PROCESO UNIFICADO

❖ EL PROCESO UNIFICADO DE DESARROLLO DE SOFTWARE

Es un marco de trabajo genérico que puede ser usado para instanciar proyectos. Define el camino del desarrollo del software (Ciclo de vida) y las actividades que hay que llevar a cabo para transformar requisitos de un usuario en un sistema de software.

1. DIRIGIDO POR CASOS DE USO.

Un CU modela los requerimientos y refleja las expectativas del cliente. Representan la interacción de un actor con una funcionalidad del sistema, es decir una porción de funcionalidad que le aporta valor a un actor.

Se progresa a través de los flujos de trabajo iniciando con un conjunto de CU.

Razones para usar casos de uso:

- **Permiten capturar requisitos funcionales** centrándose en el valor añadido para el usuario y no el de los programadores => centrado en el usuario.

- **Dirigen el proceso** porque dividen la complejidad de los requerimientos en porciones más pequeñas => tomo esas pequeñas partes y voy recorriendo el proceso a partir de los CU seleccionados =>todo el proceso se basa en ir tomando estas porciones. Se usan para definir el contenido de cada iteración.

- **Para idear la arquitectura** ya que los requerimientos son los que le dan la forma, también las restricciones de diseño (infraestructura preexistente, lenguaje, etc) retroalimentan y cambian los requerimientos.

Cualquier modelo o artefacto de cualquier flujo de trabajo se debe poder trazar hasta el CU que le dio sentido. Cosas que no han sido requeridas no deberían estar modeladas.

Se utilizan para conseguir un acuerdo con el cliente de lo que debería hacer el sistema.

2. CENTRADO EN LA ARQUITECTURA

La arquitectura de un sistema es la visión común en la que todos los empleados deben estar de acuerdo. Al arquitecto de software y a los desarrolladores les resulta útil presentar el sistema desde diferentes perspectivas para comprender mejor el diseño. Estas perspectivas son vistas del modelo de sistema. Todas las vistas juntas representan la arquitectura.

La arquitectura define:

- Organización de sistema software
- Elementos estructurales que compondrán el sistema y sus interfaces, junto con sus comportamientos, tal y como se especifican en las colaboraciones entre estos elementos
- Composición de los elementos estructurales y del comportamiento en subsistemas progresivamente más grandes.

- Estilo de la arquitectura que guía esta organización: los elementos y sus interfaces, colaboraciones y composición.

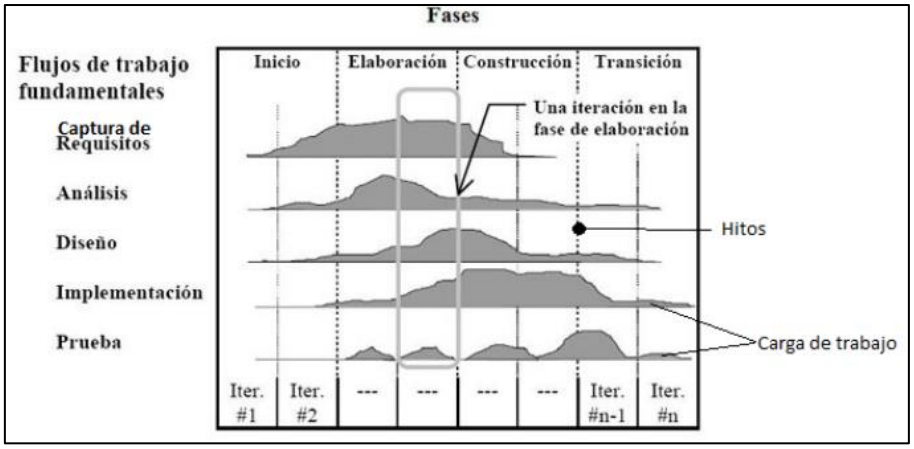
La arquitectura es necesaria porque:

