) de un individuo, grupo o sistema automatizado, que trabajan en conjunto como un equipo.
- **Actividades (“cómo”):** Es una tarea que tiene un propósito claro, es realizada por un trabajador y manipula elementos.
- **Artefactos (“qué”):** Productos tangibles producidos, modificados y usados por las actividades. Pueden ser modelos,, código fuente y ejecutables.
- **Flujo de actividades (“cuándo”):** Secuencia de actividades realizadas por trabajadores y que produce un resultado de valor observable.

Unidad 3: Desarrollo Agile

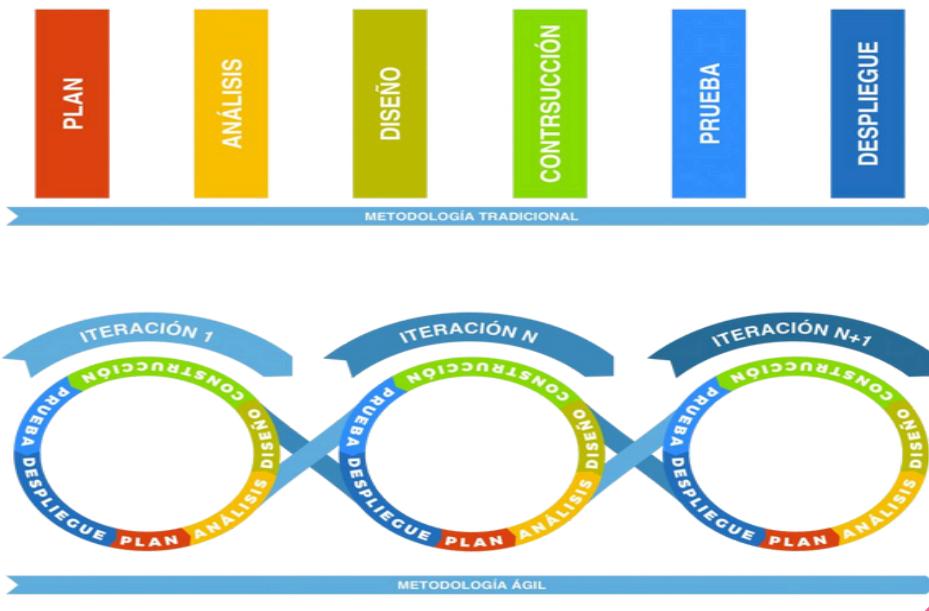
En 2001, Kent Beck y otros 16 notables desarrolladores de software, escritores y consultores formaron la “**Alianza Ágil**” y firmaron el “**Manifiesto por el desarrollo ágil de software**”. En él se establecía lo siguiente: “*Estamos descubriendo formas mejores de desarrollar software, por medio de hacerlo y de dar ayuda a otros para que lo hagan. Ese trabajo nos ha hecho valorar:*”

- Los individuos y sus interacciones, sobre los procesos y las herramientas.
- El software que funciona, más que la documentación exhaustiva.
- La colaboración con el cliente, y no tanto la negociación del contrato.
- Responder al cambio, mejor que apegarse a un plan.

¿Qué es? La ingeniería de software ágil combina una filosofía con un conjunto de lineamientos de desarrollo y pone el énfasis en:

- La satisfacción del cliente y en la entrega rápida de software incremental
- Los equipos pequeños y muy motivados para efectuar el proyecto
- Los métodos informales, los productos del trabajo con mínima ingeniería de software y la sencillez general en el desarrollo.
- Enfatizar la entrega por sobre el análisis y el diseño, y la comunicación activa y continua entre desarrolladores y clientes.

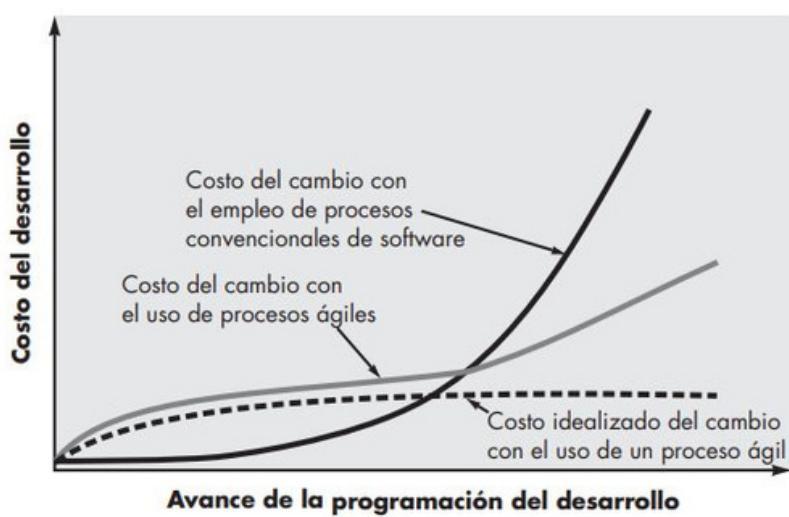
Desarrollo Agile vs Tradicional



¿Cuál es el producto final? Tanto el cliente como el ingeniero de software tienen la misma perspectiva: **el único producto del trabajo realmente importante es un “incremento de software” operativo que se entrega al cliente exactamente en la fecha acordada.** Permanecen las actividades estructurales fundamentales: **comunicación, planeación, modelado, construcción y despliegue.** Pero se transforman en un conjunto mínimo de tareas que lleva al equipo del proyecto hacia la construcción y entrega



La Agilidad y el Costo del Cambio



Unidad 3: Principios para la Agilidad

La Alianza Ágil definió 12 principios:

1. La prioridad más alta es satisfacer al cliente a través de la entrega pronta y continua
2. Son bienvenidos los requerimientos cambiantes
3. Entregar con frecuencia software **que funcione**, de dos semanas a un par de meses
4. Las personas de negocios y los desarrolladores deben trabajar juntos, a diario y durante todo el proyecto
5. Hay que desarrollar los proyectos con individuos motivados. Debe darse a éstos el ambiente y el apoyo que necesitan, y confiar en que harán el trabajo
6. El método más eficiente y eficaz para transmitir información a los integrantes de un equipo de desarrollo, y entre éstos, es la conversación cara a cara
7. La medida principal de avance es el software que funciona.
8. Los procesos ágiles promueven el desarrollo sostenible.
9. La atención continua a la excelencia técnica y el buen diseño mejora la agilidad.
10. Es esencial la simplicidad.
11. Las mejores arquitecturas, requerimientos y diseños surgen de los equipos con organización propia.
12. El equipo reflexiona a intervalos regulares sobre cómo ser más eficaz.

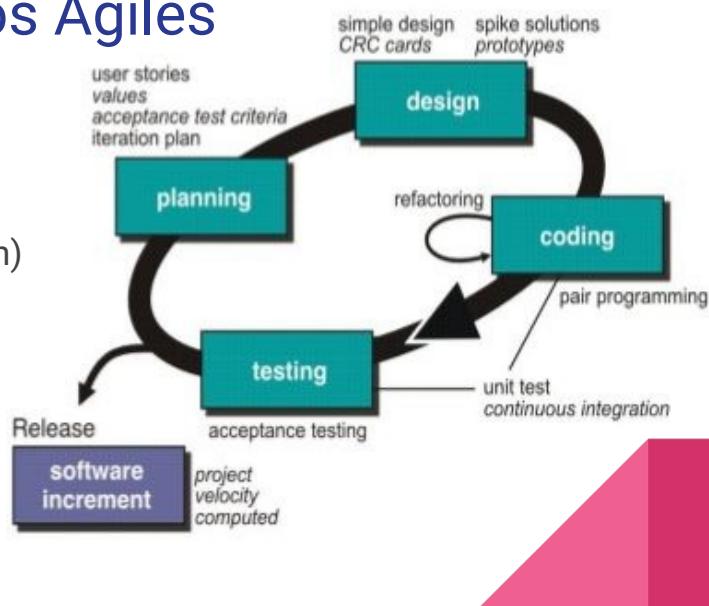
Factores humanos: *El desarrollo ágil se centra en los talentos y habilidades de los individuos, y adapta el proceso a personas y equipos específicos.*

- Competencia
- Enfoque común
- Colaboración
- Habilidad para la toma de decisiones
- Capacidad de resolver problemas difusos
- Confianza y respeto mutuo
- Organización propia

Ejemplos de procesos Ágiles

Programación Extrema (XP)

- Simplicidad
- Comunicación eficaz
- Feedback (retroalimentación)
- Disciplina
- Estimulación del rediseño
- Programación en parejas
- Prueba de humo
- Integración continua
- Pruebas de aceptación



Programación Extrema (XP) - Desventajas:

- Volatilidad de los requerimientos
- Necesidades conflictivas del cliente
- Los requerimientos se expresan informalmente
- Falta de un diseño formal

