ISW - Unidad 1 Ingeniería de software en contexto

Introducción a la Ingeniería de Software

¿Qué es el Software?

- **Definición**: conjunto de programas **+ documentación asociada** (manuales de usuario, guías técnicas, configuraciones, diagramas, etc.) y **herramientas utilizadas para su construcción**.
- **Visión por niveles de abstracción**:
  - **Requerimientos** → visión más abstracta (qué debe hacer).
  - **Diseño** → visión intermedia (cómo se organiza).
  - **Código ejecutable** → nivel más detallado (implementación).
- **Ejemplo**: un sistema bancario no solo incluye la aplicación móvil, sino también la documentación para los usuarios, los manuales técnicos, los archivos de configuración de seguridad y los scripts de base de datos.

Ingeniería de Software (IS)

- **Definición**: disciplina que cubre todas las etapas de la producción de software: **desde la especificación inicial hasta el mantenimiento** del sistema en operación.
- **Por qué es importante**:
  - Los clientes exigen **sistemas confiables, rápidos y económicos**.
  - Usar buenas prácticas **reduce costos a largo plazo**, porque el mayor gasto en software proviene de su **mantenimiento** (cambios, correcciones, mejoras).
- **Principios básicos**:
  1. Comprender primero el problema y las necesidades del cliente antes de diseñar.
  2. El diseño es clave: un buen diseño → software de calidad + fácil mantenimiento.
  3. La disciplina nace como respuesta a la **crisis del software** (ineficiencia, sobrecostos, proyectos fallidos).

Estado actual y antecedentes: la Crisis del Software

- **Origen histórico**:
  - Termino introducido en la conferencia de la OTAN (1968, Friedrich Bauer).
  - También mencionado por Edsger Dijkstra (“El humilde programador”).
  - Problema: incapacidad de producir software confiable, entendible y verificable a gran escala.
- **Causas principales**:
  - Avance del **hardware** sin métodos equivalentes en software.
  - Subestimación de la **complejidad creciente**.
  - Cambios constantes de requisitos y ausencia de disciplina formal.
  - Producto intangible → difícil medir calidad y progreso.
- **Hoy en día**:

    - **Demandas crecientes**: se requieren sistemas más grandes y entregas rápidas. Los métodos tradicionales no son suficientes → surgen **Agile, DevOps, CI/CD**.
    - **Bajas expectativas**: muchas empresas aún no aplican IS rigurosa → generan software caro y poco confiable, aceptado solo porque "funciona".
- **Ejemplo moderno**:
    - Aplicaciones móviles que salen al mercado con errores de seguridad o baja performance porque se priorizó velocidad sobre calidad.

Disciplinas que conforman la Ingeniería de Software

- **Disciplinas técnicas** (directamente ligadas al producto):
    - Requerimientos, análisis, diseño, implementación, pruebas, despliegue, documentación y capacitación.
- **Disciplinas de gestión** (enfocadas en el proyecto):
    - Planificación, monitoreo, control → metodologías como **Lean, Agile, Scrum, Kanban, PMBOK**.
- **Disciplinas de soporte** (transversales, aseguran integridad y calidad):
    - Gestión de configuración (SCM), métricas, aseguramiento de calidad (QA).

Ejemplos de proyectos de software fallidos y exitosos

Proyectos de Software Fallidos

1. FBI Virtual Case File (VCF) – 2000s

- **Objetivo**: reemplazar el sistema obsoleto de gestión de casos del FBI.
- **Presupuesto inicial**: ~170 millones de USD.
- **Problemas**:
    - Requisitos cambiantes y poco definidos.
    - Excesiva burocracia en la gestión del proyecto.
    - Falta de comunicación entre usuarios finales (agentes) y desarrolladores.
    - Tecnología poco adecuada y difícil de mantener.
- **Resultado**: cancelado en 2005, con pérdida millonaria y años de retraso en la modernización del FBI.
- **Lección**: la gestión de requisitos y la validación continua con usuarios son críticas.

2. Sistema de gestión de salud del Estado de California (2013) – Covered California

- **Objetivo**: implementar un portal web para la contratación de seguros médicos bajo el *Affordable Care Act*.
- **Problemas**:
    - Mala integración entre módulos.

    - Escasa validación de la usabilidad → usuarios no podían completar procesos.
    - Lanzamiento apresurado por presión política.
- **Resultado**: sistema colapsó en su lanzamiento, miles de usuarios no pudieron registrarse. Se requirió rehacer gran parte del software.
- **Lección**: nunca subestimar pruebas de integración y pruebas de carga, especialmente en software de misión crítica.

3. Proyecto *NHS National Programme for IT* (Reino Unido, 2002–2011)

- **Objetivo**: informatizar todo el sistema de salud público británico (historias clínicas electrónicas).
- **Presupuesto inicial**: ~6 mil millones de libras.
- **Problemas**:
    - Requisitos extremadamente ambiciosos.
    - Mala coordinación entre cientos de hospitales y proveedores.
    - Escasa flexibilidad del modelo de gestión (muy burocrático).
- **Resultado**: cancelado en 2011, tras gastar más de 10 mil millones de libras.
- **Lección**: proyectos “mega-monolíticos” suelen fracasar; mejor adoptar estrategias **incrementales y modulares**.

4. Denver Airport Baggage Handling System (1995)

- **Objetivo**: sistema automatizado de gestión de equipaje en el nuevo aeropuerto de Denver.
- **Problemas**:
    - Subestimación de la complejidad técnica.
    - Desarrollo paralelo al diseño del aeropuerto (cambios constantes).
    - Insuficiente tiempo de pruebas.
- **Resultado**: retraso de 16 meses en la inauguración, sobrecostos de ~560 millones USD. Finalmente, el sistema automatizado fue desactivado.
- **Lección**: el software crítico necesita **planificación realista** y pruebas extensivas antes de integrarse en infraestructura física.

Proyectos de Software Exitosos

1. Sistema Operativo Linux (1991–actualidad)

- **Características**:
    - Iniciado por Linus Torvalds como un proyecto abierto.
    - Modelo colaborativo de desarrollo (miles de contribuyentes).
    - Modularidad y licencias libres (GPL) impulsaron su adopción.
- **Éxitos**:
    - Hoy es la base de servidores, supercomputadoras, Android y sistemas embebidos.
- **Claves del éxito**:
    - **Transparencia**, **colaboración distribuida** y **arquitectura modular**.

2. Gmail (Google, 2004)

- **Innovaciones**:
  - Interfaz basada en **AJAX** (revolucionó el correo web).
  - Gran capacidad de almacenamiento (1 GB, cuando Hotmail daba 2 MB).
  - Filtros automáticos y búsqueda interna.
- **Claves del éxito**:
  - Lanzamiento en **beta controlada** para validar con usuarios.
  - Iteraciones ágiles y mejoras constantes.
- **Resultado**: redefinió el mercado del correo electrónico.

3. SABRE (Sistema de Reservas de American Airlines, 1960s)

- **Características**:
  - Primer gran sistema de reservas en línea de la aviación.
  - Permitía consultar y reservar vuelos en tiempo real.
- **Claves del éxito**:
  - Fuerte integración negocio–tecnología.
  - Escalabilidad para soportar millones de transacciones.
- **Impacto**: se convirtió en estándar de la industria, base de sistemas modernos de reservas (Amadeus, Galileo).

4. Spotify (2008–actualidad)

- **Innovaciones**:
  - Streaming musical con tiempos de carga casi instantáneos.
  - Recomendaciones personalizadas (algoritmos de machine learning).
  - Modelo freemium exitoso.
- **Claves del éxito**:
  - Arquitectura escalable en la nube.
  - Desarrollo ágil + squads multidisciplinarios (referencia en gestión ágil).
- **Resultado**: líder mundial en streaming musical.