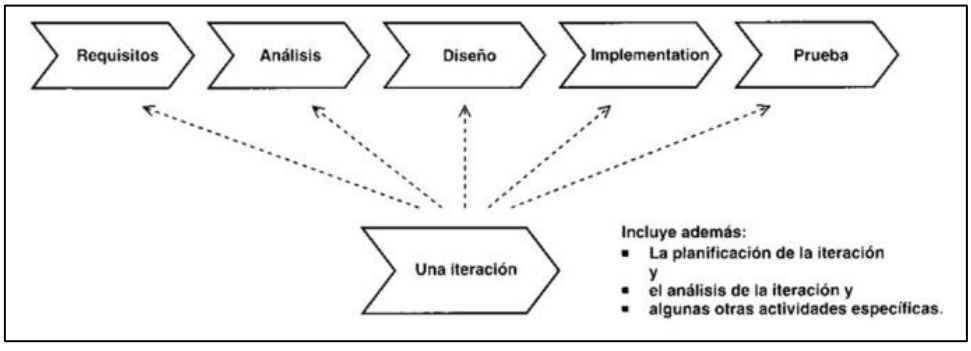
- Comprende el sistema → Para que una organización desarrolle un sistema, dicho sistema debe ser comprendido por todos los que vayan a intervenir en él. El primer requisito que tiene lugar en una descripción de la arquitectura es que se debe capacitar a los desarrolladores, directivos, clientes y otros usuarios para comprender qué se está haciendo con suficiente detalle como para facilitar su propia participación.
- Organiza el desarrollo → Una buena arquitectura es la que define explícitamente estas interfaces, haciendo que sea posible la reducción en la comunicación. Una interfaz bien definida comunica eficientemente a los desarrolladores de ambas partes qué necesitan saber sobre lo que los otros equipos están haciendo
- Fomenta la reutilización → Un buen arquitecto ayuda a los desarrolladores para que sepan dónde buscar elementos reutilizables de manera poco costosa, y para que sepan encontrar los componentes adecuados para ser reutilizados.
- Hace evolucionar el sistema → El sistema debe ser en sí mismo flexible a los cambios o tolerante a los cambios. El sistema debe ser capaz de evolucionar sin problemas

La arquitectura se ve afectada por la funcionalidad, el rendimiento, la flexibilidad, la estética, etc.

3. ITERATIVO E INCREMENTAL

Se divide el trabajo en partes pequeñas de la cual surge un incremento (crecimiento funcional en el producto). Cada fase (Inicio, Elaboración, Construcción y Transición) se divide en n iteraciones planificadas (se selecciona que CU hay que tratar, quien lo hace, etc) y todas pasan por todos los flujos de trabajo (toma de requisitos, Análisis, Diseño, Implementación y prueba). Ventajas: reduce el riesgo a un solo incremento, reducir la posibilidad de retrasos atacando los CU más importantes, los requisitos se pueden ir refinando en cada iteración (difícil definir todo al principio).





Un Ciclo está compuesto por 4 fases:

INICIO: Definir el objetivo (análisis del negocio) y si el proyecto es viable ej: no tenemos los recursos humanos necesarios, el tiempo no es suficiente, el cliente no tiene el suficiente dinero, la tecnología, etc.

Hito de los objetivos del ciclo de vida. (Visión)

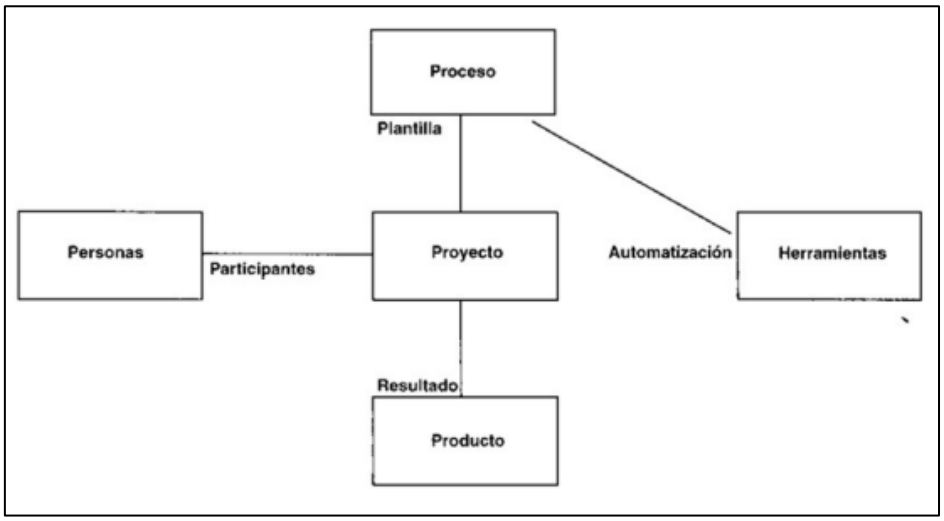
ELABORACIÓN: Defino la línea base de la arquitectura. (Al ser centrado en la arquitectura es lo primero que necesito). Con esto puedo planificar mejor las actividades y los recursos necesarios para terminar el proyecto. => todas las iteraciones de esta fase tienen que estar apuntadas a determinar la arquitectura del sistema. Hito de la arquitectura.

LÍNEA BASE DE LA ARQUITECTURA: es la suma de todas las vistas de la arquitectura. Es un sistema pequeño que representa el esqueleto del sistema. Tiene las funcionalidades más importantes.