Ejemplos de modelos ágiles -> SCRUM



Unidad 3: Otros ejemplos de modelos ágiles

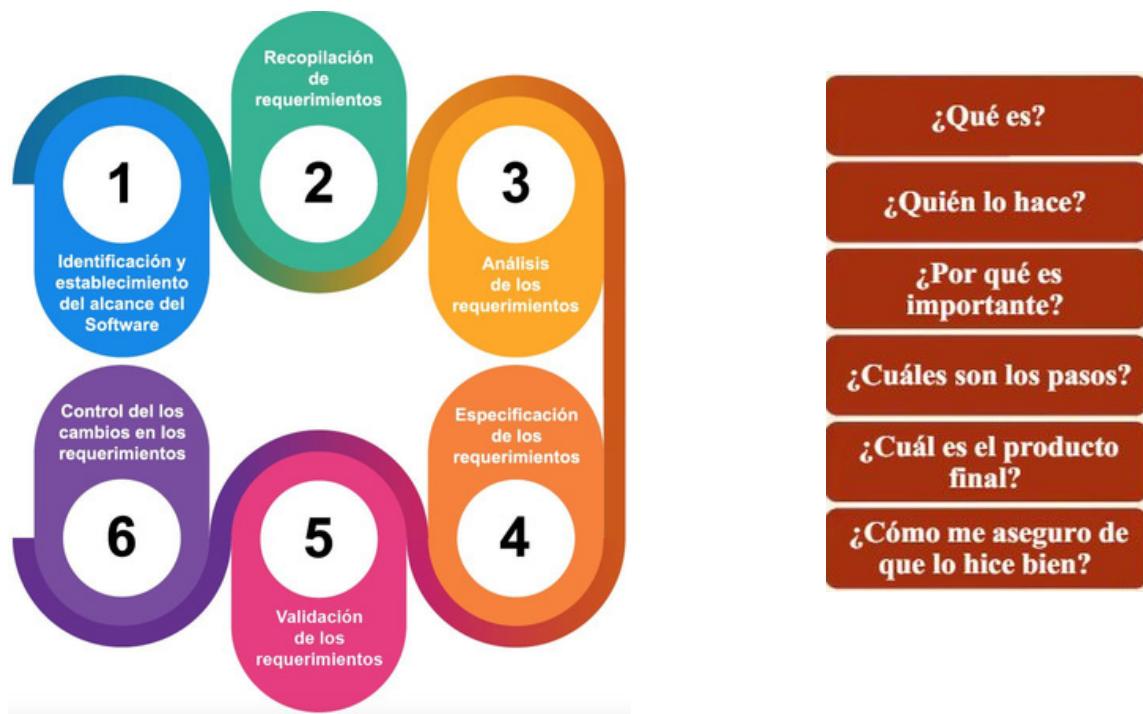
- Desarrollo adaptativo de software (DAS)
- Método de desarrollo de sistemas dinámicos (MDSD)
- Cristal
- Desarrollo impulsado por las características (DIC)
- Desarrollo esbelto de software (DES)
- Modelado ágil (MA)
- Proceso unificado ágil (PUA)

Unidad 4: Comprensión de Requerimientos

La ingeniería de requerimientos proporciona el mecanismo apropiado para:

- Entender lo que desea el cliente
- Analizar las necesidades / Evaluar la factibilidad
- Negociar una solución razonable
- Especificar la solución sin ambigüedades
- Validar la especificación

 *Ejemplo: Si el cliente dice "quiero un sistema para registrar ventas", hay que entender si se refiere a un sistema de punto de venta, una app móvil, qué tipo de productos vende, si necesita facturación, informes, usuarios con distintos permisos, etc.*



Establecer las bases:

- Identificación de los participantes
- Reconocer los múltiples puntos de vista
- Trabajar hacia la colaboración
- Hacer las primeras preguntas
 - ¿Quién está detrás de la solicitud de este trabajo?
 - ¿Quién usará la solución?
 - ¿Cuál será el beneficio económico de una solución exitosa?
 - ¿Hay otro origen para la solución que se necesita?

4. Mapa mental



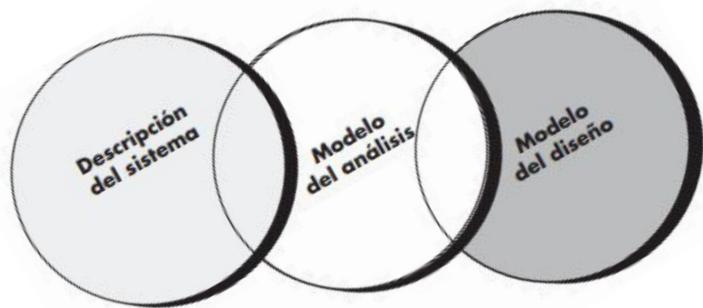
Preguntas que permiten entender mejor el problema:

- ¿Cuál sería una “buena” salida generada por una solución exitosa?
- ¿Qué problemas resolvería esta solución?
- ¿Puede mostrar (o describir) el ambiente de negocios en el que se usaría la solución?
- ¿Hay aspectos especiales del desempeño o restricciones que afecten el modo en el que se enfoque la solución?
- ¿Puede otra persona dar información adicional?
- ¿Debería yo preguntarle algo más?

Unidad 4: Modelado de Requerimientos

El modelo de requerimientos debe lograr tres objetivos principales:

- Describir lo que requiere el cliente.
- Establecer una base para la creación de un diseño de software.
- Definir un conjunto de requerimientos que puedan validarse una vez construido el software.



Durante el modelado de los requerimientos, la atención se centra en **qué**, no en **cómo**.

- ¿Qué interacción del usuario ocurre en una circunstancia particular?
- ¿Qué objetos manipula el sistema?
- ¿Qué funciones debe realizar el sistema?
- ¿Qué comportamientos tiene el sistema?
- ¿Qué interfaces se definen?
- ¿Qué restricciones son aplicables?

⌚ *El modelado no es programación, es una forma de documentar lo que el sistema debe hacer y cómo se espera que funcione.*

lo que el sistema

Reglas prácticas:

- A esta altura, no preocuparse tanto por los detalles
- Cada elemento debe dar una visión del dominio de la información, de la función y del comportamiento del sistema.
- No nos preocupemos en esta etapa por temas de infraestructura
- Minimizar acoplamiento
- El modelo de requerimientos agrega valor para todos los participantes.
- Mantener el modelo tan sencillo como se pueda

Entre las principales herramientas que podemos valernos a la hora de completar el modelado de los requerimientos, podemos destacar:

- Diagramas de actividades / canal
- Diagramas de clases
- Diagramas de estado
- El diagrama de entidad / relación
- Historias de Usuario
- Diagramas de caso de uso (UML)

Unidad 4: El Diagrama de Actividades

El diagrama de actividad UML es una representación gráfica del flujo de interacción dentro de un escenario específico (similar a un diagrama de flujo)

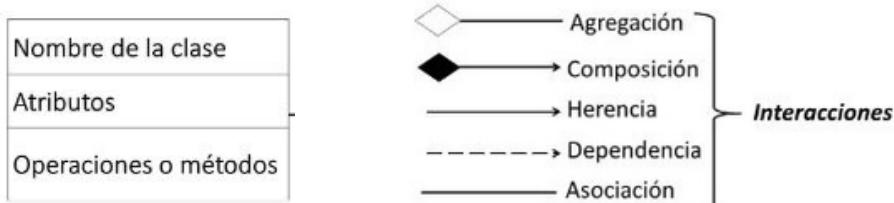
- Utiliza rectángulos redondeados para denotar una función específica del sistema.
- Las flechas se usan para representar flujo a través de éste.
- Los rombos ilustran una ramificación de las decisiones (cada flecha que salga del rombo se etiqueta)
- Las líneas continuas indican que están ocurriendo actividades en paralelo.

Unidad 4: El Diagrama de Clases

Un diagrama de clases porta una visión estática o de estructura de un sistema, es usado para modelar clases, incluidos sus atributos, operaciones, relaciones y asociaciones con otras clases.

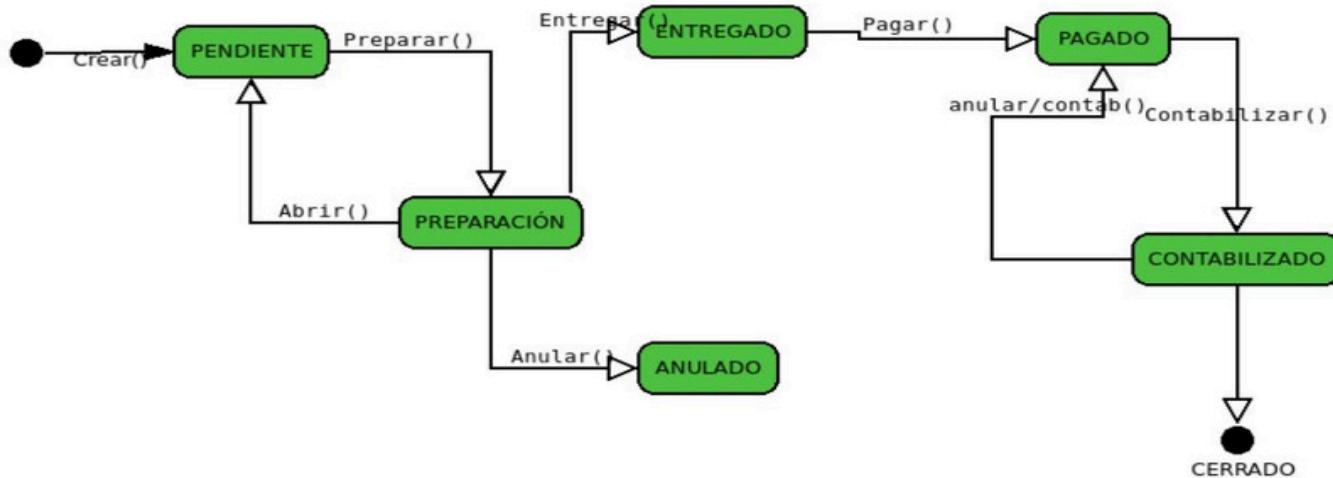
Sus elementos principales son cajas:

