TODO INGENIERÍA SOFTWARE

(a partir del tema 6)

TEORÍA TEMA 6:

Diagrama de Clases:

Un **diagrama de clases es un tipo de diagrama** que **describe la estructura de un sistema mediante** la representación de **sus clases, sus atributos, sus operaciones** (o métodos) **y las relaciones existentes entre los objetos.**

·**Clase** → **Define una categoría de objetos**, con características como los atributos y los métodos. Estas **tienen 3 compartimentos**, el de **nombre**, el de **atributo** y el de **operación**.

-Compartimento de nombre: Sirve para **indicar el nombre de la clase** (obligatoria) o si es abstracta (como <<abstract>>)

-Compartimento de atributo: Solo el nombre es obligatorio. Indica:

1) Visibilidad: pública (+), privada (-), protegida (#)
2) Tipo variable: string, integer, float …
3) Multiplicidad
4) Valor Incial: valor con el que empezará.

```
[visibilidad] nombre: tipo[multiplicidad] [= valor_inicial]
```

-Compartimento de operación: Solo nombre es obligatorio. Indica:

1) Visibilidad: pública (+), privada (-), protegida (#)
2) Tipo de retorno: string, integer, float …
3) Lista de parametros: donde se indica de cada uno su dirección (in,out,…), tipo y valor predeterminado.

```
[visibilidad] nombre [(lista_parámetros)] [: tipo_retorno]  [{propiedades}]
[dirección] nombre_parámetro : tipo_parámetro = valor_predeterminado
```

·**Objeto** → **Instancias de una clase**, objeto con valores en sus atributos.

·**Operaciones** → **Acciones que un objeto puede realizar.**

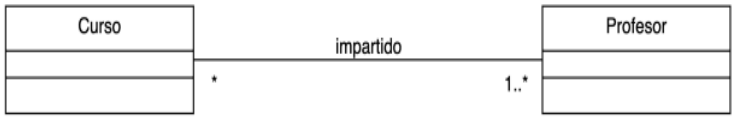
·**Relaciones** → **Son conexiones semánticas significativas entre elementos de un modelo.** Existen 3 tipos principales:

-Relación entre objetos (vínculos): Es la **conexión entre dos objetos para indicar que se comunican entre sí**.

-Relación de dependencia: Es la **conexión entre dos o más objetos para indicar que uno depende del otro**, el cambio de uno afecta al o a los otros.

-Generalización: Es la **relación entre un elemento y otro más específico**. Es, a simples rasgos, la herencia.

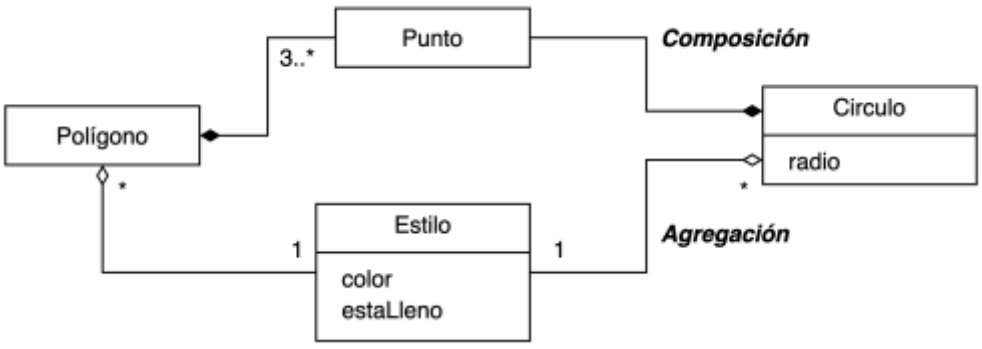
-Asociación: Es la **relación entre clase, su semántica solo indica que están relacionados**. Pueden tener nombre de la relación y la cardinalidad.



Adorno	Semántica
0..1	Cero o uno
1	Exactamente uno
0..*	Cero o más
*	Cero o más
1..*	Uno o más
n...m	Entre n y m

-Agregación: Relación entre **una clase que constituye un "todo" y otras clases que representan sus partes**. Cada clase puede existir independientemente (Camiseta y tela).

-Composición: Es **agregación fuerte entre las partes y el todo**. La **diferencia es que las partes no tienen vida independiente fuera del todo** (Cuadrado y línea).



-Clase asociación: **Útiles en relaciones de asociación con multiplicidad *, cuando los atributos no encajan en ninguna de las dos clases**. Forma **parte de la asociación**, junto a todas sus propiedades.

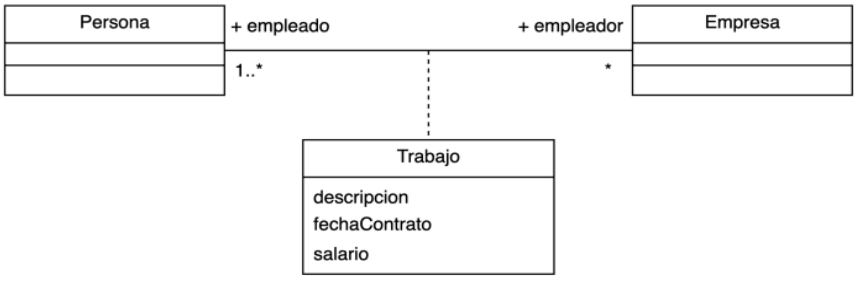
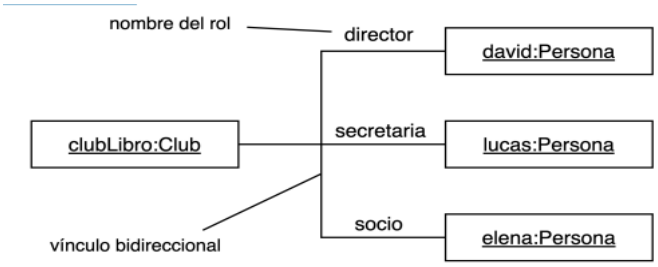


Diagrama de Objetos:

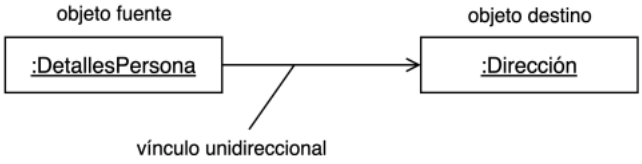
Un diagrama de objeto muestra una vista completa o parcial de los objetos en un instante de ejecución específico. Su objetivo es **modelar las instancias de los elementos presentes en los diagramas de clases**.

·Vínculos y roles → Son las relaciones que hay entre las instancias de las clases. Tenemos dos tipos:

-Vínculo bidireccional: **Ambos elementos tienen conocimiento mutuo y pueden interactuar entre sí**.