CONSTRUCCIÓN: Se crea el producto y la línea base de la arquitectura crece hasta convertirse en un sistema completo. Implementa todos los CU. Es una versión alfa o beta (puede no estar libre de defectos). Hito de la funcionalidad o capacidad operativas inicial ->obtenemos una versión operativa.

TRANSICIÓN: Se prueba el producto y se informan los defectos encontrados, se corrigen los problemas y se incorporan mejoras. Siempre las iteraciones siguen agregando características al producto. Hito del lanzamiento del producto (Release).

- **LAS 4 "P" EN EL DESARROLLO DE SOFTWARE**
  - **Personas:** Los principales autores de un proyecto software, además de los usuarios, clientes y otros interesados. Las personas son realmente seres humanos.
  - **Proyecto:** Elemento organizativo a través del cual se gestiona el desarrollo de software.
  - **Producto:** Artefactos que se crean durante la vida del proyecto
  - **Proceso:** Conjunto completo de actividades necesarias para transformar los requisitos de usuario en un producto
  - **Herramientas:** Software para automatizar las actividades definidas en el proceso



❖ CAPTURA DE REQUISITOS

El usuario nos transmite lo que debe hacer el sistema y sus expectativas -> es una fuente imperfecta de información, además puede haber ruido en la comunicación y surgir malinterpretaciones.

Objetivos:

- Descripción del sistema correcto de lo que debe y no debe hacer el software.
- Entendible por el usuario para que pueda verificar que lo que capturamos es correcto y lo acepte o no.
- Planificación y seguimiento

=> Usamos CU como la forma en la que vamos a plasmar los requisitos y guiar el proyecto.

Pasos:

Trabajo a Realizar	Artefacto Resultante
Enumerar requisitos candidatos	Lista de características
Comprender el contexto del sistema	Modelo de dominio o de negocio
Capturar los requisitos funcionales	Modelo de Caso de uso
Capturar requisitos no funcionales	Requisitos adicionales, propiedades del sistema

- **ENUMERAR LOS REQUISITOS CANDIDATOS:** lista de ideas que podemos decidir implementar en una versión futura del sistema. Esta lista de características se utiliza sólo para la planificación del trabajo. Cada característica tiene también un conjunto de valores de planificación que podríamos incluir: estado, coste estimado de implementación, prioridad, nivel de riesgo asociado a la implementación de la característica

- **COMPRENDER EL CONTEXTO DEL SISTEMA:** para capturar los requisitos correctos y para construir el sistema correcto los desarrolladores clave requieren un firme conocimiento del contexto en el que se emplaza el sistema

- **CAPTURAR REQUISITOS FUNCIONALES**: se basa en los casos de uso. Estos casos de uso capturan tanto los requisitos funcionales como los no funcionales que son específicos de cada caso de uso. Para el usuario un caso de uso es un modo de utilizar el sistema.

- **CAPTURAR REQUISITOS NO FUNCIONALES**: los requisitos no funcionales especifican propiedades del sistema, restricciones del entorno o de la implementación, rendimiento, dependencias de la plataforma, facilidad de mantenimiento, extensibilidad y fiabilidad. Los requisitos de rendimiento afectan sólo a ciertos casos de uso y por tanto deberían conectarse a ese caso de uso. Algunos requisitos no funcionales son más genéricos y no pueden relacionarse con un caso de uso o requisitos adicionales.

MODELO DE DOMINIO

Captura los tipos más importantes de objetos en el contexto del sistema. Los objetos del dominio representan las cosas que existen o los eventos que suceden. Las clases del dominio aparecen en tres formas típicas:

- Objetos del negocio que representan cosas que se manipulan en el negocio
- Objetos del mundo real y conceptos de los que el sistema debe hacer un seguimiento
- Sucesos que ocurrirán o han ocurrido

El modelo del dominio se describe mediante diagramas de UML.

El objetivo del modelado del dominio es comprender y describir las clases más importantes dentro del contexto del sistema.

Se utiliza: al describir los casos de uso y al diseñar la interfaz de usuario, para sugerir clases internas al sistema en desarrollo durante el análisis.

MODELO DE NEGOCIO

Objetivo: es identificar los casos de uso del software y las entidades de negocio relevantes que el software debe soportar, de forma que podríamos modelar sólo lo necesario para comprender el contexto.

El modelo del negocio está soportado por dos tipos de modelos UML**: modelos de casos de uso** y **modelos de objetos**.