- La parte superior contiene el nombre de la clase.
- La sección media menciona sus atributos.
- La tercera sección del diagrama de clase contiene las operaciones



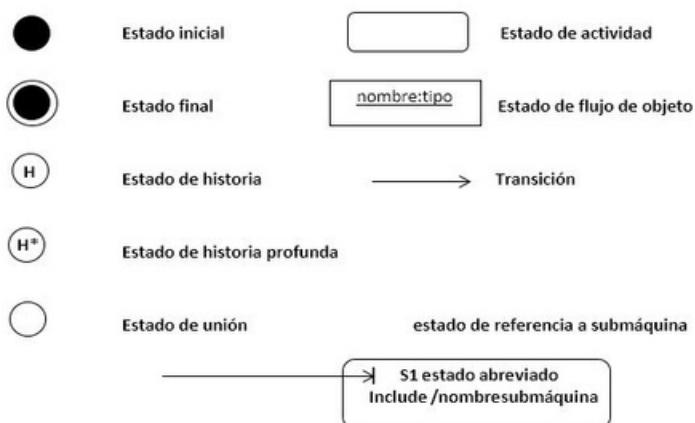
Unidad 4: El Diagrama de Estado

El **diagrama de estado** es un método de representación del **comportamiento** de un sistema, que ilustra sus estados y los eventos que ocasionan que el sistema cambie de estado. Un **estado** es cualquier modo de comportamiento observable desde el exterior.



El Diagrama de Estado - Notación

Además, el diagrama de estado indica **acciones** tomadas como consecuencia de un **evento** en particular.



El **diagrama de estado** lo utilizamos generalmente para mostrar cómo cambia un **objeto** a lo largo de su tiempo de vida. Mientras que el **diagrama de actividades** resulta más útil para describir un comportamiento paralelo, o cuando queremos mostrar qué comportamientos interactúan en varios casos de uso.

Unidad 4: El Diagrama de Canal (swimline)

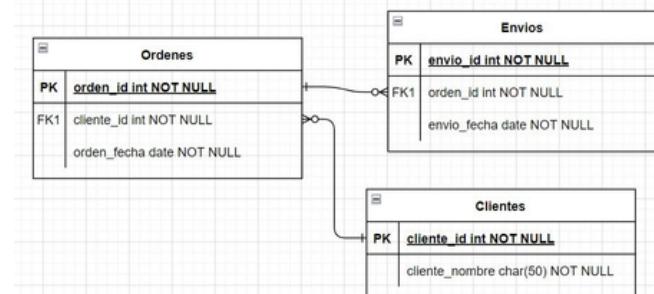
El **diagrama de canal** de UML es una variación útil del diagrama de actividades y permite representar el flujo de actividades descritas por el caso de uso.

- Indica qué actor es responsable de la acción descrita por un rectángulo de actividad.
- Las responsabilidades se representan con segmentos paralelos que dividen el diagrama en forma vertical, como los canales o carriles.

Unidad 4: El diagrama de Entidad-Relación (DER)

El diagrama **entidad-relación** (DER) representa todos los datos que se introducen, almacenan, transforman y generan dentro de una aplicación.

- Define todos los objetos de datos que se procesan dentro del sistema, la relación entre ellos e información que sea importante para las relaciones.
- Los objetos tienen **atributos** y **relaciones**.
- El DER fue propuesto por primera vez para diseñar sistemas de bases de datos relacionales.



Unidad 5: Lineamientos de la Calidad

Son una especie de reglas de buen diseño que ayudan a que un sistema sea **entendible, mantenible, escalable y confiable**.

- Arquitectura basada en patrones conocidos
- Incluir representaciones de datos, arquitectura, interfaces y componentes
- Patrones reconocibles de datos.
- Debe ser modular, con una división lógica en elementos o subsistemas
- Interfaces que reduzcan la complejidad
- Notación que comunique con eficacia

Imaginemos que construimos una casa: no alcanza con que no se caiga, también tiene que estar bien distribuida, con buenas cañerías, instalaciones seguras, etc.

- **Arquitectura basada en patrones conocidos:** como usar planos de casas ya probadas en vez de inventar una desde cero.
- **Representaciones claras de datos:** diagramas que muestren cómo se conecta todo. Ej: plano eléctrico de un edificio.
- **Patrones reconocibles de datos:** como listas, árboles, o bases de datos bien normalizadas.
- **Modularidad:** dividir el sistema en partes pequeñas y lógicas. Ej: armar un auto: un equipo el motor, otro la carrocería. Si algo falla, no tenés que desarmar todo.
- **Interfaces simples:** las distintas partes se comunican de forma clara y sencilla. Como un enchufe: no necesitás saber cómo funciona adentro, solo cómo conectarlo.

Unidad 5: Atributos de la calidad (FURPS)

Atributos de la calidad (desarrollado por HP):

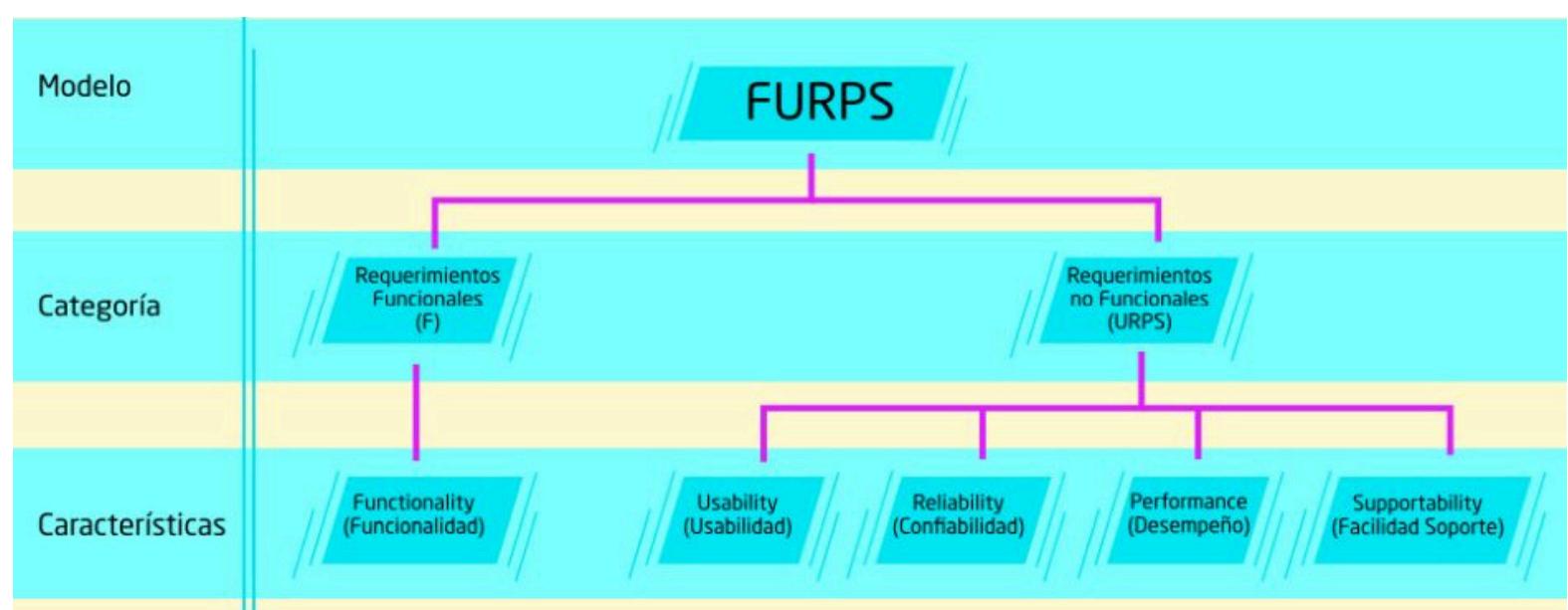
- Funcionalidad
- Usabilidad
- Confiabilidad
- Rendimiento
- Mantenibilidad.



Unidad 5: FURPS

FURPS es un acrónimo en inglés de un conjunto de elementos empleados para clasificar los atributos de calidad del software (requisitos funcionales y no funcionales):

- **Funcionalidad (Functionality)**: capacidad (tamaño y generalidad del conjunto de funciones), reutilización (compatibilidad, interoperabilidad, portabilidad), seguridad (seguridad y explotabilidad)
- **Usabilidad (Usability)** o facilidad de uso: factores humanos, estética, coherencia, documentación, capacidad de respuesta
- **Confiabilidad (Reliability)**: disponibilidad (frecuencia de fallos/robustez/durabilidad/resiliencia), extensión y duración del fallo (recuperabilidad/supervivencia), previsibilidad (estabilidad), precisión (frecuencia/gravedad del error)
- Rendimiento (Performance): velocidad, eficiencia, consumo de recursos (energía, memoria RAM, caché, etc.), rendimiento, capacidad, escalabilidad
- Mantenibilidad (Supportability) (capacidad de servicio, capacidad de mantenimiento, sostenibilidad, velocidad de reparación): capacidad de prueba, flexibilidad (capacidad de ser modificado, configurabilidad, adaptabilidad, extensibilidad, modularidad), capacidad de instalado, capacidad de ser configurado de acuerdo a la ubicación.