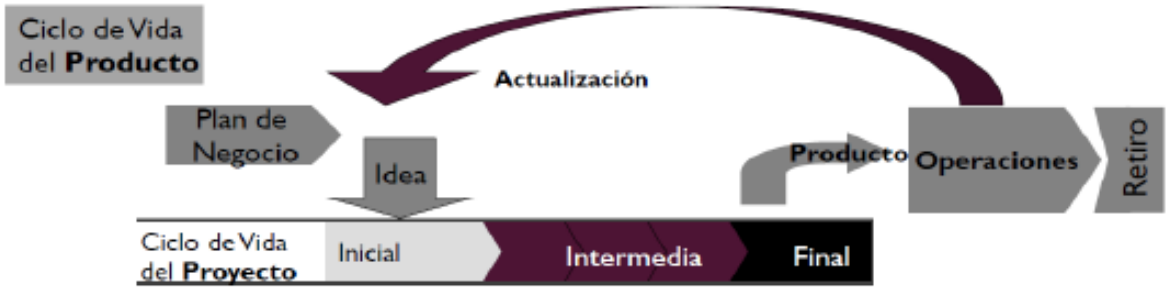
Ciclos de vida (Modelos de Proceso) y su influencia

- **Definición**: Representación simplificada de cómo progresa un proyecto o producto de software a lo largo del tiempo.
- **Elementos clave**:
  - Actividades (requerimientos, diseño, construcción, pruebas, etc.).
  - Flujo de trabajo (orden, dependencias, iteraciones).
  - Criterios para pasar de una fase a otra.

Relación ciclo de vida del proyecto vs. ciclo de vida del producto

- **Proyecto** → dura mientras se desarrolla el software.
- **Producto** → dura hasta que deja de usarse en el mercado.

- Un mismo producto puede tener **múltiples proyectos** (ej. nuevas versiones de Windows: XP, 7, 10, 11).



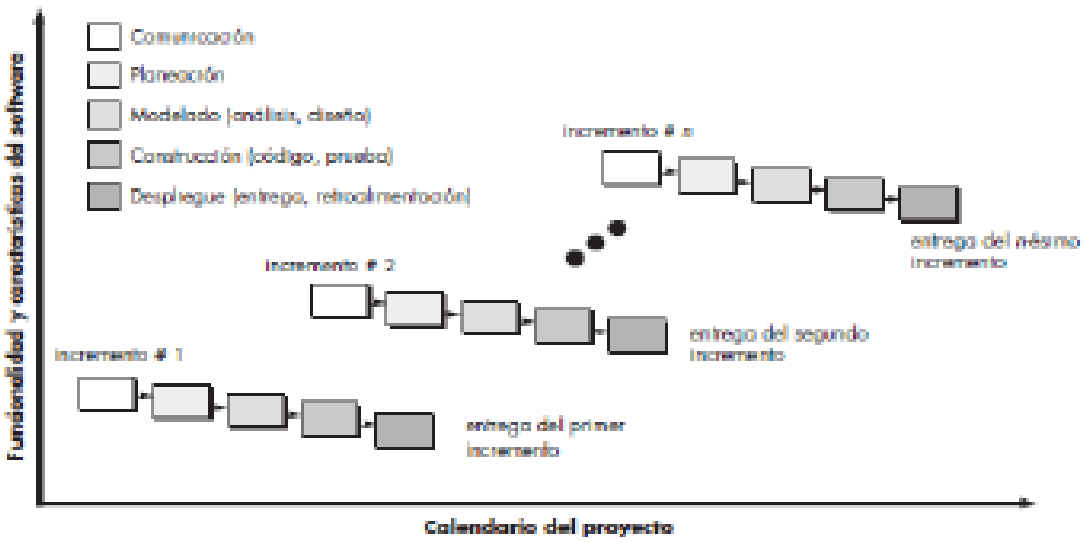
Clasificación de Ciclos de Vida:

Modelo Secuencial (Cascada)

- **Características**:
  - Flujo lineal: requisitos → diseño → implementación → pruebas → despliegue.
  - Avance solo cuando se termina la fase anterior.
  - Muy dependiente de documentación y planificación inicial.
- **Ventajas**:
  - Buena para proyectos con requisitos **claros y estables**.
  - Facilita gestión común y planificación inicial.
  - Documentación abundante → trazabilidad.
- **Desventajas**:
  - Los requisitos rara vez están claros al 100% desde el inicio.
  - El cliente ve resultados muy tarde.
  - Los cambios son costosos.
  - Poca tolerancia a la incertidumbre.
- **Ejemplo**:

- Desarrollo de software para sistemas **embebidos en aviones o naves espaciales**, donde los requisitos son estables, se prioriza la documentación y los cambios son mínimos.

Modelo Iterativo/Incremental



- **Definición**:
  Se construye el software en **múltiples versiones (incrementos)**, cada una entregando una parte funcional del sistema.
  - **Iteración** = ciclo de especificación → desarrollo → validación.
  - **Incremento** = versión del software con más funcionalidades que la anterior.
- **Uso típico**: metodologías ágiles (Scrum, XP, RUP en su versión iterativa).
- **Ejemplo práctico**: una aplicación de e-commerce puede iniciar con solo la **búsqueda de productos y carrito**, en la siguiente iteración agregar **pasarela de pagos**, y en otra **sistema de reseñas**.

Ventajas frente al modelo secuencial

1. **Adaptación al cambio**: los requisitos pueden ajustarse a medida que se avanza.
2. **Retroalimentación temprana**: el cliente participa en cada entrega y valida funcionalidades.
3. **Entrega más rápida de valor**: el cliente puede usar versiones útiles antes de tener el sistema completo.
4. **Adecuado para entornos cambiantes**: ideal cuando los requisitos evolucionan.
5. **Especificación, desarrollo y validación están integrados**, no separados.

Dificultades administrativas

- El proceso puede volverse **poco visible**: difícil medir avance con exactitud.
- **Alto costo de documentación**: cada incremento requiere actualizar manuales y especificaciones.

- La **arquitectura puede degradarse** si no se controla el diseño global (deuda técnica).

Modelo Recursivo (Ejemplo: Espiral de Boehm)

- **Definición**:
  Se utiliza en proyectos grandes y complejos, donde el **riesgo** es un factor central. El desarrollo avanza en ciclos (vueltas en espiral), en los que se:
  1. **Definen objetivos**.
  2. **Identifican y mitigan riesgos**.
  3. **Desarrolla y prueba una versión parcial/prototipo**.
  4. **Planifica la siguiente iteración**.
- **Metodología**: el proyecto se divide en **mini-proyectos**, cada uno enfocado en resolver los riesgos más críticos hasta eliminarlos.
- **Producto típico**: **prototipos sucesivos** que se refinan en cada iteración.
- **Ejemplo práctico**: en un sistema de control aéreo, se comienza con un prototipo básico que gestiona vuelos en una sola región → luego se agrega soporte para múltiples aeropuertos → después escalabilidad para vuelos internacionales.

Ventajas

- Permite **gestionar la incertidumbre** y reducir riesgos gradualmente.
- Fomenta el **reuso** de componentes probados en otros sistemas similares.
- Adecuado para **proyectos de misión crítica** (defensa, aeroespacial, banca).

Dificultades administrativas

- Puede resultar **más costoso** en tiempo y dinero que otros modelos.
- Riesgo de que la **tecnología usada quede obsoleta** si los ciclos son muy largos.
- La **documentación y mantenimiento** pueden ser deficientes si no se controlan.
- El cliente obtiene una versión final del producto **recién al final del ciclo completo**.