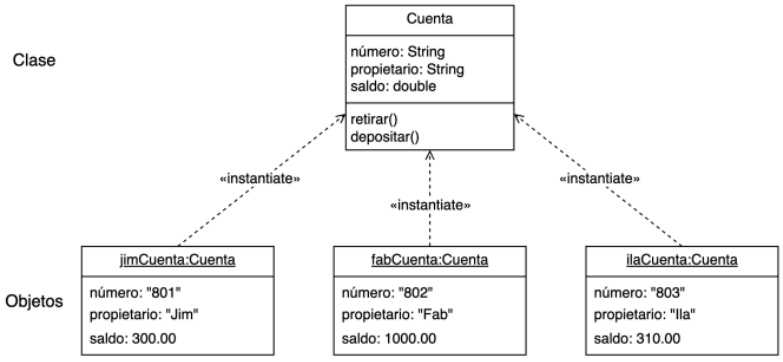
-Vínculo unidireccional: **Solo uno de los elementos tiene conocimiento del otro y puede interactuar con él, pero no al revés**.



·Estereotipos → Existen dos estereotipos en las relaciones de dependencia entre objetos y clases.

-instanceOf: **Índica que el objeto es una instancia de la clase especificada.**

-instantiate: **Indica una relación de dependencia donde se muestra explícitamente qué clase tiene la responsabilidad de crear las instancias de otra**.



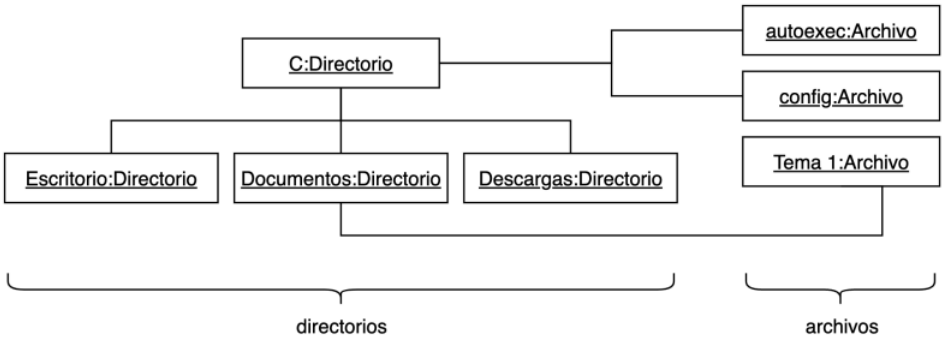
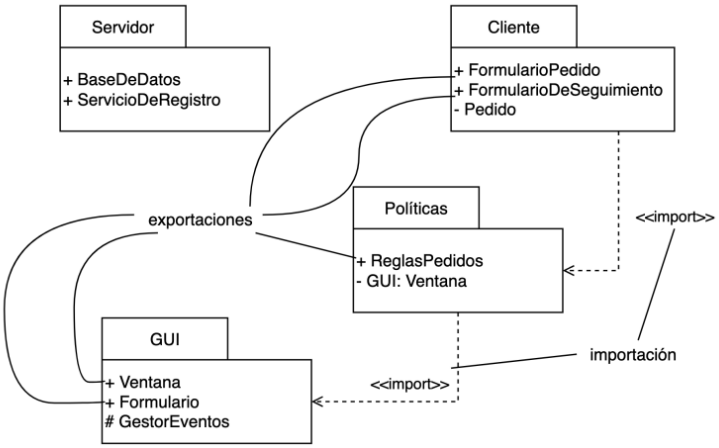


Diagrama de Paquetes:

Los **paquetes se usan para organizar elementos de modelado en grandes grupos**, permitiendo manipular elementos como una unidad.

Un **paquete puede incluir clases, interfaces, componentes, nodos, casos de uso, diagramas y otros paquetes**. Elementos de cada paquete solo pueden estar en uno, y pueden ser visibles o no.

·Importación → Esta permite que los paquetes accedan a elementos públicos de otros. Se representa con <<import>>. **Si un elemento es visible en un paquete, también lo es en sus paquetes anidados**. Los **paquetes internos pueden ver todo lo del paquete contenedor.**



·Generalización → Permite definir familias de paquetes relacionados. Sigue el principio de sustitución.

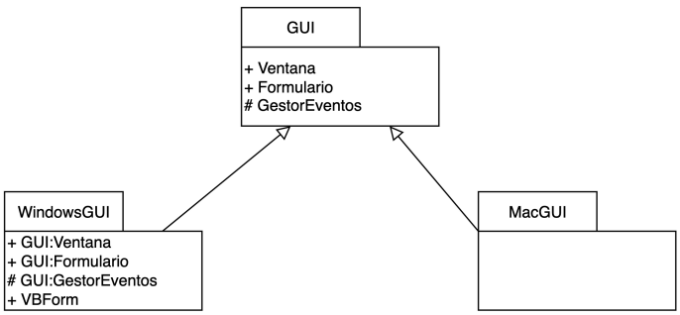


Diagrama de Componentes:

Este diagrama muestra **como se divide un sistema en componentes y las relaciones entre ellos**. Tiene una mayor nivel de abstracción que el de clases. Cada componente puede implementarse con mas de una clase en tiempo de ejecución. Permite modelar la vista lógica del sistema, bases de datos fisicas o codigo fuente.

·**Componente** → Es **una unidad autónoma y reemplazable, definida por interfaces**. Su objetivo es **encapsular funcionalidades del sistema**. Puede tener estereotipos. Hay tipos:

-Ejecutables: se ejecutan de forma independiente.

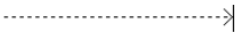
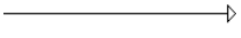
-Librerías: conjunto de objetos (estática o dinámica).

-Tabla: en una base de datos.

-Archivo: contiene código fuente o datos.

-Documento: representa información o reportes.

·**Interfaces** → Es un **elemento clave** del modelo arquitectónico**. Definen un conjunto de servicios públicos que otros pueden usar**. **Separan la especificación** (qué hace) **de la implementación** (cómo lo hace).