Ventajas del modelo de FURPS:

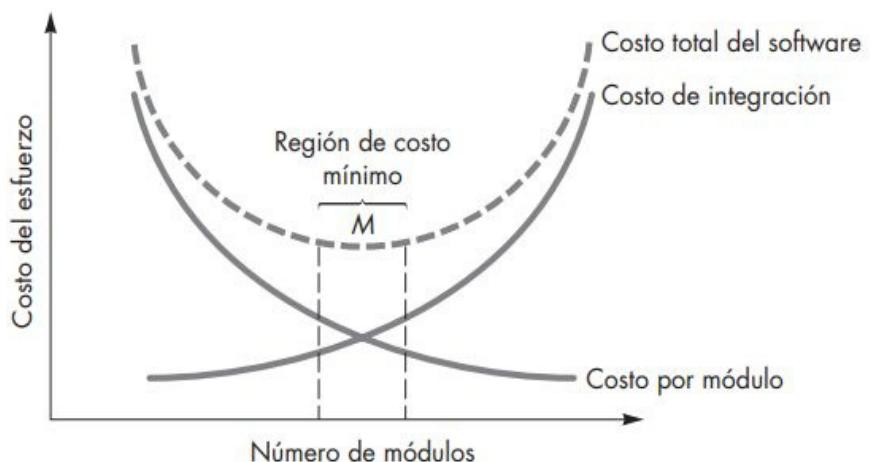
- Los criterios son claramente entendibles, esto implica su facil utilizacion.
- Tiene en cuenta las fallas en el producto y en el proceso esto permite una mayor corrección. Se podría utilizar no para una sino para varios proyectos.

Desventajas:

- Se necesita de muestras métricas lo que implica un mayor esfuerzo de tiempo y costo.

Unidad 5: Conceptos de Diseño

- Abstracciones
 - De procedimiento: secuencia de instrucciones que tienen una función específica y limitada
 - De datos: conjunto de datos con nombre que describe a un objeto de datos.
- Arquitectura: estructura de organización de los componentes de un programa (módulos), la forma en la que interactúan y la estructura de datos que utilizan.
- Patrones: nombre propio de puntos de vista que contienen la esencia de una solución demostrada para un problema recurrente dentro de cierto contexto.
- División de problemas
- Modularidad (en su punto justo)
- Ocultamiento de información
- Refinamiento
- Rediseño
- Independencia funcional
 - máxima cohesión
 - mínimo acoplamiento



Unidad 5: Administración de la Calidad

 **Máxima cohesión (cada cosa hace bien una sola tarea):** un módulo o componente debe tener una única responsabilidad clara. Ejemplo: un equipo que solo arma las ruedas de un auto. Son expertos en eso, lo hacen rápido y bien. No les pidas que pinten el auto, porque eso lo hace otro equipo. Eso es **alta cohesión**: cada grupo (módulo) hace solo lo suyo, y lo hace bien.

 **Mínimo acoplamiento (que las partes dependan lo menos posible entre sí):** los módulos deben estar lo más independientes posible unos de otros. Ejemplo: el equipo de frenos necesita llamar todo el tiempo al equipo del motor para funcionar. Si el del motor se toma vacaciones, los frenos no andan. (**alto acoplamiento**, dependencias innecesarias).

Bajo acoplamiento: el equipo de frenos recibe sólo lo que necesita, sin tener que saber cómo funciona el motor por dentro.

Para lograr software de alta calidad, deben ocurrir cuatro actividades:

- Seguir procesos y prácticas probadas de la ingeniería de software
- Administrar bien el proyecto
- Realizar un control de calidad exhaustivo
- Contar con infraestructura de aseguramiento de la calidad.

Un sistema bien diseñado tiene componentes bien definidos (**alta cohesión**) que se comunican lo justo y necesario (**acoplamiento bajo**).

En el desarrollo del software, la **calidad del diseño** es el grado en que el diseño cumple las funciones y características especificadas en el modelo de requerimientos.

La **calidad de la conformidad** es el grado en el que la implementación se apega al diseño y en el que el sistema cumple sus metas de requerimientos y desempeño.

satisfacción del usuario = producto que funciona + buena calidad + entrega dentro del presupuesto y plazo

Dimensiones de la calidad de Garvin:

- **Calidad del desempeño:** ¿el software hace lo que dice la especificación?
- **Calidad de las características:** ¿genera impacto a primera vista?
- **Confiabilidad:** ¿contiene bugs? ¿Tiene alta disponibilidad?
- **Conformidad:** ¿sigue los estándares acordados?
- **Durabilidad:** ¿es mantenable y/o extensible fácilmente?
- **Servicio:** ¿tiene un costo de mantenimiento razonable?
- **Estética:** ¿es estéticamente “lindo”?
- **Percepción:** ¿con qué percepción llega al mercado?

Factores de la calidad de McCall:



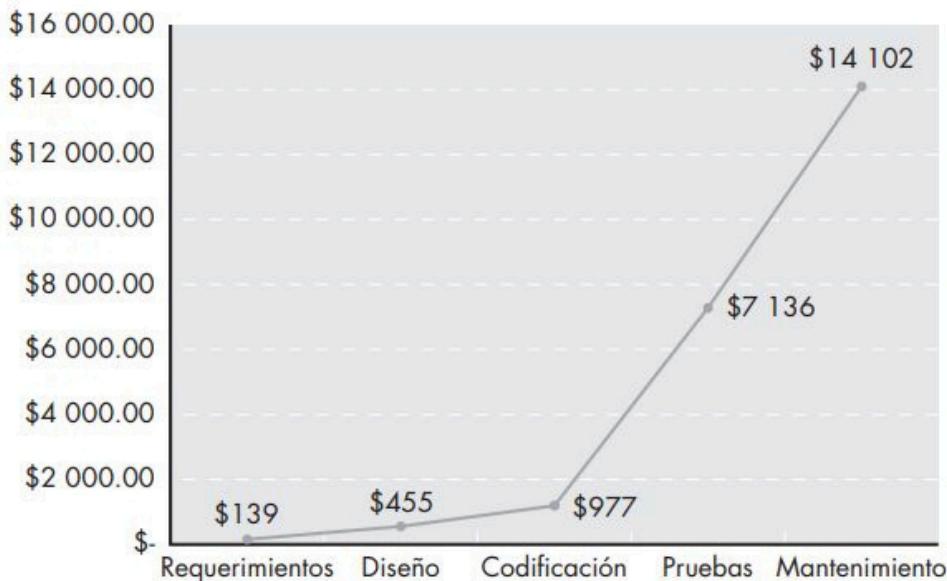
¿Qué significa que un software sea “suficientemente bueno”?

¿Es aceptable producir software “suficientemente bueno”?

El costo de la calidad:

- La calidad es importante, pero cuesta tiempo y dinero (demasiado) lograr el nivel de calidad en el software que en realidad queremos.
- La calidad tiene un costo, ***la mala calidad tiene uno mucho más alto.***
- Los costos se pueden clasificar en:
 - Costos de prevención: procesos, planes, creación de casos de prueba
 - Costos de evaluación: ejecución de pruebas, evidencias, bug fixing
 - Costos de falla: costos internos (previo a entregar el producto al cliente) y costos externos (cuando entregamos el producto final, o al menos una versión)

Costos relativos a corregir defectos y errores



- **Riesgos:** la mala calidad trae riesgos, algunos muy serios
- **Negligencia y responsabilidad:** discusiones que pueden llevar a cancelación de contratos y retención de pagos.
- **Calidad y Seguridad:** no hay software seguro si no nos centramos en la calidad desde el diseño mismo. Las posibles fallas se empiezan a reducir desde la concepción de la arquitectura del sistema.