Administración de Proyectos según el Ciclo de Vida

- **Impacto directo**: la **gestión de proyectos** depende fuertemente del ciclo de vida elegido.
  - Ejemplo: en cascada se puede planificar todo desde el inicio; en iterativo la planificación es **adaptativa** e incremental.
- **Si los requisitos son estables** → se prefiere un modelo **secuencial**.
- **Si los requisitos cambian con frecuencia** → conviene un modelo **iterativo o ágil**.
- **Si el proyecto es de gran escala con alto riesgo** → se recomienda un modelo **recursivo/espiral**.
- **Consecuencia práctica**:
  - En cascada → gestión tradicional, cronogramas rígidos, énfasis en documentación.

- En iterativo/ágil → gestión adaptativa, sprints, feedback constante, reuniones frecuentes.
- En espiral → gestión basada en **evaluación continua de riesgos** y prototipos.

Procesos de Desarrollo de Software: Empíricos vs. Definidos

¿Qué es un proceso de desarrollo de software?

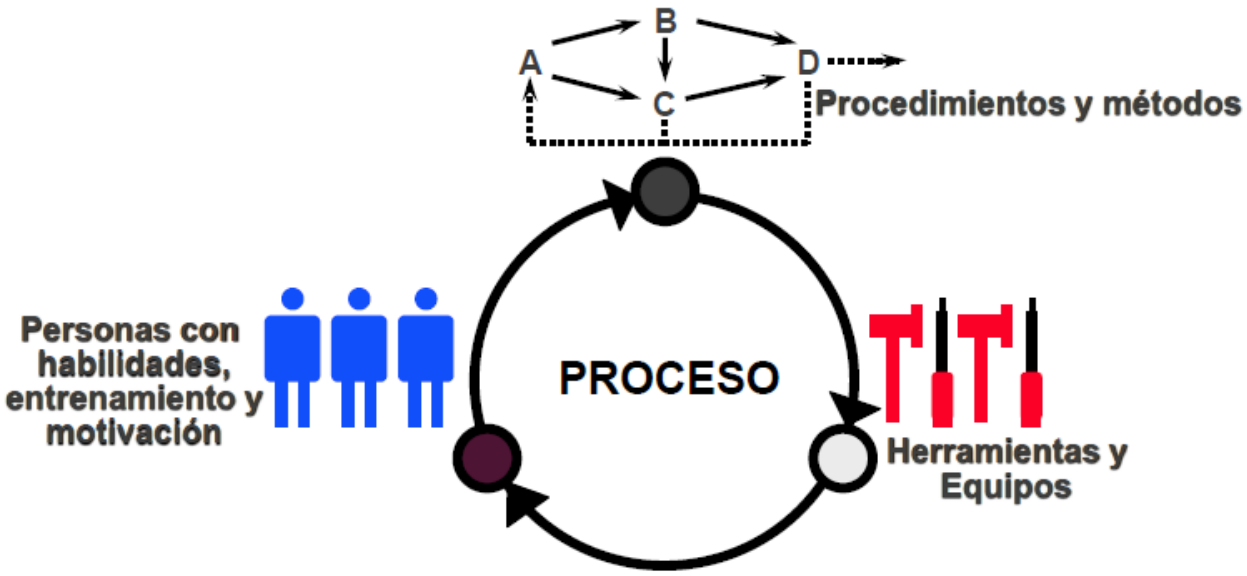
- **Definición IEEE**: un proceso es una secuencia de pasos ejecutados para un propósito dado.
- **Proceso de software**: conjunto de **actividades, métodos, prácticas y transformaciones** usadas para desarrollar o mantener software y sus artefactos (documentación, configuraciones, pruebas, manuales).

Actividades fundamentales:

1. **Especificación** → definir con el cliente qué debe hacer el software y bajo qué restricciones.
2. **Desarrollo** → diseño y programación.
3. **Validación** → verificar que el software cumple con lo que el cliente quiere.
4. **Evolución** → modificar y adaptar el software a necesidades cambiantes del negocio o mercado.

Factores determinantes del proceso:

1. **Procedimientos y métodos** → reglas, actividades y documentación que aseguran transparencia.
2. **Personas motivadas y capacitadas** → el software es intensivo en talento humano.
3. **Herramientas y equipos** → soporte tecnológico y automatización para aumentar eficiencia (ej: CI/CD, testing automático, IDEs).



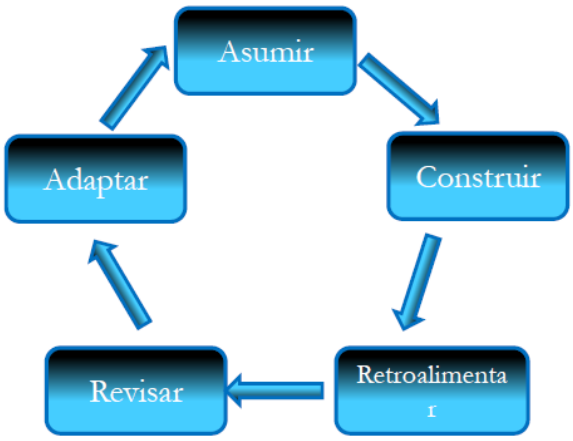
Procesos Definidos

- **Características principales**:
  - **Deterministas**: se espera que con las mismas entradas se logren los mismos resultados.
  - Inspirados en **líneas de producción industriales**.
  - Basados en **predictibilidad y repetibilidad**: estimar con precisión tiempos, costos y riesgos.
  - Fuerte énfasis en **documentación y control administrativo**.

- **Ventajas**:
  - Útiles en entornos donde la **seguridad, estabilidad y trazabilidad** son críticos (ej.: sistemas aeroespaciales, médicos, banca).
  - Facilitan la **auditoría y certificación** (ISO, CMMI).
- **Limitaciones**:
  - En la práctica, es imposible obtener siempre el mismo resultado, ya que el software depende del **contexto, equipo y momento histórico**.
  - La **burocracia puede superar al desarrollo real**: se documenta más de lo que se programa.
- **Ejemplo real**:
  - **Sistema de control de tráfico aéreo europeo (Eurocontrol)**: usa procesos altamente definidos porque el margen de error debe ser **cero**.
  - Normativas como **DO-178C** (aviónica) o **ISO 26262** (automoción) exigen procesos definidos para certificar software crítico.

Procesos Empíricos

- **Características principales**:
  - No son deterministas: la misma entrada puede producir distintos resultados según el contexto.
  - Basados en **experiencia, retroalimentación y adaptación continua**.
  - Utilizan **ciclos cortos de inspección y adaptación** → permiten ajustar mejor el proceso en sistemas complejos y cambiantes.
  - Requieren **transparencia total**: todos los miembros del equipo deben conocer estado, avances y problemas.



- **Ventajas**:
  - Excelentes para proyectos con **incertidumbre** o requisitos en constante evolución.
  - Fomentan **creatividad, innovación y rapidez de respuesta**.
  - Promueven la mejora continua al compartir aprendizajes dentro del equipo.
- **Limitaciones**:
  - Resultados no siempre extrapolables → lo que funcionó en un proyecto no necesariamente funciona en otro.
  - Menos previsibles en costo y tiempo → requieren confianza del cliente.
- **Ejemplo real**:
  - **Scrum**: metodología ágil basada en procesos empíricos. Se planifica por sprints cortos, se revisa el avance y se ajusta el plan.
  - **Spotify model**: organización de equipos en squads y tribus que adaptan continuamente prácticas a su contexto.