Relaciones	Notación
Dependencia	
Generalización (Herencia)	
Interfaces proporcionadas	
Interfaces requeridas	

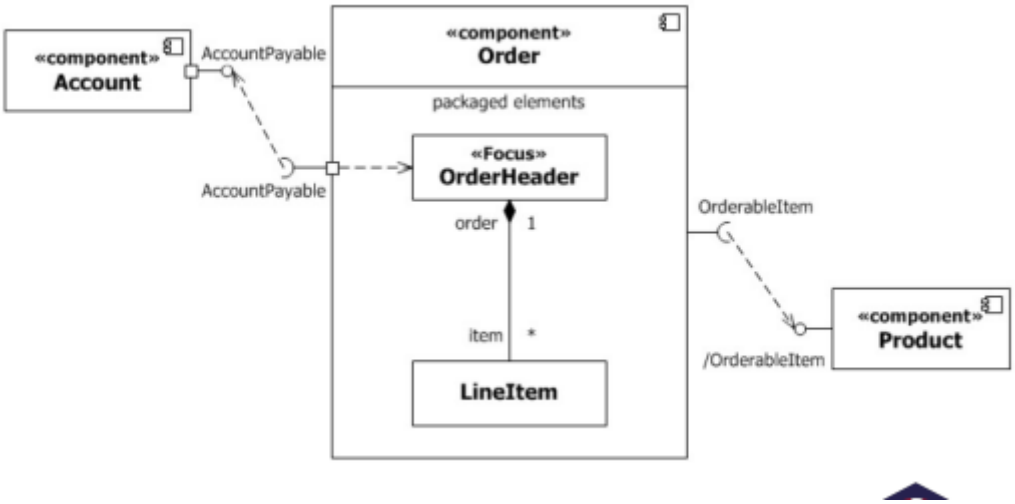
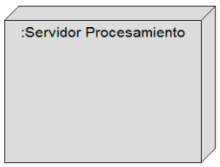


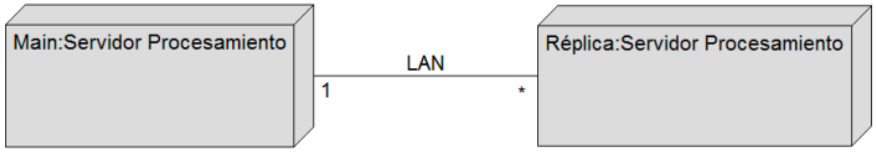
Diagrama de Despliegue:

Este muestra la topología hardware del sistema. **Relaciona elementos del modelo con información**. Este lo que hace es **indicar la distribución de los componentes**, **evaluar el rendimiento del sistema** y **analiza redundancia**, etc.

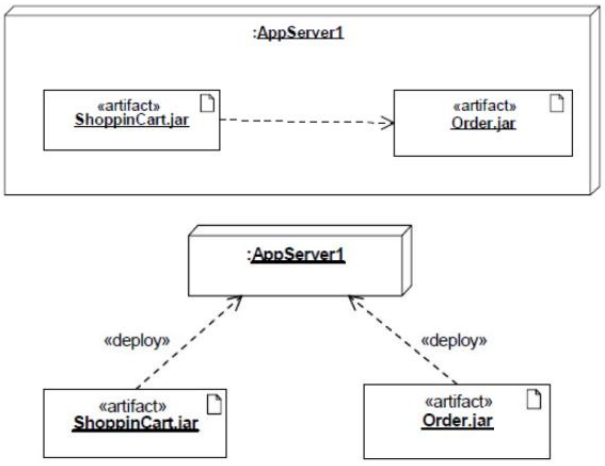
·**Nodos** → Son un **elemento físico en tiempo de ejecución que representa un recurso computacional**. Modelan la **topología del hardware del sistema**.



·**Relaciones** → Conectan los nodos del diagrama. Pueden representar el tipo de comunicación y la cardinalidad.



·**Relaciones** → **Representan elementos de implementación**. Se despliegan en nodos y pueden ser archivos fuente, ejecutables, librerías, scripts, tablas, documentos.



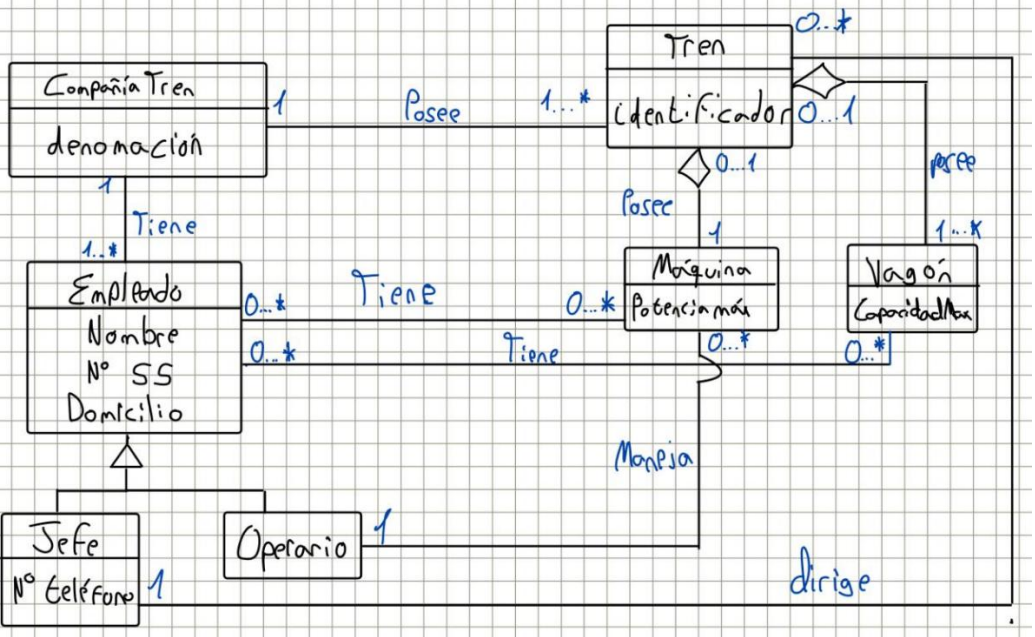
EJERCÍCIOS TEMA 6

Problema 1 – Diagrama de Clases

Modelar parte de la funcionalidad requerida para un sistema de Gestión de Trenes de Compañías Ferroviarias correspondiente a la siguiente especificación:

- Todas las compañías (con su denominación) a considerar posee al menos un tren.
- Cada tren está compuesto por una máquina y al menos un vagón.
- Pueden existir vagones y máquinas no asignados a tren alguno.
- Cada tren tiene un código identificador propio de su compañía, los vagones una capacidad máxima y las máquinas una potencia máxima.
- Una compañía tiene al menos un empleado del que se almacenan sus principales datos, como son su nombre, número de la seguridad social y el domicilio.
- Según su trabajo estos pueden ser jefes u operarios.
- Si es jefe se almacena también el número de teléfono.
- Cada empleado puede tener asignados un conjunto de máquinas y/o vagones.
- Cada tren tiene siempre asignado su jefe (necesariamente presencial) y cada máquina tiene un operario que la conduce.