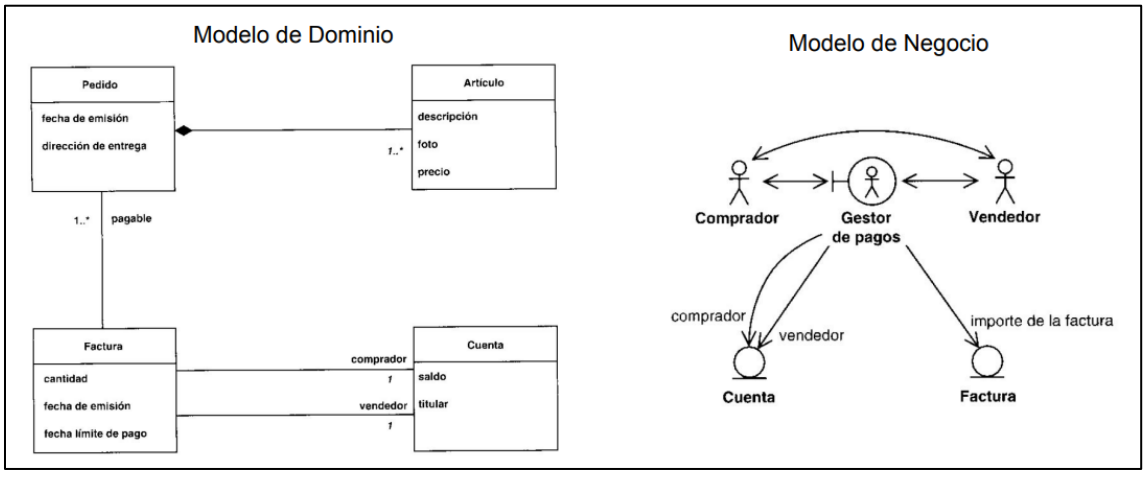
El modelo de negocio describe los procesos de negocio de una empresa en términos de casos de uso del negocio y actores del negocio con los procesos del negocio y los clientes.

El **modelo de casos de uso** se describe mediante diagramas de casos de uso.

Un **modelo de objetos** del negocio describe cómo cada caso de uso de negocio es llevado a cabo por parte de un conjunto de trabajadores que utilizan un conjunto de entidades del negocio y de unidades de trabajo.

Una **entidad de negocio** representa algo que los trabajadores utilizan en un caso de uso del negocio.

Una **unidad de trabajo** es un conjunto de esas entidades que conforman un todo reconocible para un usuario final.



CAPTURA DE REQUISITOS EN CASOS DE USO

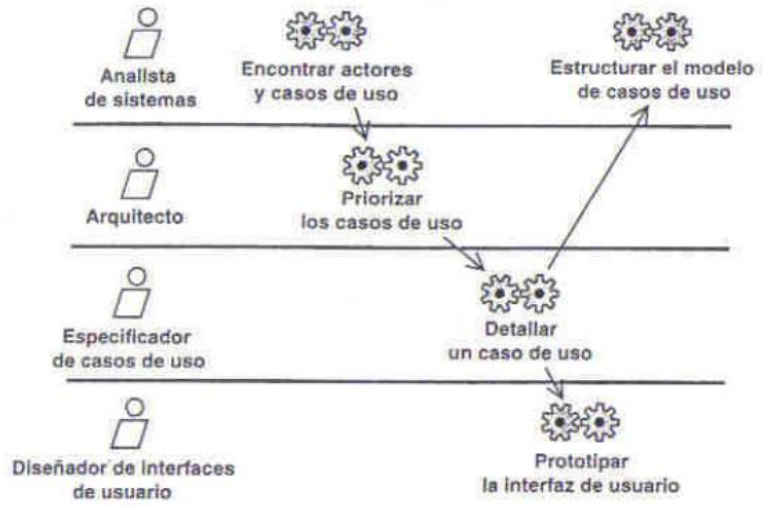
- **Encontrar Actores y CU (Analista):** Identificar quienes o que interactúan con el sistema, encontrar los CU, describir brevemente el objetivo de los CU, armar el modelo de CU, Definir un glosario de términos.
  Entrada = Modelo de negocio - Listado de requisitos
  Resultado = Modelo de CU esbozo (diseño provisional) – Glosario

- **Priorizar CU (Arquitecto):** Identificamos los CU principales y establecemos prioridades. Basado en la arquitectura => arrancamos por los CU de mayor prioridad. (Vista de la Arq.).
  Resultado= Descripción de la arquitectura (funcionalidad más importante y crítica)

- **Detallar los CU (Especificador):** Describe el flujo de sucesos en detalle para cada CU (Precondiciones, curso normal, curso alternativo, postcondiciones, requisitos no funcionales).
  Para formalizar la descripción y que sea más fácil de interpretar (sobre todo si tiene muchos flujos alternativos) podemos hacer diagramas de estados y de actividad.
  Resultado = Detalles de CU

- **Prototipado de Interfaces (Diseñador):** Sirve para validar que la interpretación sea correcta. Cuando el usuario lo vea puede notar si falta algo.
  Resultado = Interfaces de usuario.