- **El efecto de la administración del proyecto:**
 - Decisiones de estimación
 - Decisiones de programación
 - Decisiones de toma de riesgos

Lograr la calidad del software: la calidad del software no sólo se ve. Es el resultado de la buena administración del proyecto y de una correcta práctica de la ingeniería de software, basado en 4 actividades:

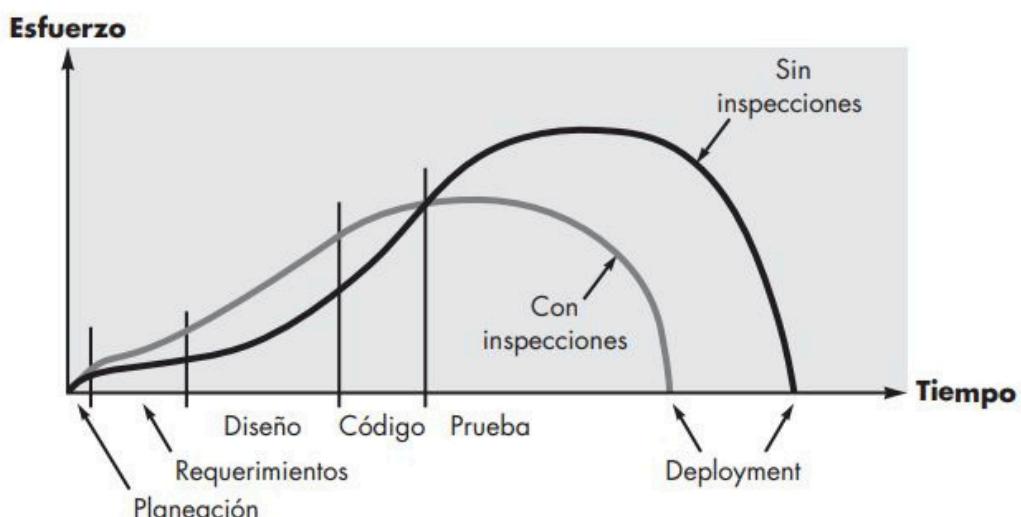
- **Métodos de la ingeniería de software:** desde la concepción del sistema mismo
- **Técnicas de administración de proyectos:** asociadas a la administración de la calidad.
- **Control de calidad (QC):** serie de etapas de prueba para detectar los errores en procesamiento lógico, manipulación de datos y comunicación con la interfaz
- **Aseguramiento de la calidad (QA):** conjunto de funciones de auditoría y reportes para evaluar la eficacia y completitud de las acciones de control de calidad

Unidad 5: Técnicas de Revisión

- Las revisiones del software “purifican” los productos del trabajo de la ingeniería de software, incluso los modelos de requerimientos y diseño, código y datos de prueba. Actúan como “filtro”.
- Las revisiones técnicas son el mecanismo más eficaz para detectar los errores en una etapa temprana del proceso de software.
- Vamos a diferenciar:
 - Error: problema de calidad descubierto antes de entregar el producto al cliente
 - Defecto: problema descubierto después de la entrega del producto

Métricas de revisión:

- Esfuerzo de preparación (horas hombre)
- Esfuerzo de validación (horas hombre)
- Esfuerzo de repetición (horas hombre)
- Tamaño del producto del trabajo, TPT (historias de usuario, clases, etc)
- Errores menores detectados, (número de errores “menores”)
- Errores graves detectados, (número de errores más complejos de corregir)
- Densidad de errores: errores totales / TPT



Esfuerzo en cada etapa, con y sin revisiones

Revisiones informales: mucho más ágiles, pero menos eficientes. Se recomienda usar un checklist.

Revisiones técnicas formales (RTF), que buscan los siguientes objetivos:

- Descubrir los errores en funcionamiento, lógica o implementación
- Verificar que el software que se revisa cumple sus requerimientos
- Garantizar que el software sigue estándares predefinidos;
- Obtener software desarrollado de manera uniforme
- Incluye walkthroughs + inspecciones
- Se debe preparar una reunión de revisión, y salir de ella con el reporte de revisión correspondiente.



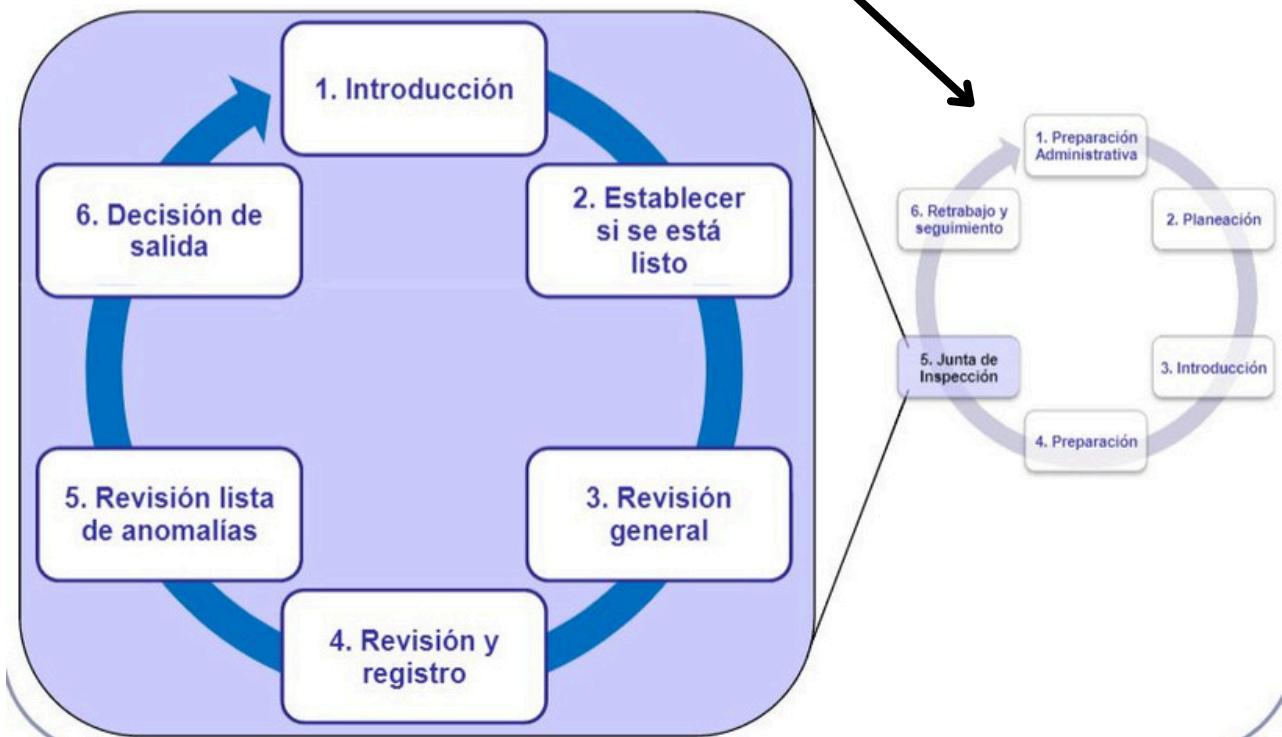
Lineamientos para la revisión:

- Revise el producto, no al productor
- Establezca una agenda y síganla. Limite el debate y las contestaciones (moderador)
- Enuncie áreas de problemas, pero no intente resolver cada uno. Tome notas por escrito.
- Limite el número de participantes e insista en la preparación previa
- Desarrolle una lista de verificación para cada producto que se revise.
- Asigne recursos y programe tiempo para las RTF
- Dé una capacitación previa a todos los revisores
- Revise las primeras revisiones



Roles durante la inspección:

- Líder (moderador)
- Escriba
- Lector
- Autor
- Inspector



Desventajas de las revisiones:

- Requiere tiempo de expertos
- No se verifican características no funcionales (ejemplo: rendimiento)
- Validan cumplimiento de lo especificado, en lugar de validar lo que quiere el cliente

Diferencias entre recorrido e inspección:

- Propósito menos formal, más educativo
- Menos participantes, y menos requerimientos de entrada
- Menos información de salida

Recuerde que:

- Si hablamos de **Control de Calidad (QC)**, pensamos en verificar la calidad del **producto**
- Si en cambio hablamos de **Aseguramiento de Calidad (QA)**, nos referimos a los **procesos de gestión de calidad**.



Unidad 6: Estrategias de prueba de software

Prueba (Testing): es el proceso de ejecutar un programa con el objetivo de encontrar errores.

Ejemplo: Tenés una calculadora y escribís un test para verificar que $2 + 2$ devuelve 4.

Depuración (Debugging): es el proceso de encontrar y corregir los errores que aparecen durante la ejecución o prueba del software.

Ejemplo: El resultado de $2 + 2$ da 5. Usás el depurador del IDE, pones un breakpoint en la función `suma()` y descubrís que se está sumando mal por un error de tipeo.

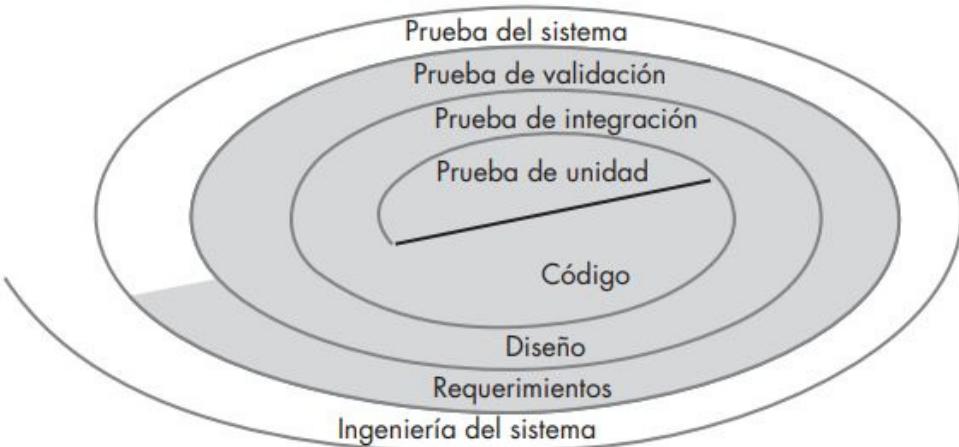
Validación: es confirmar que el producto cumple con las necesidades del usuario final.