Partiendo de esta información obtener el modelo de clases de la parte del dominio del problema especificado, indicando: Clases, Información interna de cada una, Relaciones, Multiplicidad, Navegabilidad con la semántica de la relación.

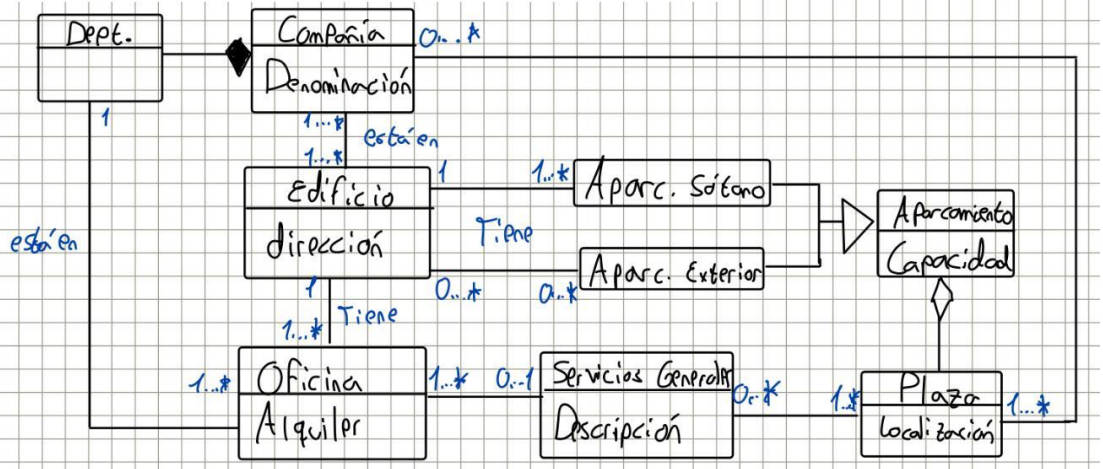


Problema 2 – Diagrama de Clases

Se ha de modelar parte de la funcionalidad requerida para un subsistema de gestión de oficinas y aparcamientos de un recinto industrial. Los requisitos de almacenamiento de información que se necesitan se nos han definido en lenguaje natural en los siguientes párrafos:

- El sistema precisa conocer la distribución de un recinto industrial y el reparto de espacio entre distintas compañías ubicadas en él.
- En el recinto industrial existen varios edificios, cada uno de ellos tiene ubicados un conjunto de oficinas y al menos un aparcamiento en su sótano, y puede tener asociado otros aparcamientos exteriores.
- Cada aparcamiento tiene un conjunto de plazas, proporcionando una determinada capacidad. Cada plaza tiene su localización. Solo hay aparcamientos externos o de sótano. Un aparcamiento exterior puede estar asociado a varios edificios.
- En el recinto industrial se ubican varias compañías, de las que interesa su denominación y el espacio asignado.
- Una compañía se ubica oficialmente en al menos un edificio. Cada compañía está compuesta por varios departamentos, y estos ocupan una o más oficinas. Una oficina solo acoge a un departamento. A su vez una compañía tiene asignadas una o más plazas de aparcamiento. Tanto edificios como plazas de aparcamiento pueden estar asignadas a una o más compañías.
- La ocupación de oficina viene dada por un alquiler para un periodo de tiempo a un precio predeterminado. Una asignación de plaza de aparcamiento viene dada mediante una autorización para un horario fijo a un determinado precio de alquiler.
- Finalmente se encuentran los servicios generales del recinto industrial, definidos mediante una descripción. Cada uno de estos servicios ocupan una o más oficinas y tienen asignadas una o más plazas de aparcamiento: todo ello de uso libre y gratuito.

Partiendo de esta información obtener el modelo de clases de la parte del dominio del problema especificado, indicando las clases con sus atributos más relevantes, y las relaciones entre ellas, incluyendo multiplicidad, navegabilidad y su nombre o roles.