- **Estructurar el modelo de CU (Analista):** Se buscan acciones comunes o compartidas por varios CU (relaciones include, extend, generalización), se eliminan redundancias, se dividen un CU muy complejo en 2, etc.
  Resultado: Modelo de CU bien estructurado.



❖ GESTIÓN DE CONFIGURACIÓN DE SOFTWARE (GCS)

CONFIGURACIÓN: Disposición de las partes que componen una cosa y le dan su forma y propiedades. Conjunto de cosas que forman algo

ELEMENTOS DE CONFIGURACIÓN DEL SOFTWARE(ECS):

- Programas de computadora (tanto en forma de código fuente como ejecutable)
- Documentos que describen los programas de computadora (tanto técnicos como de usuario).
- Datos (contenidos en el programa o externos a él).

CONFIGURACIÓN DE SOFTWARE: es el conjunto de todos los ECS

GESTIÓN: Es un conjunto de actividades diseñadas para controlar el cambio. Es una actividad de protección del proceso que reduce los riesgos. Sirven para identificar el cambio, controlarlo, implementarlo adecuadamente e informar a las partes interesadas. Ya que no importa en qué punto del ciclo de vida estemos, los cambios son inevitables.

FUENTES DEL CAMBIO:

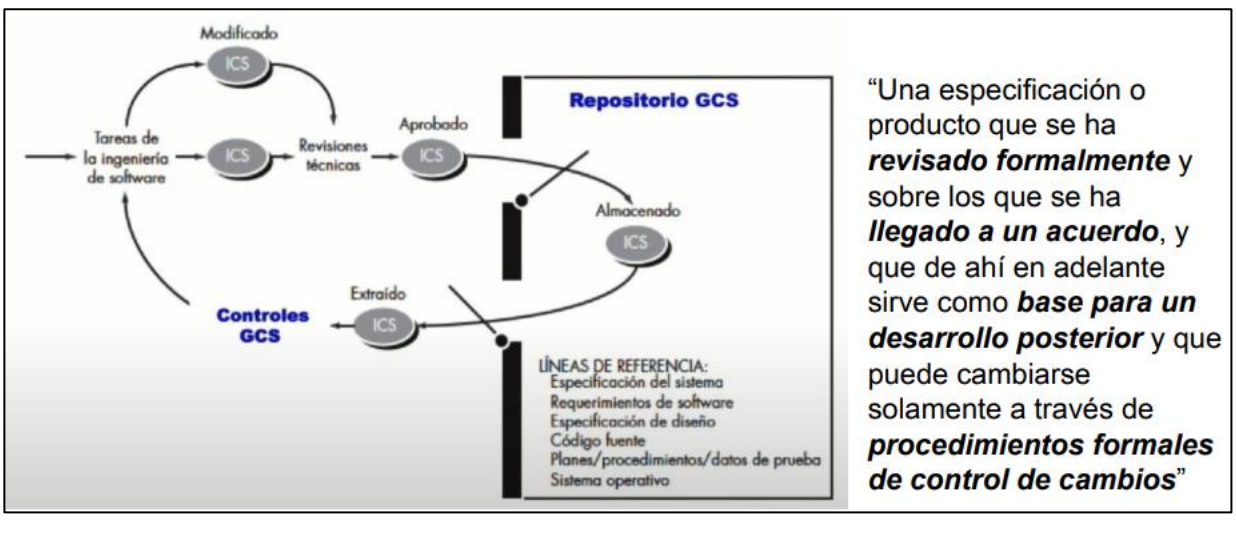
- Nuevos negocios o condiciones comerciales que dictan los cambios en los requisitos del producto o en las normas comerciales.
- Nuevas necesidades del cliente que demandan la modificación de los datos producidos, funcionalidades entregadas por productos o servicios entregados por un sistema.
- Reorganización o crecimiento/reducción del negocio que provoca cambios en las prioridades del proyecto.
- Restricciones presupuestarias o de planificación.

La GCS es una actividad de protección que esta presente durante todo el ciclo de vida del desarrollo de software.