Comparación directa

Aspecto	Procesos Definidos	Procesos Empíricos
Base	Repetibilidad y predictibilidad	Experiencia, inspección y adaptación
Enfoque	Control documental y procedimientos claros	Ciclos cortos, feedback continuo

Aspecto	Procesos Definidos	Procesos Empíricos
Aplicación típica	Sistemas críticos, entornos regulados	Sistemas empresariales, startups, innovación
Ventaja principal	Estabilidad y trazabilidad	Flexibilidad y adaptación al cambio
Riesgo	Burocracia excesiva	Incertidumbre en tiempo y costo
Ejemplo	Software de aviónica, banca	Scrum, desarrollo de apps móviles

Ciclos de Vida y Procesos de Desarrollo de Software

Ventajas y desventajas de los ciclos de vida

1. **Modelo Secuencial (Cascada)**
   - Ventajas:
     - Simple de entender y administrar.
     - Adecuado cuando los requisitos son **claros y estables**.
     - Facilita la planificación inicial y la documentación completa.
     - Útil para equipos poco experimentados porque aporta estructura.
   - Desventajas:
     - Poco flexible ante cambios.
     - El cliente recibe el producto funcional al final, lo que retrasa la validación real.
     - Errores en etapas iniciales se arrastran y su corrección es costosa.
2. **Modelo Iterativo-Incremental**
   - Ventajas:
     - Entregas tempranas y parciales (el cliente empieza a usar antes el producto).
     - Se adapta a **requerimientos cambiantes**.
     - Favorece la retroalimentación y la participación del cliente.
   - Desventajas:
     - Más difícil de administrar (muchas iteraciones).
     - Riesgo de degradación arquitectónica si no se controla bien.
     - Mayor carga documental y de coordinación.
3. **Modelo Recursivo (Ej.: Espiral)**
   - Ventajas:
     - Ideal para proyectos grandes y complejos.
     - Orientado a la **gestión de riesgos**.
     - Se obtiene aprendizaje en cada iteración y se pueden hacer prototipos.
   - Desventajas:
     - Puede ser costoso en tiempo y dinero.
     - Difícil definir objetivos claros en cada iteración.
     - El producto final suele obtenerse en etapas avanzadas.

Criterios para la elección de un ciclo de vida

1. **Modelo Secuencial (Cascada) →**
   Cuando los **requerimientos son claros y exhaustivos**, la incertidumbre es baja, y el cliente puede esperar hasta el final para recibir el producto.

   Ejemplo: desarrollo de software para facturación en una empresa con procesos estables.

2. **Modelo Iterativo-Incremental →**
   Si los requisitos son **volátiles**, hay incertidumbre y el cliente quiere resultados **rápidos**, con entregas parciales y funcionalidad básica primero.

   Ejemplo: aplicación móvil donde los usuarios piden mejoras continuas.

3. **Modelo Recursivo (Espiral) →**
   Si el objetivo es **minimizar riesgos** en sistemas complejos y de gran escala, con posibilidad de prototipar y validar continuamente.

   Ejemplo: software para control de tráfico aéreo, donde se prueban alternativas en cada fase para reducir riesgos.

Diferentes modelos de ciclo de vida

1. **Code and Fix**
   - Sin especificación ni diseño formal; se programa directamente y se corrige hasta que el cliente quede conforme.
   - Ventajas: rápido inicio, pocos recursos necesarios, útil en prototipos desechables.
   - Desventajas: alto costo de mantenimiento, falta de control, difícil de escalar.
   - Ejemplo: un prototipo rápido de un videojuego indie.
2. **Modelo en Cascada Puro**
   - Fases secuenciales con revisiones al final de cada una.
   - Permite detectar errores tempranos y es estructurado.
   - Muy rígido, dependiente de requisitos iniciales completos.
   - Ejemplo: software para un banco donde los procesos están bien definidos.
3. **Modelo en Cascada con fases solapadas**
   - Las fases se superponen, reduciendo documentación y aumentando la velocidad.
   - Más difícil hacer seguimiento.
   - Ejemplo: desarrollo de sistemas ERP donde algunas fases pueden ejecutarse en paralelo.
4. **Modelo de Entrega por Etapas**
   - El cliente recibe versiones parciales en intervalos planificados.
   - Da sensación de progreso y valor temprano.
   - Ejemplo: una app SaaS que va liberando módulos funcionales.
5. **Modelo en Cascada con Retroalimentación**
   - Permite volver a etapas previas, aunque con **alto costo**.

- □ Ejemplo: un sistema contable en el que se vuelve atrás para ajustar requerimientos legales.

6. **Modelo en Cascada con Subproyectos**
   - Divide el proyecto en módulos después del diseño arquitectónico.
   - □ Ejemplo: un ERP dividido en subproyectos como inventario, finanzas, RRHH.
7. **Modelo en Espiral**
   - Ciclos iterativos con análisis de riesgos al inicio de cada fase.
   - □ Excelente para proyectos de larga duración y alto riesgo.
   - □ Identificar riesgos puede ser más costoso que el propio desarrollo.
   - □ Ejemplo: software militar o de telecomunicaciones.
8. **Modelo Evolutivo**
   - Se fija el tiempo o el alcance, mientras el otro se adapta.
   - □ Útil con requisitos poco claros.
   - □ Depende de que el diseño permita modificar fácilmente el sistema.
   - □ Ejemplo: desarrollo de un marketplace donde el alcance crece según necesidades.
9. **Diseño para Cronograma**
   - Se prioriza entregar un producto en la fecha acordada, aunque no sea la versión final.
   - □ Ejemplo: lanzamiento de software antes de una feria tecnológica.
10. **Diseño para Herramientas**
    - Se construye solo con lo soportado por herramientas disponibles.
    - □ Ejemplo: desarrollo rápido con frameworks preexistentes para cumplir con deadlines ajustados.
11. **Prototipación Evolutiva**
    - Requisitos cambiantes, baja documentación, desarrollo guiado por retroalimentación.
    - □ Permite visualizar rápido el producto.
    - □ Difícil planificar releases con exactitud.
    - □ Ejemplo: app de salud en la que los médicos van ajustando funcionalidades.

Componentes de un Proyecto de Sistemas de Información

Proyecto de Software

Un **proyecto de software** es un **esfuerzo temporal**, organizado con recursos humanos y técnicos, para crear un **producto, servicio o resultado único**.

□ **Características**

1. **Resultado único**

- Aunque se usen tecnologías similares, cada proyecto es irrepetible.
- □ Ejemplo: dos sistemas de gestión para hospitales distintos tendrán módulos similares (turnos, historias clínicas), pero **nunca serán idénticos** porque difieren en regulaciones, procesos internos y presupuestos.

2. **Orientado a objetivos claros**
   - No ambiguos → deben guiar al equipo.
   - Medibles → permiten evaluar avances.
   - Alcanzables → factibles de cumplir con los recursos y capacidades.
   - □ Ejemplo: *"Desarrollar una app móvil para reservas de hotel que soporte al menos 10.000 usuarios concurrentes en 6 meses"*.
3. **Duración limitada en el tiempo**
   - Tiene inicio y fin (a diferencia de una línea de producción continua).
   - □ Ejemplo: el desarrollo de un sistema de pagos online dura 12 meses; al finalizar, los recursos se reasignan.
4. **Conjunto de tareas interrelacionadas**
   - Se definen dependencias, asignación de recursos y cronograma.
   - □ Ejemplo: no se puede programar un módulo de base de datos antes de definir el modelo entidad-relación.

Administración de Proyectos de Software

La **gestión de proyectos** consiste en organizar personas y recursos para cumplir con:

- Tiempo acordado.
- Presupuesto definido.
- Requerimientos del cliente.