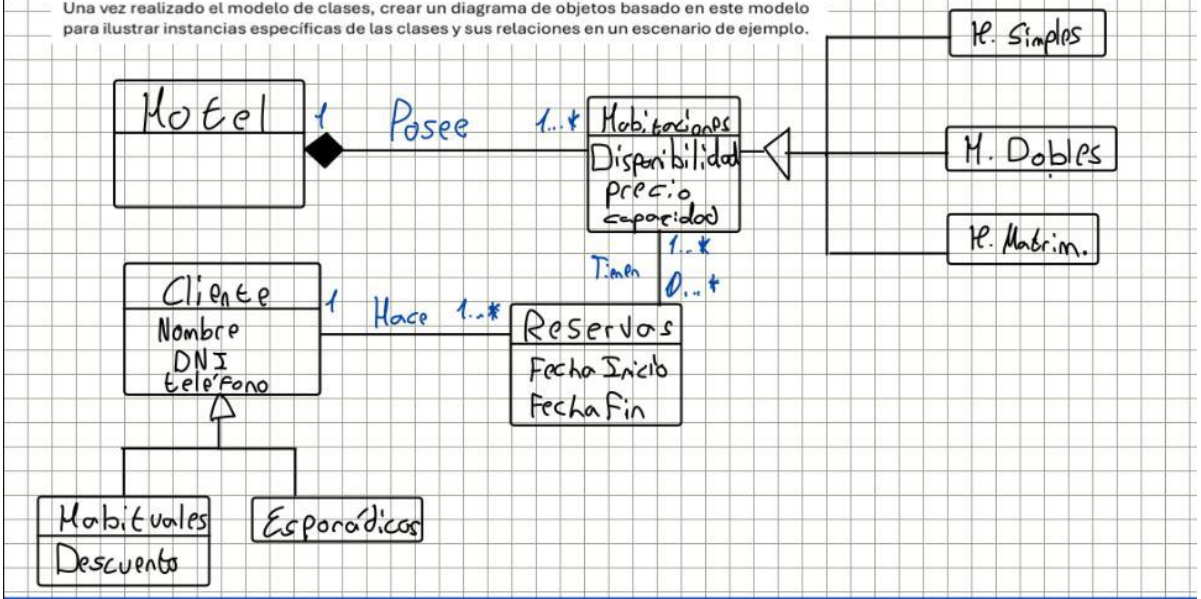


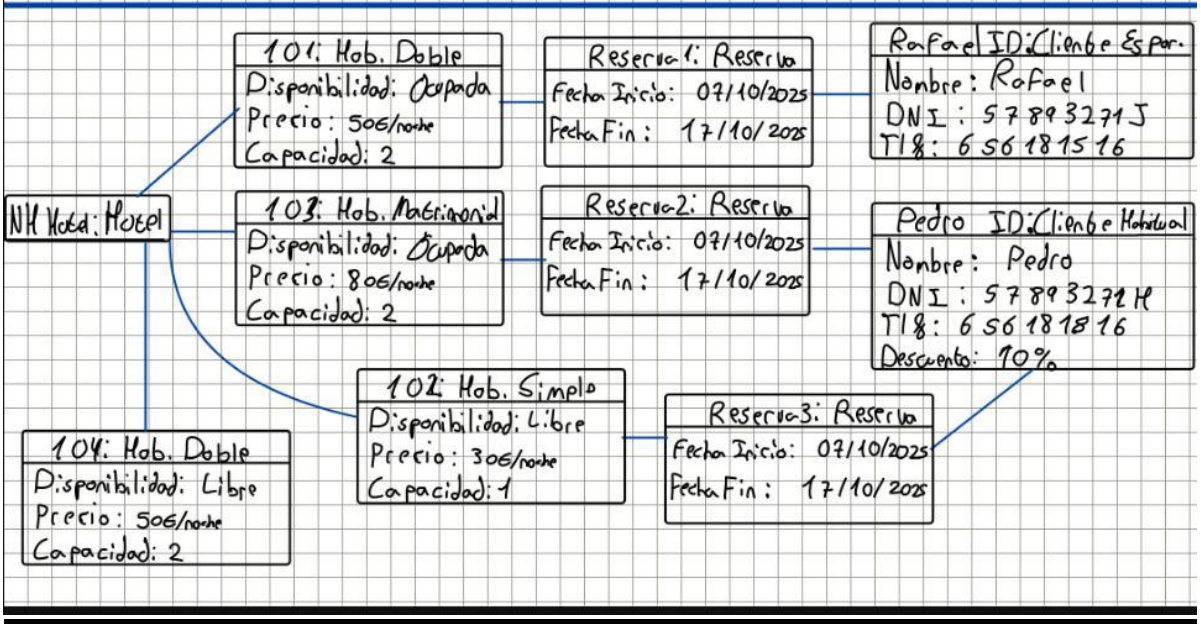
Problema 3 – Diagrama de Clases y Objetos

El dueño de un hotel te pide desarrollar un modelo de clases para un sistema que permita consultar las habitaciones disponibles y gestionar las reservas de su hotel, con las siguientes características:

- El hotel debe gestionar la disponibilidad de las habitaciones, distinguiendo entre habitaciones ocupadas y disponibles.
- Cada hotel puede tener varias habitaciones de distintos tipos y debe poder gestionarlas adecuadamente.
- Existen tres tipos de habitaciones: simples, dobles y matrimoniales. Cada tipo de habitación tiene características propias, incluyendo un precio y la capacidad máxima de huéspedes que pueden variar según el tipo de habitación.
- El sistema debe registrar dos tipos de clientes: habituales y esporádicos. Para cada cliente se almacenará su nombre, documento de identidad y teléfono. Los clientes habituales cuentan con un descuento especial.
- Cada reserva contiene información sobre la fecha de inicio y la fecha de fin de la estancia, y puede estar vinculada a una o varias habitaciones.
- Un cliente puede realizar múltiples reservas en el hotel, pero cada reserva debe estar asociada a un único cliente.

Una vez realizado el modelo de clases, crear un diagrama de objetos basado en este modelo para ilustrar instancias específicas de las clases y sus relaciones en un escenario de ejemplo.





TEORÍA TEMA 7

·Técnicas estructuradas → Son las **técnicas antecesoras a los diagramas OO para diseñar y hacer la arquitectura de los Sistemas Software**. **Según el enfoque de representación**, clasifica las **técnicas según la forma en que se representan los componentes y aspectos del sistema**. **Según el enfoque de modelado**, clasifica las **técnicas en función de los modelos del sistema que se crean**.

·Clasificación según el enfoque de representación → Este enfoque **organiza las técnicas según como se representan los elementos del sistema**, permitiendo distintos niveles de detalle:

-Gráficas: **Usan símbolos para mostrar visualmente componentes específicos, como funciones o flujos de datos**, combinándose con otras técnicas para ofrecer una visión completa.

-Textuales: **Describen en detalle los componentes de los gráficos usando un lenguaje formal**, para mayor precisión.

-Marcos o Plantillas: **Organizan información de los componentes ya descritos, documentando sus propiedades de manera estructurada**.

-Matriciales: **Verifican la precisión y consistencia de los modelos, mostrando referencias cruzadas entre componentes**.

·Clasificación según el enfoque de modelado → El **enfoque de modelado organiza las técnicas según el tipo de modelo creado**. Tipos:

-Modelado basado en la información: Este **enfoque representa las entidades del sistema y sus relaciones**, **describiendo los datos que maneja, su transformación y salida**. Su objetivo es **capturar el dominio de datos del sistema, detallando los datos que se aceptan, procesan y producen**. Sus técnicas principales son:

1) Diagramas Entidad-Relación (ER): **Representan entidades y relaciones**.
2) Diagramas de Estructura de Datos: **Muestran la organización interna de los datos**.
3) Matriz Entidad/Entidad: **Organiza las relaciones entre entidades**.

-Modelado basado en el tiempo: Este **enfoque describe cómo el sistema responde a eventos específicos, enfocándose en cuándo se activan funciones o procesos** en respuesta a estos. Su objetivo es

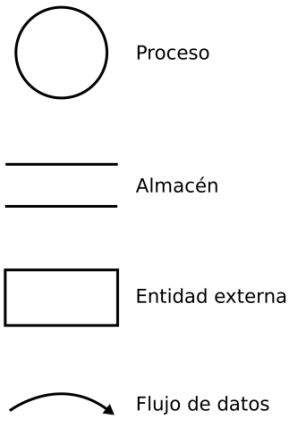
definir la secuencia de acciones activadas por eventos en el sistema. Sus técnicas principales son:

1) Diagramas de Transición de datos: **Muestran estados del sistema y transiciones según eventos**.
2) Listas de Eventos: **Enumeran eventos que desencadenan respuestas**.
3) Redes de Petri: **Modelan secuencias y paralelismos de eventos**.
4) Diagramas de Historia de Vida: **Describen el ciclo de vida de entidades según eventos**.
5) Matriz Evento/Entidad: **Relaciona eventos con entidades dependientes.**

-Modelado basado en la función: Este **enfoque representa las funciones y procesos del sistema y cómo interactúan con los datos**, detallando las tareas y transformaciones que realiza. Su objetivo es **especificar los procesos y operaciones del sistema y cómo manejan la información**.