Ejemplo: El cliente pidió una app para registrar ventas, pero el sistema que hicimos solo permite consultar. No valida bien.

Verificación: es confirmar que el producto se ha construido correctamente según las especificaciones.

Ejemplo: El formulario tiene que validar que el campo “email” esté completo y con un formato válido. Si eso funciona, está verificado.

- La **prueba** es un conjunto de actividades que pueden planearse por adelantado y realizarse de manera sistemática
- **Prueba** y **depuración** son actividades diferentes, pero la depuración debe incluirse en cualquier estrategia de prueba.
- La **verificación** se refiere al conjunto de tareas que garantizan que el software implementa correctamente una función específica.
- La **validación** es un conjunto diferente de tareas que aseguran que el software que se construye sigue los requerimientos del cliente.



- **¿Cuándo terminan las pruebas?**
- **¿Cómo se sabe que se ha probado lo suficiente?**

Criterios para completar pruebas:

- ¿Cuándo terminan las pruebas?
 - **Nunca se termina de probar; la carga simplemente pasa de usted al usuario final.**
- ¿Cómo se sabe que se ha probado lo suficiente?
 - **Las pruebas terminan cuando se agota el tiempo o el dinero**

Aspectos estratégicos para completar pruebas de forma exitosa:

- Especificar los requerimientos del producto en forma cuantificable.
- Establecer de manera explícita los objetivos de las pruebas
- Desarrollar un perfil para cada categoría de usuario de software.
- Desarrollar un plan de prueba que enfatice **pruebas de ciclo rápido**.
- Construir software “robusto” que esté diseñado para probarse a sí mismo.
- Usar revisiones técnicas efectivas como filtro previo a las pruebas, valorando la estrategia de prueba y los casos de prueba.
- Aplicar mejora continua para el proceso de pruebas

Una **prueba unitaria** (o **unit test**) es una prueba que se hace sobre **una unidad mínima del código**, como una **función** o **método**, de forma **aislada** para asegurarse de que hace lo que se espera.

En otras palabras: "**Estoy probando que esta función, por si sola, funcione bien.**

- Permiten detectar errores temprano, antes de integrar todo el sistema.
- Ayudan a mantener el código funcionando cuando hacés cambios.
- Funcionan como **documentación viva**: muestran cómo debe comportarse el código.

```
js

function sumar(a, b) {
    return a + b;
}
```

Una prueba unitaria podría ser:

```
js

test("sumar 2 y 3 debería devolver 5", () => {
    expect(sumar(2, 3)).toBe(5);
});
```

Principales clases de pruebas: **Prueba de integración**

- Son una técnica sistemática para construir la arquitectura del software mientras se llevan a cabo pruebas para descubrir errores asociados con la interfaz.
- El objetivo es tomar los componentes probados de manera individual y construir una estructura de programa que se haya dictado por diseño

Principales clases de pruebas: **Prueba de regresión**

- Cada vez que se agrega un nuevo módulo como parte de las pruebas de integración, el software cambia.
- Se establecen nuevas rutas de flujo de datos, ocurren nuevas operaciones de entrada/salida y se invoca nueva lógica de control
- Dichos cambios pueden causar problemas con las funciones que anteriormente trabajaban sin fallas

La **prueba de regresión** es la nueva ejecución de algún subconjunto de pruebas que ya se realizaron a fin de asegurar que los cambios no propagaron efectos colaterales no deseados.

- Las pruebas de regresión se pueden realizar manualmente, al volver a ejecutar un subconjunto de todos los casos de prueba o usando herramientas de captura/reproducción automatizadas
- La **suite de prueba de regresión** (el subconjunto de pruebas que se va a ejecutar) contiene tres clases diferentes de casos de prueba:
 - Una muestra representativa de pruebas que ejercitará todas las funciones de software.
 - Pruebas adicionales que se enfocan en las funciones del software que probablemente resulten afectadas por el cambio.
 - Pruebas que se enfocan en los componentes del software que cambiaron

Principales clases de pruebas: **Prueba de humo**, un enfoque de prueba de integración que se usa cuando se desarrolla software de producto.

- Se diseña como un mecanismo de ritmo para proyectos críticos en el tiempo
- No tiene que ser exhaustiva, pero debe exponer los problemas principales
- Proporciona algunos beneficios en proyectos de software complejos y cruciales en el tiempo:
 - Se minimiza el riesgo de integración, se ejecutan diariamente
 - La calidad del producto final mejora
 - El diagnóstico y la corrección de errores se simplifican
 - El progreso es más fácil de valorar.

Unidad 6: Estrategias de prueba de software O.O.

El objetivo de probar es encontrar el **mayor número posible de errores** con una **cantidad manejable de esfuerzo** aplicado durante un **lapso realista**.

- El concepto de **unidad** cambia: en general, una **clase** encapsulada es el foco de la **prueba de unidad**. Es decir, que su equivalente es la **prueba de clase**
- La **prueba de clase** la dirigen las operaciones encapsuladas por la clase y el comportamiento de estado de ésta (en vez del detalle algorítmico de un módulo y en los datos que fluyen a través de las interfaces).
- Ya no es posible probar una sola operación en aislamiento, sino como parte de una clase. Y aquí entra a jugar la **herencia**, donde deberemos probar en cada subclase (contextos diferentes).

Los **controladores** o representantes son herramientas útiles para completar las pruebas, reemplazando alguna interfaz o en donde está involucrada alguna clase que aún no está implementada.

Dobles de prueba (Test Doubles): los usamos cuando no queremos usar la **versión real de un componente** (como una base de datos, una API externa o un servicio de correo), ya sea porque:

- es lento
- no está disponible
- o no queremos que tenga efectos reales (como enviar un mail)

Unidad 6: Estrategias de prueba de software

Dummy: es un objeto que se pasa pero **no se usa**. Solo está para completar los parámetros.

📌 Ejemplo:

```
java

Usuario u = new Usuario("Juan");
Notificador dummyNotificador = new DummyNotificador(); // No hace nada

Servicio servicio = new Servicio(dummyNotificador);
servicio.procesar(u); // El notificador no se usa
```

🧠 Es como un “relleno” necesario para que el código compile o funcione.

Stub: devuelve **una respuesta fija y predecible** cuando se le llama.

📌 Ejemplo:

```
js

const stubAPI = {
    obtenerPrecioDolar: () => 100 // Valor fijo
};

expect(calcularPrecioFinal(10, stubAPI)).toBe(1000);
```

🧠 Se usa cuando querés simular respuestas, **sin lógica real**.

Fake: tiene **una implementación funcional simplificada**, lo suficiente para pruebas.

📌 Ejemplo:

```
python

class FakeBaseDeDatos:
    def __init__(self):
        self.usuarios = []

    def guardar_usuario(self, usuario):
        self.usuarios.append(usuario)
```

🧠 Imita la base de datos pero en memoria. **Sí guarda, pero de forma simple**.

Mock: verifica si se llamó a ciertos métodos y con qué parámetros.

💡 Ejemplo:

```
const mockNotificador = {  
    enviar: jest.fn()  
};  
  
procesarPedido(pedido, mockNotificador);  
  
expect(mockNotificador.enviar).toHaveBeenCalledWith(pedido.cliente);
```

💡 Se usa cuando te interesa **verificar interacciones**, no solo resultados.

Spy (Espía): igual que el objeto real, pero **espía lo que ocurre** internamente (cuántas veces se llamó, qué retornó, etc.).

💡 Ejemplo:

```
const api = {  
    obtenerDolar: () => 100  
};  
jest.spyOn(api, 'obtenerDolar');  
  
calcularPrecioFinal(10, api);  
  
expect(api.obtenerDolar).toHaveBeenCalled();
```

💡 Un **spy** deja pasar la ejecución real, pero **observa** cómo fue utilizada.

🔧 En resumen

- Usás **Dummy** para completar sin usar.
- Usás **Stub** para respuestas fijas.
- Usás **Fake** si querés algo funcional pero simple.
- Usás **Mock** si querés comprobar si se llamó algo.
- Usás **Spy** si querés espiar al objeto real.