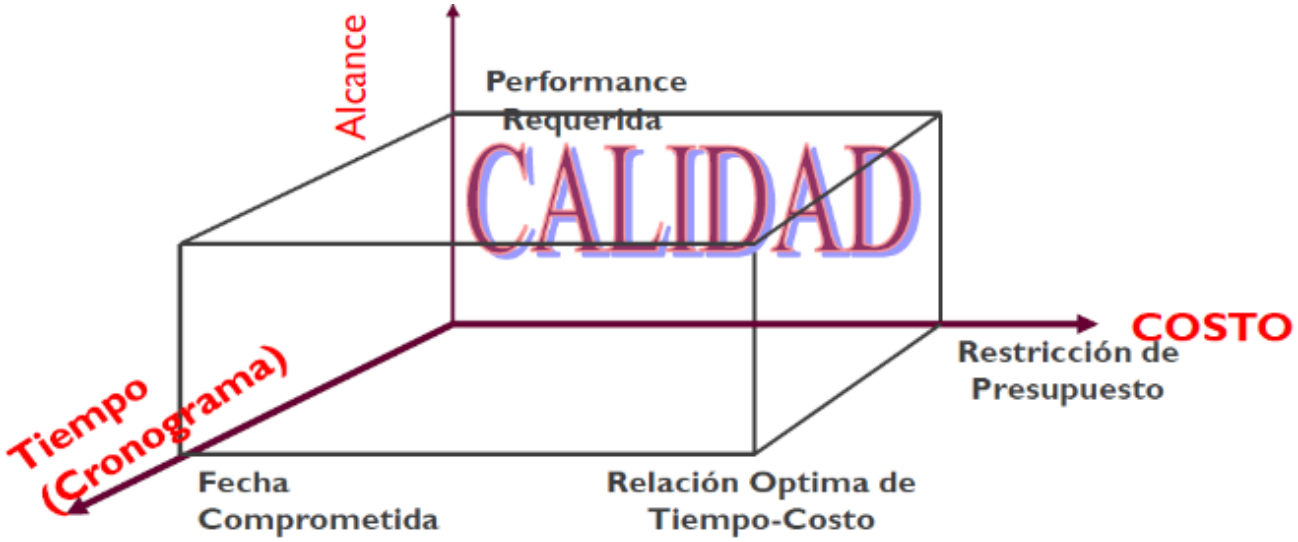
□ Objetivos principales

1. Entregar el software en el **tiempo acordado**.
2. Mantener los **costos dentro del presupuesto**.
3. Asegurar que el software cumpla con las **expectativas del cliente**.
4. Mantener un **equipo motivado y productivo**.

⚠□ Complejidades en proyectos de software

1. **Producto intangible**
   - No se ve el progreso como en la construcción de un edificio.
   - □ Ejemplo: un cliente puede impacientarse porque no percibe "avances visibles" hasta que se entregan prototipos.
2. **Excepcionalidad**
   - Cada proyecto es único, no siempre se pueden reutilizar aprendizajes de uno a otro.
   - □ Ejemplo: un sistema bancario en la nube puede no servir como referencia directa para un sistema de salud por diferencias regulatorias.
3. **Procesos variables**
   - Dependen de la organización, el equipo y el contexto.
   - □ Ejemplo: una startup adopta metodologías ágiles (Scrum), mientras que una multinacional puede usar procesos más definidos (CMMI, ISO).

La Triple Restricción (Enfoque Tradicional)



Es el **equilibrio entre tres variables clave**:

1. **Alcance**: funcionalidades y requisitos del producto.
2. **Tiempo**: plazo de entrega.
3. **Costo**: presupuesto y recursos.

➡ Estas tres variables afectan directamente la **calidad** del software.

 Principios

- No se pueden fijar **simultáneamente las 3 variables**:
  - Se puede fijar **alcance + costo**, pero entonces el **tiempo** debe ajustarse.
  - Se puede fijar **alcance + tiempo**, pero el **costo** aumentará.
- La **calidad no debe negociarse**: un software sin testing puede cumplir plazo y costo, pero será un fracaso.

 Ejemplos prácticos

1. **Aumento de alcance**
   - Cliente pide nuevas funcionalidades en medio del proyecto.
   - Resultado: se requiere más tiempo y presupuesto.
   - Si no se ajusta ninguna, se compromete la calidad (testing recortado, bugs).
2. **Fijar fecha de entrega rígida**
   - Proyecto debe presentarse en un evento (ej. lanzamiento en feria tecnológica).
   - Resultado: se reduce alcance (se entrega solo lo más crítico).
3. **Restricciones de presupuesto**
   - El cliente tiene un presupuesto limitado.

- Resultado: el alcance debe adaptarse, entregando un producto mínimo viable (MVP).

 En conclusión:

- La **administración de proyectos** busca encontrar el **balance entre alcance, tiempo y costo**, sin comprometer la calidad.
- Una **mala gestión** casi siempre lleva al fracaso: retrasos, sobrecostos o insatisfacción del cliente.
- La **planificación** es esencial, pero más importante aún es el **acto de planificar**, porque permite identificar riesgos, estimar alcances y preparar alternativas.

Roles y Componentes Humanos en un Proyecto de Software

Líder de Proyecto

El **líder de proyecto** es la persona responsable de **coordinar, decidir y representar** al equipo ante los diferentes **stakeholders** (clientes, gerencias, patrocinadores, contratistas). En enfoques **tradicionales** suele concentrar todas las decisiones, mientras que en **metodologías ágiles** se transforma en un rol más facilitador y de apoyo.

Aptitudes principales

1. **Motivación** → impulsa al equipo a trabajar a su máxima capacidad.
   - Ejemplo: reconocer logros públicamente para mantener la moral alta.
2. **Organización** → estructura tareas, adapta procesos y garantiza orden.
   - Ejemplo: definir un cronograma claro con dependencias y responsables.
3. **Innovación** → estimula la creatividad y la búsqueda de soluciones nuevas.
   - Ejemplo: alentar a probar una nueva herramienta de integración continua.
4. **Empatía** → comprender la situación personal y profesional de los miembros.
5. **Comunicación** → transmitir objetivos, riesgos y avances de forma clara.
6. **Liderazgo** → inspirar confianza, guiar al equipo y gestionar conflictos.

Habilidades técnicas y administrativas

- **Resolución de problemas**: enfrenta obstáculos técnicos y organizativos, fomenta la búsqueda de soluciones en el equipo.
- **Identidad administrativa**: asume control y responsabilidad en momentos críticos.
- **Orientación al logro**: recompensa el esfuerzo y motiva a asumir riesgos controlados.
- **Influencia**: construye un equipo efectivo mediante retroalimentación constante.

Responsabilidades clave

1. Definir alcance, recursos y presupuesto.
2. Identificar stakeholders y riesgos.
3. Estimar tiempos y tareas.

4. Preparar planes de contingencia.
5. Controlar hitos y cambios en el proyecto.
6. Reportar estado a la dirección/cliente.

➡ **Ejemplo real**: En un proyecto de software bancario, el líder debe negociar con la gerencia las fechas de entrega, mantener informado al cliente, y al mismo tiempo motivar a los programadores que enfrentan problemas técnicos con la integración de sistemas antiguos.

Equipo de Proyecto

Es el **grupo de personas que trabaja de forma colaborativa** para alcanzar los objetivos del proyecto. Se caracteriza por la **complementariedad de habilidades** y la **responsabilidad compartida**.