1) Lenguaje Estructurado: **Define procesos o funciones con mayor precisión.**
2) Árboles y Tablas de Decisión: **Representan decisiones complejas y reglas de decisión**.
3) Diagramas de Descomposición Funcional: **Dividen el sistema en subfunciones**.
4) Matriz Función/Entidad: **Relaciona funciones con entidades involucradas**
5) Diagramas de Flujo de Datos (DFD)
6) Diccionario de Datos

·Diagramas de flujo de datos (DFD) → Es el **diagrama que modela las funciones del sistema y el flujo de datos entre ellas**. Representa **gráficamente cómo la información fluye y se transforma desde la entrada hasta la salida**. Usa niveles de detalle para ofrecer una visión estructurada del sistema. Tipos de objetos en el diagrama:



-Proceso: **Transforma datos de entradas en salidas**, **representando funciones del sistema**. Las salidas en base a entradas y datos adicionales. Si falta o sobran datos, es un error. Nombre y numeración único, que **contenga solo la lógica del sistema**.

-Almacenes de datos: **Representan información almacenada temporalmente**. Funcionan como **dispositivos lógicos de almacenamiento**. Repetibles en DFD para mayor legibilidad. Tiene **niveles de ubicación, alto** si conectan con muchos procesos, **o bajo** si solo lo hace con algunos específicos. **Puede formar una estructura compleja, representándose con un diagrama ER**. Toda la información se registra en el Diccionario de Datos para mantener precisión y consistencia.

-Entidades externas: **Interactúan con el sistema, pero no forman parte de él** (personas, departamentos o subsistemas). Representan **fuentes o destinatarios de información**. Es la interfaz entre el sistema y el entorno. Se **representan en el Diagrama de Contexto y también pueden aparecer en niveles inferiores**.

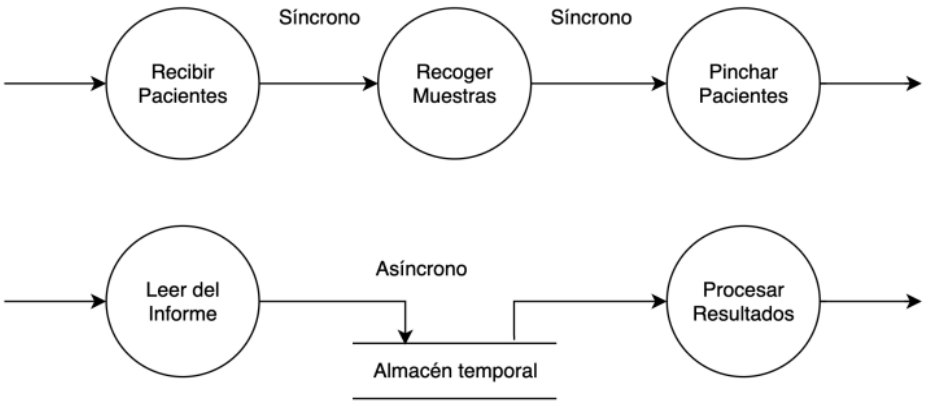
-Flujo de datos: Es el **camino por donde circulan los datos entre partes del sistema**, representado por arcos dirigidos.

Destino/Fuente	Proceso	Almacén	Entidad Externa
Proceso	SI	SI	SI
Almacén	SI	NO	NO*
Entidad Externa	SI	NO*	NO

(* significa que solo se permite si el almacén externo actúa como interfaz entre el sistema y la entidad o solo aparece en el Diagrama de Contexto)

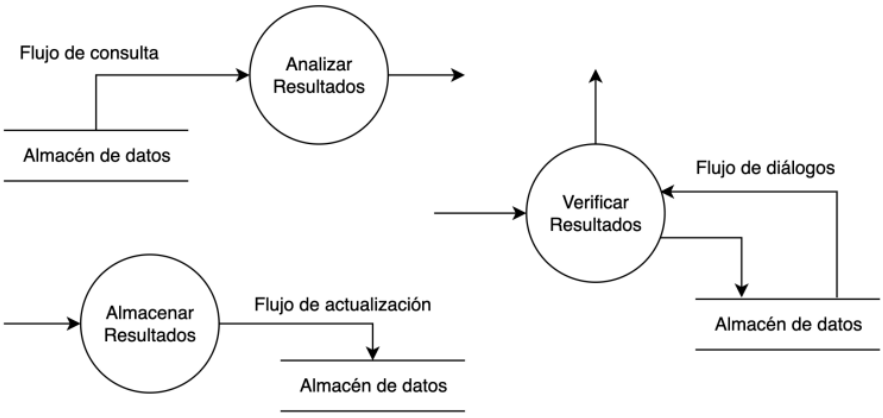
-Conexión entre procesos:

Dos procesos pueden conectarse directamente mediante un flujo de datos si la información es síncrona → el proceso receptor inicia justo cuando el proceso emisor termina



-Conexión procesos-almacenes:

1) Flujo de consulta: **Permite al proceso acceder a datos del almacén** para hacer una consulta.
2) Flujo de actualización: **Modifica el almacén con información nueva**, o eliminando/modificando información de este.
3) Flujo de diálogo: **Combina consulta y actualización,** permitiendo acceder y modificar información en un solo proceso.



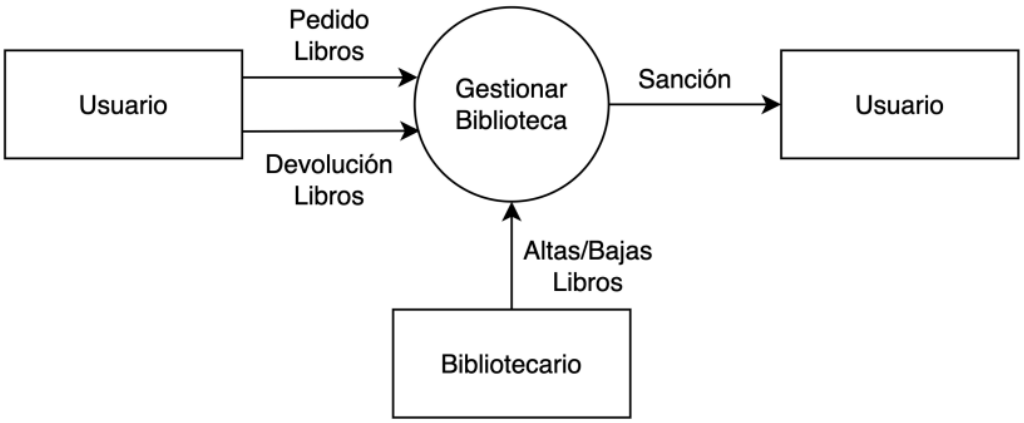
-Conexión procesos-almacenes: Es una **descomposición del sistema por capas**, que son las siguientes:

- Nivel 0: **Diagrama de Contexto**, que define los límites del sistema.
- Nivel 1: **Diagrama de Subsistemas**, mostrando las principales divisiones funcionales.
- Nivel 2: **Diagramas de Funciones de Subsistemas**, detallando las operaciones dentro de cada subsistema.