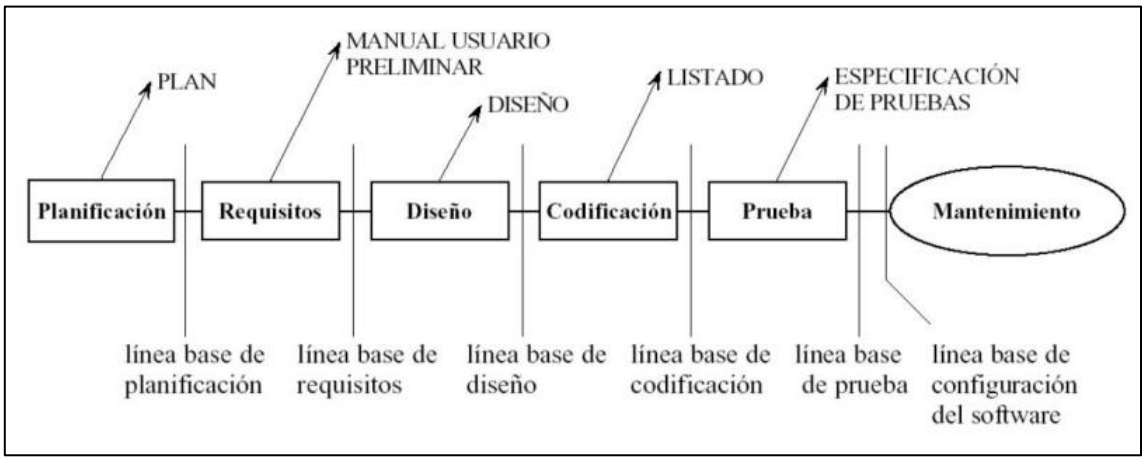
OBJETIVOS DEL GCS:

1. Identificar los productos de trabajo que pueden cambiar
2. Definir mecanismos para administrar distintas versiones de los productos de trabajo y controlar los cambios.
3. Garantizar que el cambio se implementó de forma adecuada.
4. Auditar los cambios e informar a todo el equipo e interesados en cada cambio
5. Generar reportes e informar al equipo.

LÍNEA BASE → Es un producto que se revisó formalmente y se aprobó y que todos están de acuerdo, sirve como base para continuar el desarrollo y solo puede cambiarse a través de procedimientos de control de cambios formales. Es una línea de referencia.



LÍNEAS BASES



- **ANÁLISIS**

Terminar de definir qué tengo que hacer y estructurar los requerimientos, se centra en definir una arquitectura inicial de alto nivel.

OBJETIVOS:

- Especificación más precisa de los requisitos
- Utiliza el lenguaje de los desarrolladores
- Estructura los requerimientos de modo de facilitar su comprensión, separación y modificación, y en general, su mantenimiento.
- Primera aproximación al diseño. Mayor abstracción, hacemos hincapié en la funcionalidad, es decir que tiene que hacer el sistema y no tanto como lo va hacer
- Se centra en la primera parte de la fase de elaboración.

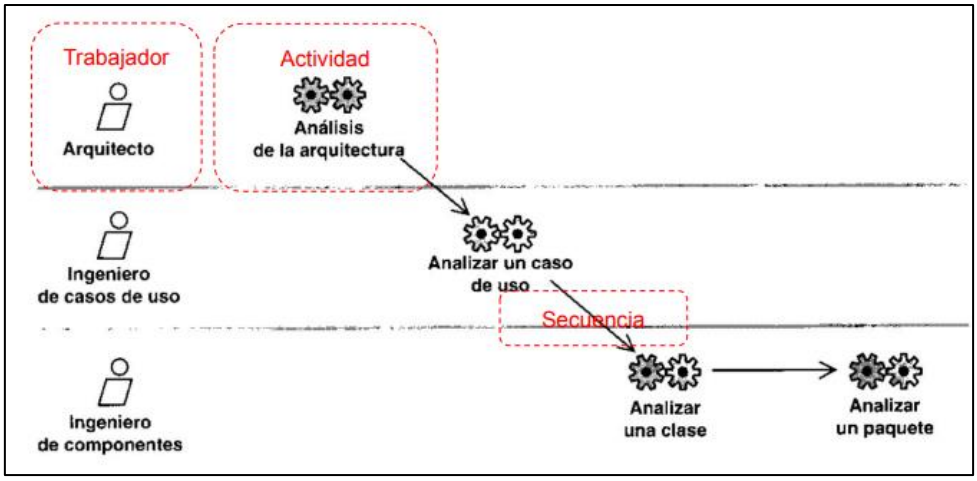
MODELO DE CASOS DE USO VS MODELO DE ANÁLISIS