 Características

1. Diversidad → diferentes conocimientos, experiencias y personalidades.
2. Tamaño pequeño → facilita comunicación y coordinación.
3. Cohesión → espíritu de grupo y sinergia (el resultado es mayor que la suma de esfuerzos individuales).
4. Estabilidad → evitar cambios frecuentes en los integrantes.
5. Responsabilidad conjunta → éxito o fracaso es asumido por todos.

 Ejemplos de funcionamiento de un equipo cohesivo

- El grupo establece estándares propios de calidad (ej. nivel de cobertura en pruebas unitarias).
- Los miembros comparten conocimiento → un senior capacita a un junior.
- Apoyo mutuo → si un programador se retrasa, otro lo ayuda para cumplir el hito.
- Fomento de la mejora continua → retrospectivas al final de cada iteración.

➡ **Ejemplo real**: En un equipo Scrum, el desarrollador de frontend colabora con el tester para detectar errores antes de llegar a la revisión con el cliente, evitando retrabajo y fortaleciendo la confianza.

Stakeholders

Los stakeholders **son todos los** interesados **en el proyecto, internos o externos, que** pueden influir o ser afectados por el resultado.

 Tipos principales

1. **Internos**
    - Equipo de proyecto.
    - Líder de proyecto.
    - Dirección de la empresa.
2. **Externos**
    - Cliente o usuario final.
    - Patrocinador (quien financia).
    - Proveedores y contratistas.

➡ **Ejemplo real**:
En el desarrollo de un sistema de gestión hospitalaria:

- **Stakeholders internos** → desarrolladores, testers, líder del proyecto.
- **Stakeholders externos** → médicos (usuarios finales), dirección del hospital (cliente), empresa que provee la base de datos (proveedor).

 En conclusión:

- El **líder de proyecto** es clave para guiar al equipo y negociar con stakeholders.
- El **equipo** debe ser cohesivo, pequeño y responsable colectivamente.
- Los **stakeholders** definen los intereses y expectativas que deben equilibrarse.

Plan de Proyecto en el Desarrollo de Software

El **plan de proyecto** es la **hoja de ruta** que documenta cómo se va a ejecutar, controlar y finalizar un proyecto de software.
Responde a cuatro preguntas fundamentales:

- **Qué** se va a hacer → Alcance.
- **Cuándo** se va a hacer → Calendario.
- **Cómo** se va a hacer → Proceso, tareas y ciclo de vida.
- **Quién** lo va a hacer → Responsables y equipos.

1. Definición del Alcance del Proyecto

- **Alcance del producto**: qué funcionalidades tendrá el software (ej. login, reportes, integraciones).
- **Alcance del proyecto**: todo el trabajo necesario para entregar el producto (ej. análisis de requerimientos, codificación, testing de ciertos módulos).

 Ejemplo:

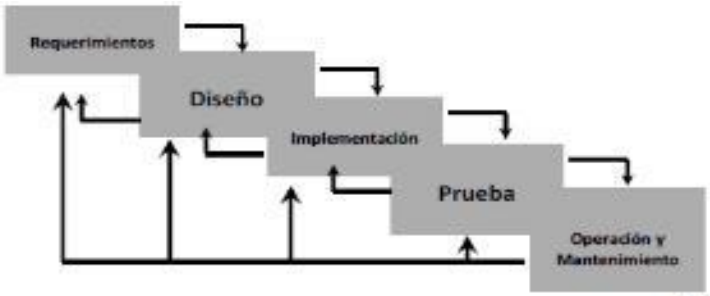
- Producto → "App de banca móvil con transferencias, pagos y consultas de saldo".
- Proyecto → "Definir requerimientos, programar módulo de transferencias, probar integración con el sistema central".

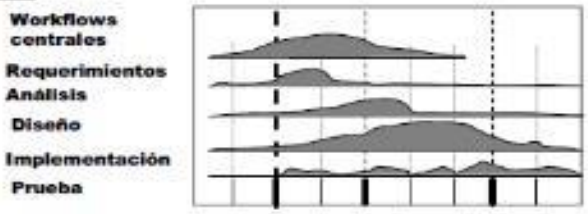
 Dato clave: El alcance del proyecto es **menor** que el del producto.

2. Definición del Proceso y Ciclo de Vida

El proceso y el ciclo de vida responden al **cómo** se va a desarrollar el software.

- **Tradicional** → se puede elegir cascada, espiral, incremental, etc.
- **Ágil** → siempre iterativo e incremental.





El ciclo de vida define:

1. Qué tareas técnicas hacer en cada fase.
2. Quién participa en cada fase.
3. Cómo se aprueban fases y entregables.
4. Cómo revisar y validar el producto.

 Ejemplo:

- En cascada: análisis → diseño → codificación → pruebas → entrega.
- En ágil: backlog → sprint → entrega parcial → retroalimentación.

3. Estimación

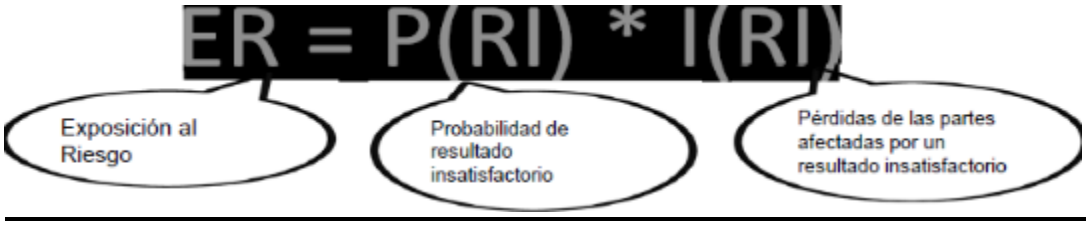
La **estimación** busca predecir tiempo, esfuerzo, costo y recursos.

Orden recomendado en enfoque tradicional:

1. **Tamaño del producto** → líneas de código, puntos de función, cantidad de requerimientos.
2. **Esfuerzo** → horas-persona necesarias.
    - Ejemplo: un proyecto con buenos analistas de requisitos puede reducir un 29% el esfuerzo; con malos analistas puede aumentar un 42%.
3. **Calendario** → duración (meses, trimestres).
4. **Costos** → depende de esfuerzo, tamaño y duración (salarios, infraestructura, licencias).

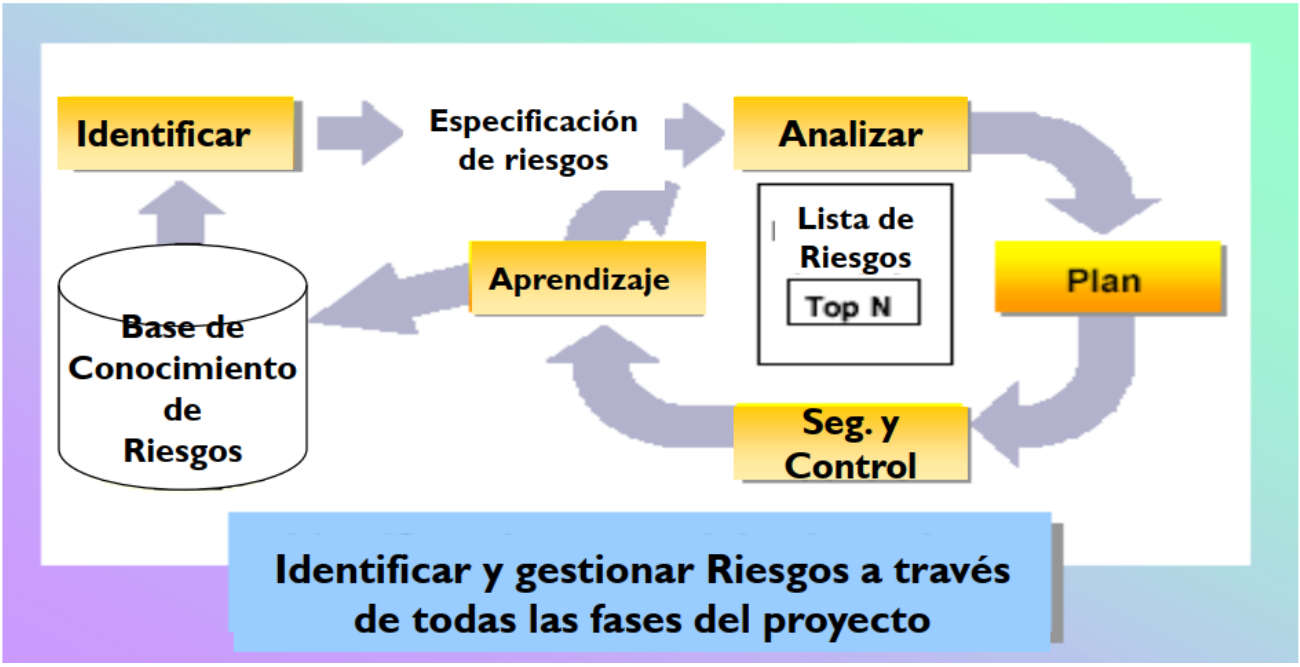
 Dato clave: El 80% de los costos suelen ser esfuerzo humano.