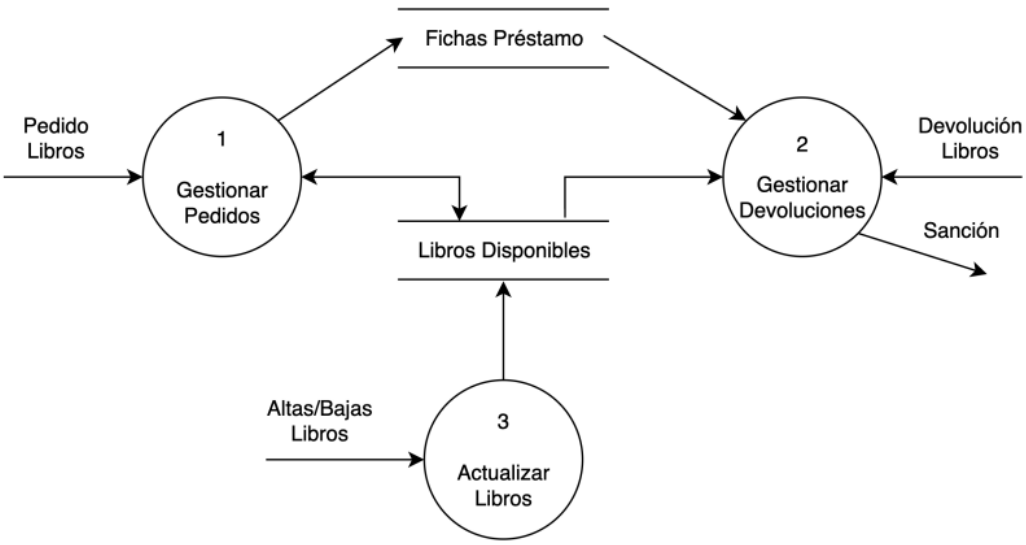
- Nivel N: **Diagramas de Procesos**, describiendo subfunciones en detalle según sea necesario.

Se debe hacer una **numeración anidada, desde el nivel 1**. Además, los flujos de datos de un diagrama hijo deben coincidir con los del diagrama padre, además de sus entradas y salidas.

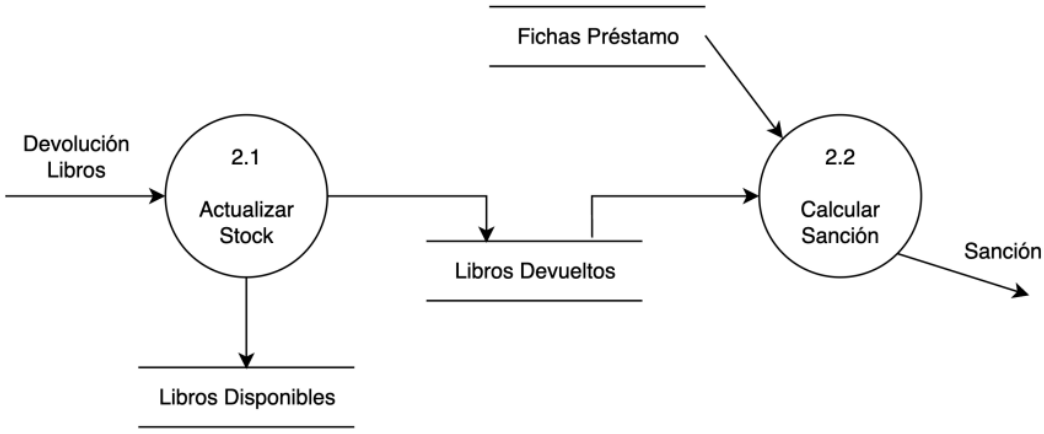
Diagrama de Contexto



Nivel 1: Gestionar Biblioteca



Nivel 2: Gestionar Devoluciones



ETC… (si es necesario)

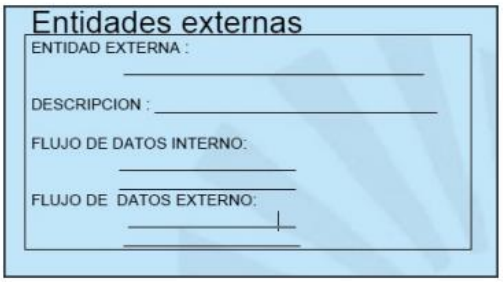
·**Diccionario de datos**→ Es una **lista organizada de datos utilizados en el sistema**. **Representa los datos que aparecen en los flujos y almacenes de los DFD**. Su objetivo es **proporcionar un listado completo y preciso de todos los datos del sistema**.

Su **contenido es: Nombre, significado y composición de los flujos y almacenes del DFD**. No solo cataloga datos, sino también flujos, almacenes y procesos, con sus descripciones y detalles. Tipos de descripciones:

-Elementos dato: Es la **unidad básica e indivisible del sistema**. + Importante.

-Estructura de datos: **Conjunto de datos elementales relacionados, usado en flujos y almacenes de datos**.