Describe el problema

Modelo de Casos de Uso	Modelo de Análisis
Descrito con lenguajes del cliente	Descrito con lenguaje del desarrollador
Visión externa del sistema	Visión interna del sistema
Estructurado por casos de uso, proporciona la estructura a la vista externa	Estructurado por clases y paquetes, proporciona la estructura a la vista interna
Utilizada como contrato entre cliente y desarrollador sobre qué debería hacer el sistema	Utilizado por desarrolladores para comprender cómo debería darse forma al sistema
Puede contener redundancias, inconsistencias entre requisitos	No debe contener redundancias, inconsistencias entre requisitos
Captura la funcionalidad del sistema, incluida la funcionalidad significativa para la arquitectura	Esboza cómo llevar a cabo la funcionalidad dentro del sistema. 1º aproximación al diseño
Define casos de uso que se analizarán con más profundidad en el modelo de análisis	Define realizaciones de casos de uso, y cada una de ellas representa el análisis de un caso de uso del modelo de caso de uso

Esquematiza el problema desde un punto de vista más cercano al desarrollador

FLUJO DE TRABAJO

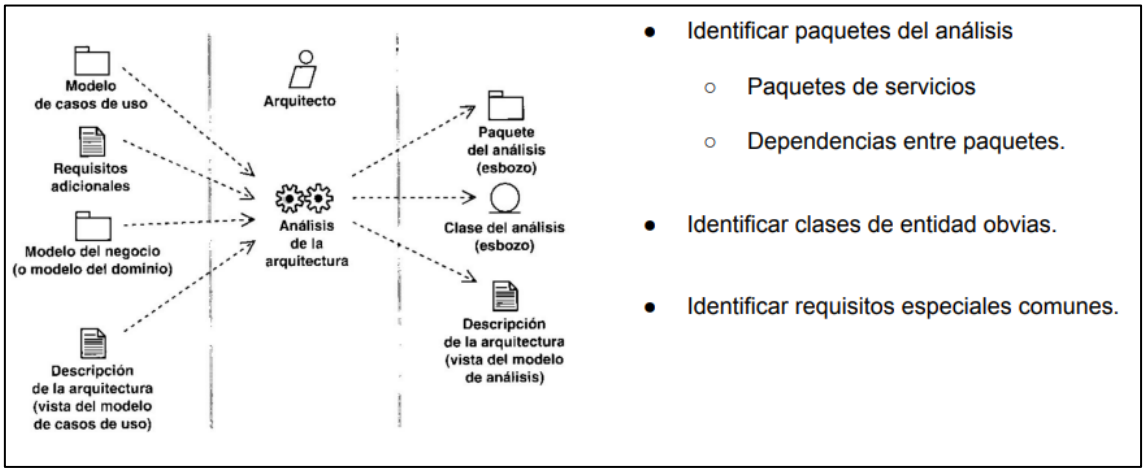


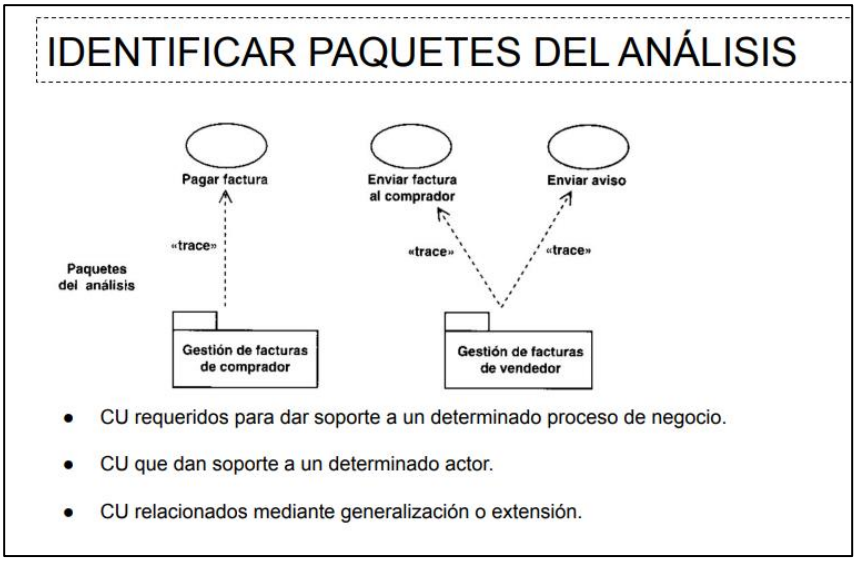
Análisis de la arquitectura (Arquitecto):

1. Identificación de paquetes de análisis. Estos organizan el modelo en piezas más pequeñas y manejables asociando CU dependiendo su funcionalidad. Se busca que sean altamente cohesivos (clases fuertemente relacionadas) y paquetes débilmente acoplados (poca interacción entre paquetes). También hay paquetes de servicios (notificaciones, permisos de usuario, seguridad, etc) que dan soporte a otras partes del sistema (permiten la reutilización de código). Identificar dependencias entre paquetes.