4- Gestión de Riesgos



Un **riesgo** = probabilidad × impacto.
Clasificación:

1. **De proyecto** → afectan recursos o calendario. Ej: renuncia de un programador clave.
2. **De producto** → afectan calidad. Ej: un componente adquirido no funciona como se esperaba.
3. **Organizacionales/negocio** → afectan a la empresa. Ej: un competidor lanza antes un producto similar.



Actitudes frente al riesgo

- **Negación** → ignorarlos.
- **Reactiva** → esperar a que ocurra.
- **Proactiva** → mitigarlos antes y tener planes de contingencia.

 Ejemplo práctico:

- Riesgo: "cambio de requerimientos durante el desarrollo".
- Plan de mitigación: reuniones quincenales con el cliente.

- Plan de contingencia: replanificar alcance del sprint afectado.

Factores de Riesgo Comunes en Software

- Volatilidad de requerimientos.
- Abandono de integrantes del equipo (rotación laboral).
- Falta de capacidad técnica.
- Rápido avance tecnológico.

5 Asignación de responsabilidades

- Consiste en **asignar roles a las personas** del equipo de trabajo.
- No se usa el término “recursos humanos” → se enfatiza que son **personas con habilidades**.
- En proyectos pequeños, una persona puede asumir varios roles.
- Para explicitar los roles y responsables, se suele usar una **tabla con: persona – rol – responsabilidades**.
- La **selección del personal** es crítica, porque las capacidades del equipo afectan directamente la calidad del software y el cumplimiento de tiempos.
  □ El desarrollo de software es una **actividad humano-intensiva**.

6 Programación de proyectos (calendarización)

- Define **en detalle cada tarea** del desarrollo.
- Parte de la estimación inicial del calendario, pero lo lleva al **máximo nivel de detalle**:
  - Qué tarea.
  - Quién la realiza.
  - Cuándo se realiza.
  - Cuánto esfuerzo requiere (en teoría).
- Se usa generalmente un **Diagrama de Gantt**, con herramientas como *Microsoft Project*.

7 Definición de métricas

- Una **métrica** es una medida numérica que aporta visibilidad sobre el estado de:
  - Proyecto.
  - Proceso.
  - Producto.
- Deben ser **claras, representativas y útiles** para el seguimiento.
- En metodologías tradicionales → se definen métricas explícitas.
- En metodologías ágiles → el mayor indicador de avance son los **incrementos entregados** en cada iteración.

Métricas básicas en proyectos de software:

- **Tamaño del producto** (LOC, requerimientos, puntos de función, etc.).
- **Esfuerzo** (horas-persona).
- **Calendario** (tiempos de ejecución).

- **Defectos** (cantidad, densidad).
- **Costos**.

8 Definición de controles

- Se basan en las **métricas definidas**.
- Permiten **verificar y controlar** que todo se ejecute conforme a lo planificado.
- Incluyen:
  - **Reuniones periódicas**.
  - **Reportes de avance**.
  - **Informes de estado**.
- Su objetivo es **detectar desvíos a tiempo** y corregirlos.

 En conclusión:
El **plan de proyecto** no solo organiza el trabajo, sino que también anticipa problemas y define cómo responder. Una buena planificación **no garantiza el éxito**, pero una mala planificación casi siempre lleva al fracaso.

Planes de Soporte

● Planes de Testing.
● Planes de Mantenimiento.
● Planes de subcontrataciones.
● Planes de calidad.
● Planes de capacitaciones.
● Planes de iteraciones (si el ciclo de vida es iterativo/incremental).

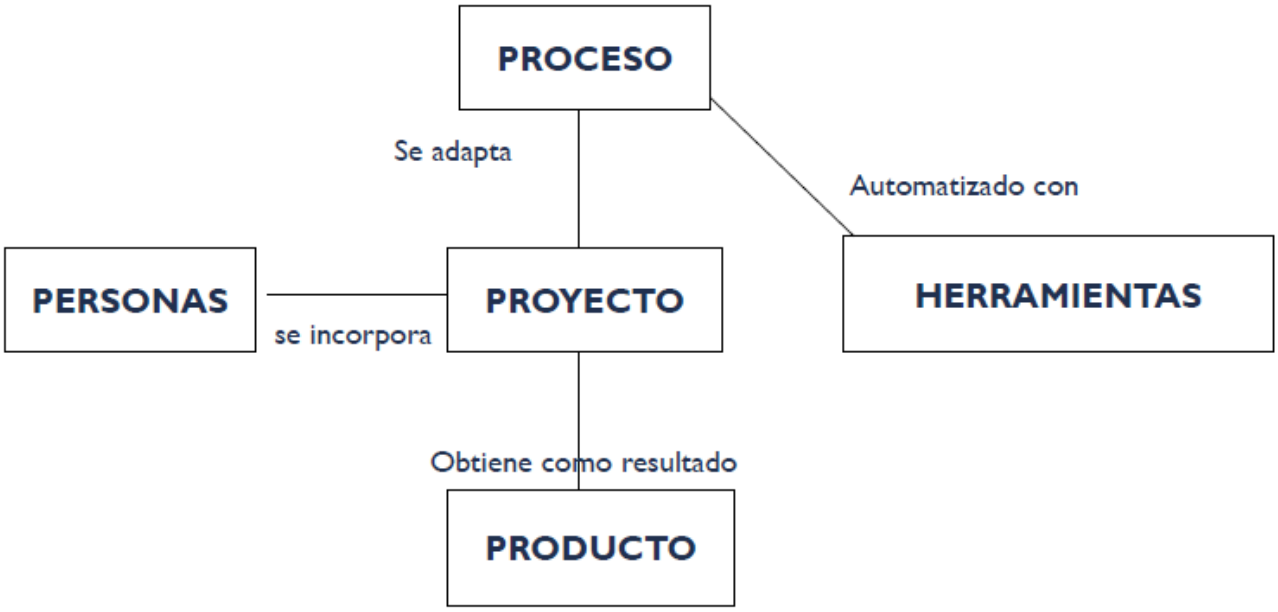
Causas de fracasos de proyectos

● Fallas de toma de REQUERIMIENTOS (el 80%).
● Fallas al definir el problema.
● Planificar basado en datos insuficientes.
● La planificación la hizo el grupo de planificaciones.
● No hay seguimiento del plan de proyecto.
● Plan de proyecto pobre en detalles.
● Planificación de recursos inadecuada.
● Las estimaciones se basaron en “supuestos” sin consultar datos históricos.
● Nadie estaba a cargo.