Tipo	¿Devuelve algo?	¿Tiene lógica real?	¿Registra llamadas?	¿Se usa el objeto real?
Dummy	✗	✗	✗	✗
Stub	✓ (valor fijo)	✗	✗	✗
Fake	✓	✓ (simplificada)	✗	✗
Mock	✓ o ✗	✗	✓	✗
Spy	✓	✓	✓	✓

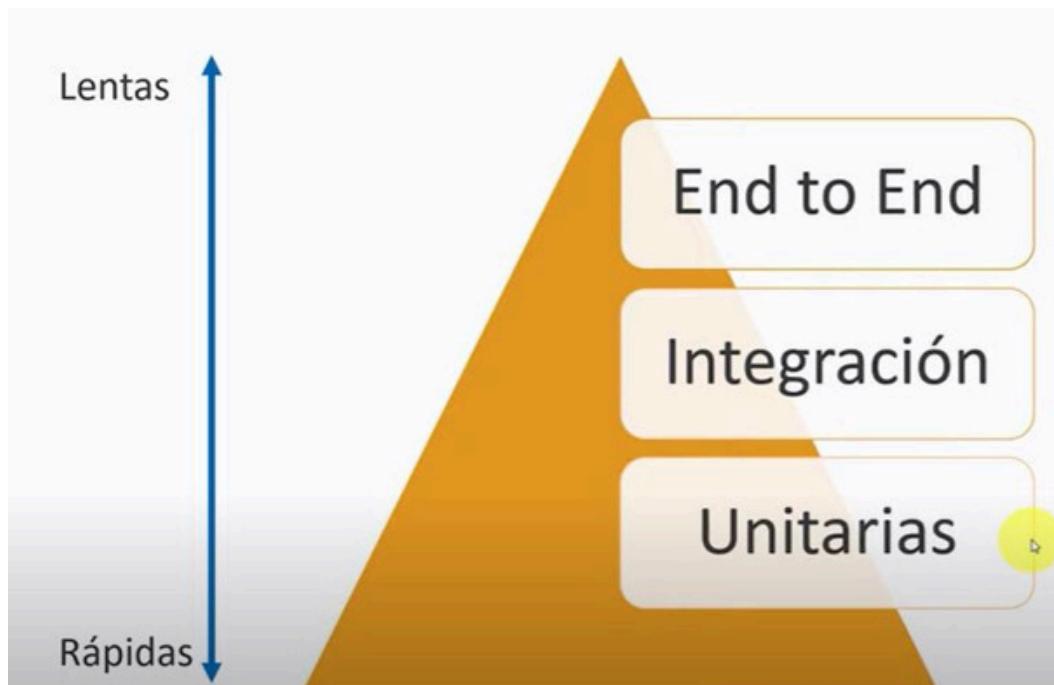
Pruebas de validación (E2E):

- Comienzan cuando terminan las pruebas de integración
- Vemos las acciones visibles del usuario y las salidas del sistema reconocibles por él.
- Incluye un plan de pruebas que indica clases de pruebas que se van a realizar
- Como resultados se espera:
 - ✓ La característica de función o rendimiento está de acuerdo con las especificaciones y se acepta
 - 🐞 Se descubre una desviación de la especificación y se crea una lista de deficiencias.
- Las pruebas de validación deben ser complementadas con una **revisión de la configuración**

Pruebas alfa y beta: muchos constructores de productos de software usan un proceso llamado prueba alfa y prueba beta para descubrir errores que al parecer **sólo el usuario final es capaz de encontrar.**

- Pruebas **alfa**: se llevan a cabo en el sitio del desarrollador por un grupo representativo de usuarios finales, en un ambiente controlado
- Pruebas **beta**: se realizan en uno o más sitios del usuario final, ya sin el desarrollador presente. Es una aplicación “en vivo” del software en un ambiente que no puede controlar el desarrollador. El cliente registra todos los problemas (reales o imaginarios) que se encuentran durante la prueba beta y los reporta.

Se la conoce también como **prueba de aceptación del cliente**



Algunos tipos adicionales de prueba (ampliaremos en próximas unidades):

- **Pruebas de recuperación:** donde forzamos sistemáticamente la falla del sistema y evaluamos la forma en que este se recupera (y el tiempo que tarda)
- **Pruebas de seguridad:** intenta verificar que los mecanismos de protección que se construyen en un sistema en realidad lo protegerán de cualquier penetración impropia. La prueba intenta penetrar al sistema. ¡Cualquier cosa vale!
- **Pruebas de esfuerzo:** ejecuta un sistema en forma que demanda recursos en cantidad, frecuencia o volumen anormales
- **Pruebas de performance:** pone a prueba el rendimiento del software en tiempo de corrida.
- **Pruebas de despliegue:** configuración en distintas plataformas, preparación de “instaladores”.

La **depuración** ocurre como consecuencia de las pruebas exitosas, pero es una de las partes más frustrantes de la programación ya que implica el reconocimiento de que se cometió un error. Puede llevar a elevada ansiedad y falta de voluntad para aceptar la posibilidad del error.

Posibles resultados durante la depuración:

- La causa se encontrará y corregirá
- La causa no se encontrará. En el último caso, la persona que realiza la depuración puede sospechar una causa, diseñar un caso de prueba para trabajar hacia la corrección del error en forma iterativa.

La **corrección del error** puede generar nuevos errores. Hay algunas preguntas simples que deben plantearse antes de hacer la “corrección”:

- **¿La causa del error se reproduce en otra parte del programa?** A veces un defecto es causado por un patrón de lógica erróneo que puede reproducirse en alguna otra parte. Seguir ese patrón nos puede ayudar a detectar más errores.
- **¿Qué “siguiente error” puede introducirse con la corrección que está a punto realizar?** Hay que prestar especial atención a partes altamente acopladas del sistema que estamos probando.
- **¿Qué debió hacerse para evitar este error desde el principio?** Esto ayuda a remover futuros errores en el producto.

Unidad 7: Administración de Proyectos de Software

La **administración del proyecto** involucra planificación, monitoreo y control del personal, procesos y acciones que ocurren, a medida que el software evoluciona desde un concepto preliminar hasta su despliegue operativo completo.

El **proyecto** debe planificarse, estimándose el esfuerzo y el cronograma necesarios para concluir las tareas.

La **comunicación** con el cliente y con otros participantes debe ocurrir de modo que los requerimientos sean comprensibles.

La administración efectiva de un proyecto de software se enfoca en las “cuatro P”:

● **Personal:** se requiere habilidad para atraer, desarrollar, motivar, organizar y conservar la fuerza de trabajo. Es necesario poner atención en áreas como:

- Comunicación y coordinación
- Ambiente de trabajo
- Capacitación
- Compensación
- Análisis y desarrollo de competencias
- Desarrollo profesional
- Desarrollo de grupos de trabajo
- Desarrollo de equipo/cultura

● **Producto:** antes de poder planear un proyecto, deben establecerse los objetivos y el ámbito del producto, considerar soluciones alternativas e identificar las restricciones técnicas y administrativas. A partir de allí logramos:

- Estimaciones razonables (y precisas) del costo
- Una valoración efectiva del riesgo
- Una descomposición realista de las tareas del proyecto
- Un calendario de proyecto manejable

Primero se deben definir los objetivos y el ámbito del producto. Los objetivos identifican las metas para el producto sin considerar cómo se lograrán estas metas (“QUE”, no “COMO”). Recién luego se plantean soluciones alternativas, teniendo en cuenta las restricciones de contexto (tiempo, presupuesto, etc).

- **Proceso:** un proceso de software proporciona el marco conceptual desde el cual puede establecerse un plan completo para el desarrollo de software. Ese plan incluye:
 - Un pequeño número de **actividades de marco** conceptual se aplica a todos los proyectos de software, sin importar su tamaño o complejidad.
 - Diferentes **tareas** (tareas, hitos, productos operativos y puntos de aseguramiento de calidad) permiten que las actividades del marco conceptual se adapten a las características del proyecto de software y a los requerimientos del equipo del proyecto.
 - Las **actividades sombrilla** (como QA, la administración de configuración del software y las mediciones) recubren el modelo de proceso. Las actividades sombrilla son independientes de cualquier actividad del marco conceptual y ocurren a lo largo del proceso.
- **Proyecto:** Los proyectos de software se planean y controlan, única forma de manejar su complejidad.

La tasa de falla de los proyectos sigue siendo alta, superior al 50%, con problemas de cumplimiento tanto de presupuesto como de cronograma.

Para evitar el fracaso del proyecto, un gerente de proyecto y los ingenieros de software que construyan el producto deben:

 - Evitar un conjunto de señales de advertencia comunes (riesgos)
 - Entender los factores de éxito cruciales que conducen a una buena administración del proyecto
 - Usar el sentido común para planificar, monitorear y controlar el proyecto
- Los principales participantes de un proyecto de software son:
 - **Usuarios finales:** las personas que finalmente van a usar el software
 - **Clientes:** que especifican los requerimientos para el software que se va a fabricar, y participantes secundarios
 - **Profesionales:** que aportan las habilidades técnicas que se necesitan para crear un producto o aplicación.
 - **Gerentes de proyecto (técnicos):** quienes deben planificar, motivar, organizar y controlar a los profesionales que hacen el trabajo de software.
 - **Gerentes ejecutivos:** definen los temas empresariales que con frecuencia tienen una influencia significativa sobre el proyecto

Los líderes de equipo idealmente debieran tener una mezcla de habilidades interpersonales, además de sus competencias técnicas. De ellos se espera:

- **Motivación:** debe alentar siempre al equipo y festejar sus logros.
- **Organización:** moldear los procesos existentes (o inventar nuevos)
- **Ideas e innovación:** alentar a las personas a crear y sentirse creativas
- **Resolución de problemas:** diagnosticar los conflictos técnicos, pensar o ayudar a pensar una solución, y aplicar lecciones aprendidas
- **Buena administración:** asumir el control cuando sea necesario
- **Logros:** recompensar la iniciativa y el logro
- **Influencia y construcción de equipo:** comprender las señales y reaccionar ante necesidades de las personas

Para lograr un **equipo de alto rendimiento**, debemos asegurar que:

- Los miembros del equipo se tienen confianza entre sí.
- La distribución de habilidades debe ser adecuada para el problema
- Si no se pueden solucionar disconformidades de algún miembro del equipo, es posible que tenga que excluirlo a fin de mantener la cohesión del equipo.
- El gerente debe ayudar a crear cohesión de equipo
- No caigamos en alguno de estos escenarios:
 - Una atmósfera de trabajo frenético
 - Alta frustración que causa fricción entre los miembros del equipo
 - Un proceso de software fragmentado o pobemente coordinado
 - Una definición poco clara de los roles en el equipo de software
 - Continua y repetida exposición al fracaso.