2. Identificar las clases obvias, fáciles de identificar. (Clases entidad)

3. Identificar los requisitos especiales o requisitos no funcionales (durante todo el proceso)



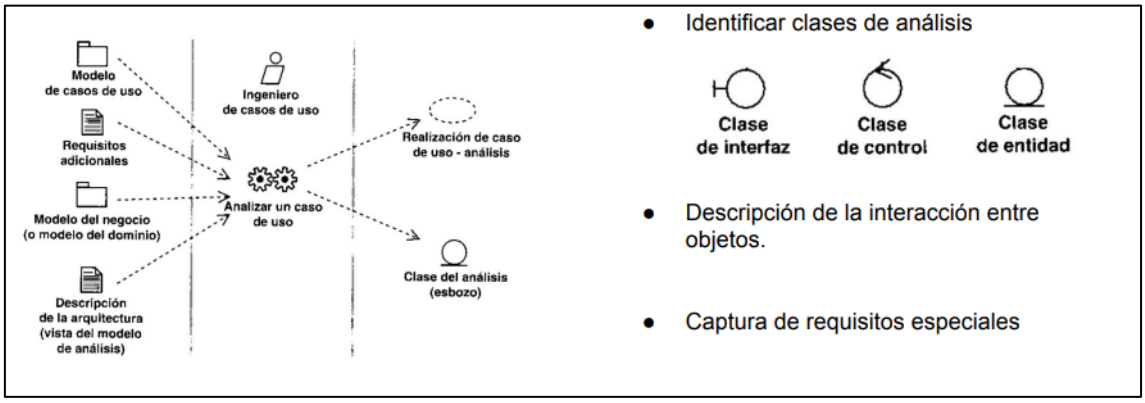


Analizar un CU (Ing de CU):

1. Identificar las clases del análisis del CU (entidad, control, interfaz). Esbozamos nombre, responsabilidad, atributos y relaciones. (Visión estática).

2. Describir interacciones entre los objetos del análisis (instancias de esas clases que usan métodos para interactuar) usando Diagramas de interacción (Secuencia o Colaboración) (Visión dinámica) Nota: tener en cuenta el flujo de sucesos que describimos cuando detallamos el CU en la captura de requisitos.

3. Capturar requisitos especiales o no funcionales.

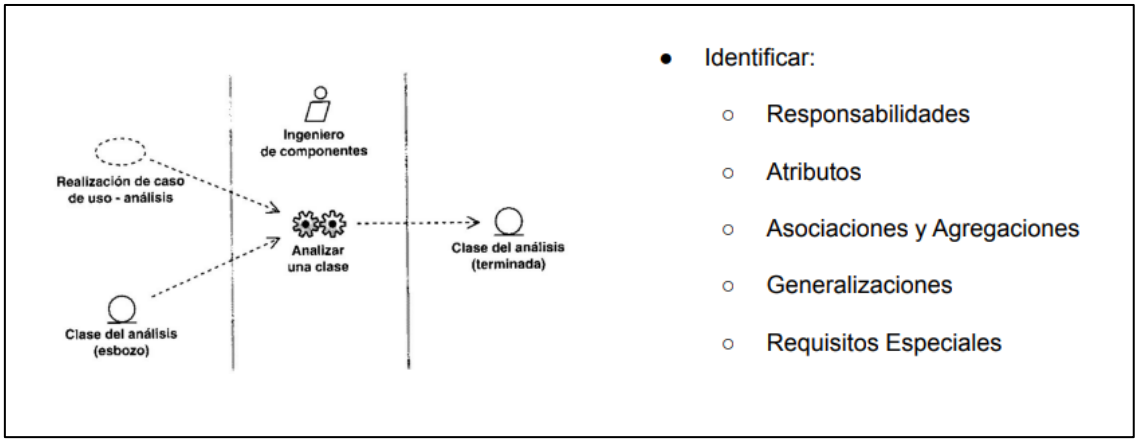


Analizar una clase (Ing en componentes): Luego de identificarlos en el paso anterior:

1. Identificar sus responsabilidades (en el análisis todavía no se vuelven métodos)

2. Identificar atributos y relaciones (Asociación, agregación, generalización, cardinalidad)

3. Capturar requisitos especiales o no funcionales.



Analizar un paquete (Ing de componentes): En el paso anterior identificamos que clase colabora con cual = > de esa forma volver a analizar los paquetes para que terminen siendo lo más independiente posible -> modularidad => más fácil de mantener, modificar, reutilizar, etc. Buscamos descubrir dependencias para poder estimar los efectos de cambios futuros, asegurarnos que los paquetes tengan las clases correctas . Nunca van a ser totalmente independientes sino serían sistemas distintos.