Vínculo Proceso – Proyecto – Producto

En un **proyecto de desarrollo de software** existen tres elementos clave que se relacionan de manera directa:

1. **Proceso** → marco conceptual y metodológico que define *cómo* se construirá el software.
2. **Proyecto** → organización de recursos, personas y actividades que aplican ese proceso en un caso concreto.
3. **Producto** → resultado final: el software (o servicio) desarrollado que satisface las necesidades del cliente.



1. El Proceso

- Es una **definición teórica** de las actividades necesarias para transformar requerimientos en un producto/servicio.
- Se adapta al tipo de proyecto, ya que **no existe un proceso ideal universal**.
- Se expresa mediante un **modelo de ciclo de vida**: secuencial, iterativo o recursivo.
- Incluye actividades fundamentales (planificación, análisis, diseño, codificación, pruebas, mantenimiento).
- Integra **disciplinas de gestión** (planificación, control, estimaciones, riesgos) y **disciplinas de soporte** (aseguramiento de calidad, documentación, métricas, etc.) que atraviesan todas las fases.

 Ejemplo: el modelo en cascada puede usarse para sistemas estables con requisitos claros, mientras que el modelo iterativo-incremental es mejor si los requisitos cambian.

2. El Proyecto

- Es la **unidad organizativa y temporal** que aplica un proceso adaptado a un caso específico.
- Integra:
  - **Personas** → equipo de trabajo con roles y responsabilidades definidos.
  - **Recursos** → tiempo, presupuesto, infraestructura, herramientas.
  - **Actividades** → aquellas definidas en el proceso, pero **ajustadas** a las necesidades concretas del proyecto.

 Ejemplo: dos proyectos que usan “modelo en espiral” pueden diferir, ya que uno puede poner más esfuerzo en análisis de riesgos y otro en prototipado.

3. El Producto

- Es el **resultado único** del proyecto: un software o servicio (ej. **SaaP** → software como producto, o **SaaS** → software como servicio).
- Su definición comienza antes del proyecto:
  - **Objetivos** → qué se quiere lograr (ej. “automatizar la gestión de clientes”).
  - **Ámbito del producto** → qué funciones y comportamientos tendrá (ej. “registro de clientes, consultas, generación de reportes”).

 El **proyecto** adopta un **proceso** para que, con la participación de un equipo, esos objetivos y funciones se materialicen en un producto.

4. Rol de las Personas y Herramientas

- El software es una actividad **humano-intensiva**:
  - Un “buen proceso” + “planificación detallada” no garantizan éxito si no existe un equipo capacitado.
  - Se requiere un equipo con habilidades técnicas y de gestión, usando solo las prácticas del proceso que realmente agreguen valor.
- **Herramientas** → apoyan la automatización de tareas dentro del proceso.
  - Ejemplo: CASE para modelado UML, Jira para gestión de proyectos, Git para control de versiones.

5. Resumen Gráfico del Vínculo

- **Proceso** = cómo se trabaja (marco conceptual + actividades).
- **Proyecto** = quién y con qué recursos aplica el proceso.
- **Producto** = qué se obtiene como resultado.

 Relación:

- Un **proceso** se adapta al **proyecto**.
- El **proyecto** organiza recursos y personas.
- El **producto** es el resultado de aplicar ese proceso en ese proyecto.

 En síntesis:
El éxito en la gestión de un proyecto de software depende de la **correcta articulación del proceso (metodología), el proyecto (personas y recursos) y el producto (resultado esperado)**. Si alguno de los tres falla (ej. mal proceso, equipo débil o producto mal definido), el resultado será insatisfactorio.

No Silver Bullet: Lo esencial y lo accidental en la ingeniería del software

 La idea central:
En el desarrollo de software **no existen "balas de plata"**, es decir, no hay una única técnica, herramienta o avance que logre **mejorar drásticamente (×10) la productividad, simplicidad o confiabilidad** en menos de una década.
El software es **intangible, abstracto y complejo por naturaleza**, por lo que los problemas "esenciales" no tienen una solución mágica.

Diferencia entre hardware y software

- **Hardware** → tangible, medible, susceptible de mejoras físicas (velocidad, miniaturización, potencia).
- **Software** → intangible, abstracto, inherentemente complejo y difícil de visualizar.

Dificultades en el software

1. Dificultades esenciales (propias de la naturaleza del software)

Son inevitables y no se pueden eliminar con herramientas:

1. **Complejidad**
   - Aumenta exponencialmente con el tamaño del software.
   - Provoca problemas de comunicación, errores en planificación, excesos de costo.
2. **Conformidad**
   - El software debe ajustarse estrictamente a los requerimientos del cliente, que cambian o son ambiguos.
3. **Mutabilidad**
   - El software siempre está sujeto a cambios y evoluciones.
   - A diferencia de productos físicos, no se reemplaza fácilmente, sino que se modifica.
4. **Invisibilidad**
   - No puede representarse geométricamente.
   - Los diagramas ayudan, pero nunca representan el software de forma completa.

2. Dificultades accidentales (ya mitigadas con avances tecnológicos)

Históricamente fueron grandes problemas, pero hoy tienen soluciones:

- **Lenguajes de alto nivel** → abstraen del hardware.
- **Tiempo compartido** → reduce tiempos de respuesta.

- **Entornos de programación unificados** → integran librerías, compiladores, frameworks.

Propuestas y limitaciones

 Resolución de dificultades accidentales

- **Programación orientada a objetos (POO)** → mejora el modelado, pero no elimina la complejidad esencial.
- **IA y sistemas expertos** → ayudan en pruebas y recomendaciones, pero dependen del conocimiento humano.
- **Programación automática / gráfica** → aumentan la abstracción, pero no eliminan el factor humano.
- **Verificación formal de programas** → asegura corrección matemática, pero no garantiza cumplir expectativas del cliente.
- **Entornos inteligentes y estaciones de trabajo poderosas** → ayudan en eficiencia, pero no transforman radicalmente la productividad.

Resolución de dificultades esenciales

- **Comprar antes que construir**
  - Usar software existente (COTS) siempre que sea posible → más barato y probado.
  - Limitación: no siempre hay soluciones para necesidades muy específicas.
- **Prototipado y refinamiento de requerimientos**
  - Construir prototipos rápidos para clarificar lo que realmente quiere el cliente.
  - El feedback iterativo reduce ambigüedades.
- **Desarrollo incremental**
  - Construir software en partes pequeñas y funcionales.
  - “Crecer el software” → entregar versiones mínimas pero usables antes que esperar un gran producto al final.
- **Personas capacitadas y creativas**
  - El factor humano es clave: buenos diseñadores y líderes son igual de valiosos.
  - La creatividad puede superar la rigidez de procesos.

Conclusión de Brooks

- **No existe una bala de plata** → ningún avance único resolverá la “crisis del software”.
- Los progresos serán **graduales, múltiples y combinados**.
- La clave está en:
  - Usar software ya existente.
  - Prototipar y refinar requerimientos.
  - Adoptar desarrollo incremental.
  - Apostar por personas talentosas y creativas.

En síntesis: El software seguirá siendo complejo, mutable e invisible. No habrá magia que lo simplifique de golpe, pero sí se puede **mejorar de forma progresiva** con buenas prácticas, herramientas adecuadas y equipos capacitados.

Proximanente agregaréla unidad 2