Un **equipo ágil** es el antídoto a muchos de los problemas del trabajo en un proyecto de software. Se alienta la satisfacción del cliente y la entrega incremental temprana del software, pequeños equipos de proyecto enormemente motivados, métodos informales, mínimos productos operativos y simplicidad de desarrollo global.

Estos equipos son normalmente autogestionados, con autonomía para tomar las decisiones administrativas y técnicas del proyecto necesarias.

Para lograr esto, un equipo ágil puede realizar reuniones grupales diarias para coordinar y sincronizar el trabajo que debe realizarse en ese día.

Unidad 7: El Producto

Al iniciar un proyecto nos piden estimaciones precisas y un plan completo. Pero no hay información sólida de que valerse y requerimientos cambiantes.

¿Qué hacemos?

Una salida es examinar el **producto**: establecer y acotar el ámbito del mismo.

- Determinar el ámbito del software: contexto, objetivo, principales funciones
 - Objetos de entrada y de salida
 - Transformaciones necesarias
 - Usuarios simultáneos
 - Tiempo de respuesta máximo
- Descomponer el problema (divide y reinarás): las funciones del software se evalúan y refinan para tener más detalle **antes** de comenzar la estimación

Unidad 7: El Proceso

Las actividades del marco conceptual son aplicables a todos los proyectos de software. El problema es seleccionar el **modelo de proceso** que sea adecuado para el software que el equipo del proyecto intenta construir:

- Para los clientes que solicitaron el producto
- Para el personal que hará el trabajo
- Para las características del producto en sí
- Para el entorno de proyecto donde trabaja el equipo de software

La planificación del proyecto incluye la fusión de producto y proceso. Cada función debe pasar a través del conjunto de actividades de marco conceptual.

Ejemplo: **comunicación, planificación, modelado, construcción y despliegue**.

Unidad 7: Fusión de Producto y Proceso

El gerente de proyecto debe estimar los requerimientos de recursos para cada celda de la matriz, fechas de inicio y término de las tareas asociadas con cada celda, y los productos operativos que se van a producir.

ACTIVIDADES COMUNES DEL MARCO CONCEPTUAL DEL PROCESO		comunicación	planificación	modelado	construcción	despliegue
Tareas de la ingeniería de software						
Funciones del producto						
Entrada de texto						
Edición y formato						
Edición automática de copia						
Capacidad de plantilla de página						
Indexado automático y T de C automática						
Gestión de archivo						
Producción de documento						

Unidad 7: El Proyecto

Para administrar un **proyecto** de software exitoso, se debe comprender **qué puede salir mal**, de modo que los problemas pueden evitarse. Esto se llama **mitigación de riesgos**.

- Top 10 de riesgos típicos en la mayoría de los proyectos:
 - El equipo no entiende las necesidades del cliente.
 - El ámbito del producto está pobremente definido
 - Los cambios se gestionan pobremente
 - Cambia la tecnología elegida.
 - Las necesidades de negocio cambian
 - Las fechas límite son irreales.
 - Los usuarios son resistentes.
 - El equipo del proyecto carece de personal con skills adecuadas.
 - Pérdida de patrocinio
 - Los gerentes y el equipo no tienen buenas prácticas
 - No se tienen en cuenta las lecciones aprendidas.

Unidad 7: El Proyecto - 5 enfoques de sentido común

- **Comenzar con el pie derecho**
 - Trabajar duro para entender el problema que debe resolverse
 - Establecer objetivos y expectativas realistas
 - Construir el equipo correcto y darle autonomía, autoridad y herramientas.
- **Mantener la cantidad de movimiento**
 - Proporcionar incentivos para mantener la rotación de personal en un mínimo
 - El equipo debe enfatizar la calidad en cada tarea que realice
 - Mantener al gerente ejecutivo lo más lejos posible del equipo
- **Siga la pista al progreso:** tener métricas confiables y valorar el progreso
- **Tome decisiones inteligentes**
 - Simplificar cada vez que se pueda. Utilizar componentes e interfaces estándar
 - Asignar más tiempo de lo estimado a tareas complejas o riesgosas.
- **Realice un análisis postmortem:** recopilar feedback y documentar lecciones aprendidas

Unidad 7: El Proyecto y el principio W5-HH

Siete preguntas que conducen a una definición de las características clave del proyecto y al plan de proyecto resultante:

- ¿Por qué (**why**) se desarrollará el sistema?
- ¿Qué (**what**) se hará?
- ¿Cuándo (**when**) se hará?
- ¿Quién (**who**) es responsable de cada función?
- ¿Dónde (**where**) se ubicará en la organización?
- ¿Cómo (**how**) se hará el trabajo, técnica y organizativamente?
- ¿Cuánto (**how much**) se necesita de cada recurso?

Unidad 7: El Proyecto

Prácticas cruciales:

- Administración del proyecto basada en métrica
- Estimación empírica de costo y calendario
- Rastreo del valor ganado
- Rastreo de defecto contra metas de calidad
- Administración consciente del personal
- Gestión eficiente de la matriz de interesados
- Administración efectiva de los riesgos

Unidad 7: Métricas de proceso y de proyecto

La **medición cuantitativa** permite comprender mejor el proceso y el proyecto, al darnos un mecanismo de evaluación **objetiva**. Si no se mide, el juicio puede basarse solamente en la evaluación **subjetiva**

- **En el proceso:** se mide pensando en mejoras continuas
- **En el proyecto:** ayuda en la estimación, control de calidad, valoración de productividad y control de proyecto.
- **En el producto:** ayuda a valorar la calidad del mismo

Se debe definir un conjunto limitado de medidas de proceso, proyecto y producto, que son fáciles de recopilar. Se las normalizan usando métricas de tamaño o de función y el resultado se compara con promedios anteriores para proyectos similares.

Vemos tendencias y sacamos conclusiones.

Unidad 7: Métricas de proceso

Las métricas de proceso de software aportan beneficios significativos si se trabaja para mejorar su nivel global de madurez de proceso. Sin embargo, éstas pueden tener mal uso, lo que crea más problemas de los que resuelven.

- Sentido común y sensibilidad organizacional al interpretar datos de métricas.
- Dar feedback regular a los individuos y equipos que recopilan medidas y métricas.
- No usar métricas para valorar a los individuos.
- Consensuar metas y métricas claras que se usarán para lograr las primeras.
- Nunca usar métricas para amenazar a los individuos o a los equipos
- Es positivo si una métrica muestra un problema. Es la oportunidad de mejorar.
- No obsesionarse con una sola métrica ni excluir otras importantes

Unidad 7: Métricas de proyecto

Las métricas de proyecto tienen un uso más táctico: permiten adaptar el flujo de trabajo del proyecto y las actividades técnicas. Son útiles para adaptar el flujo de trabajo del proyecto y las actividades técnicas.

- En la etapa de **estimación**, miramos métricas de proyectos anteriores
- Durante la **ejecución** tenemos métricas para comparar **actual vs forecast**, ayudando a monitorear el avance del proyecto

La intención de las métricas de proyecto es doble:

- Se usan para gestionar el calendario y ajustarlo, evitar demoras y mitigar potenciales problemas y riesgos.
- Se usan para valorar la calidad del producto y modificar el enfoque técnico para mejorar la calidad.

Unidad 7: Métricas de calidad del Software

Un gerente de proyecto también debe evaluar la calidad conforme avanza el proyecto.

Algunas mediciones de calidad son:

- **Exactitud:** grado en el cual el software realiza la función requerida, y se define en función de los errores encontrados en producción, en un periodo dado.
- **Capacidad de mantenimiento:** facilidad con la que un programa puede corregirse si se encuentra un error.
- **Integridad:** mide la habilidad del sistema para resistir ataques a su seguridad. Se basa en el concepto de amenaza (probabilidad) y la seguridad (capacidad para repeler el ataque)

- **Usabilidad:** es un intento por cuantificar la facilidad de uso, usualmente ayudado con encuestas a usuarios finales.

Unidad 7: Métricas de calidad

Buenas métricas de calidad para código en desarrollo:

- Porcentaje de código que ha sido revisado (idealmente 100%).
- Porcentaje del código que tiene cobertura de tests. No menor a 70%
- $(\text{Bugs de Desarrollo} / \text{Bugs Totales}) \times 100$
- $(\text{Bugs corregidos} / \text{Bugs Totales}) \times 100$
- $(\text{Bugs sin solución o Next version} / \text{Bugs Totales}) \times 100$
- Nro. de errores aceptados (sin solución)
- Nro. De Errores Stoppers identificados.