Esquemas para realizar DD



EJERCICIOS TEMA 7

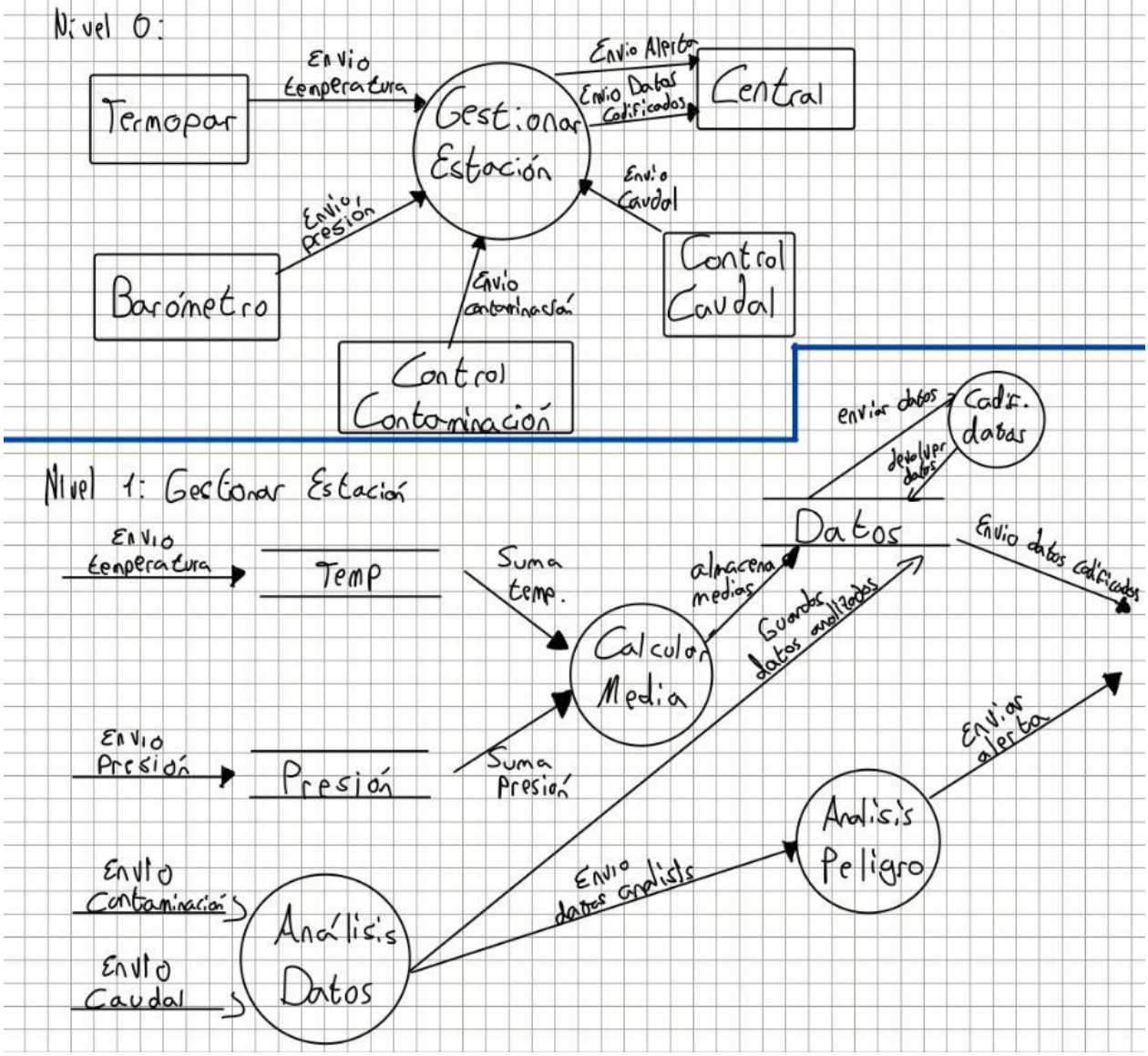
Problema 1

Se va a construir una estación meteorológica automática junto a un río. Esta estación medirá datos atmosféricos, así como niveles de contaminación del río y los transmitirá, vía satélite, a la central de datos.

Las especificaciones de funcionamiento son estas:

- La temperatura se mide a través de un termopar, estas medidas se realizan cada minuto. Cada 10 minutos se hace la media de las temperaturas leídas y se almacena su valor.
- Los datos de la presión se leen con un barómetro cada cuarto de hora y se calcula y guarda su media cada hora.
- También cada hora, se analizan 3 parámetros de nivel de contaminación de las aguas y se registran sus valores. Si algún parámetro pasa cierto umbral de peligro se genera una señal de alarma y se envía automáticamente a la central.
- Así mismo se mide el caudal del río cada 2 horas. Si se produce una crecida de forma brusca se envía una señal de alarma.
- Cada 2 horas la estación automática recopila sus datos y los transmite a la central vía satélite. Para ello, previamente tiene que codificar dichos datos en un formato estándar de control de errores para realizar transmisiones tolerantes a fallos.

Se pide realizar un Diagrama de Flujo de Datos que modele el sistema anterior considerando solo los dos primeros niveles.



TEORÍA TEMA 8

·Diseño de Software → Este se enfoca en **crear productos de software de calidad mediante principios y buenas prácticas**. Busca la **mejor solución para cumplir los requisitos**, uniendo el mundo de los humanos con el de las maquinas. Hay varios conceptos a entender:

-Abstracción: Trata **de resolver problemas en diferentes niveles de detalle**.

-Arquitectura: **Define la estructura del software y sus interrelaciones** utilizando **patrones arquitectónicos para resolver problemas recurrentes**.

-Patrones: **Soluciones reutilizables a problemas comunes** de diseño.

-Separación de intereses: **Dividir un problema en partes independientes**, para **facilitar la solución de problemas y reducir complejidad**.

-Modularidad: **Evita diseños monolíticos** (entramado de control, datos y procesos) **que hacen que el software sea difícil de comprender**.

-Ocultación de la información: **Limita datos y acceso a estos al usuario** promedio.

-Independencia funcional: **Cada módulo cumple una función diferente**. Mejora a la hora de desarrollar y mejora en la reusabilidad.

-Refinamiento: **Proceso iterativo de diseño de arriba hacia abajo para descomponer funciones abstractas en elementos concretos**.

-Refactorización: **Reorganiza el código sin cambiar su comportamiento**.

·Cohesión → **Evalúa que tan relacionadas están las responsabilidades y las funciones dentro de un módulo**.

-Alta cohesión: **Aquí hay funciones relacionadas y enfocadas en un tema**. Fácil de mantener y reutilizar.

-Baja cohesión: **Aquí hay funciones dispersas y sin relación clara**. Difícil de comprender y actualizar.

·Acoplamiento → **Mide la interdependencia entre diferentes módulos del sistema**.

-Bajo acoplamiento: **Mínima interdependencia entre módulos**. Cambios no afectan a otros módulos.

-Alto acoplamiento: **Módulos muy interconectados**. Cambios afectan a todo el programa, muy peligroso.

Alta cohesión + Bajo acoplamiento = Software fácil de mantener y escalar

·**DRY** (Don't Repeat Yourself) → **Evitar duplicación de código**.

·**YAGNI** (You Ain't Gonna Need It) → **No hacerlo hasta que sea necesario**.

·**KISS** (Keep It Simple, Stupid) → **Mantenerlo simple**.

·**SOLID** → **Conjunto de cinco reglas fundamentales para el diseño de clases en POO**. Su objetivo es promover un **código limpio, comprensible y mantenible**. Estas son:

-Responsabilidad Única (**S**RP): **1 clase, 1 única responsabilidad**.

-Abierto-Cerrado (**O**CP): Las **clases deben ser extensibles, no modificables**.

-Sustitución de Liskov (**L**SP): **Las clases derivadas deben poder sustituir a sus clases base sin problemas**.

-Segregación de Interfaces (**I**SP): **No obligues a una clase a implementar interfaces que no usa**.

-Inversión de Dependencias (**D**IP): **Las clases de alto nivel no deben depender de clases de bajo nivel, sino de interfaces o abstracciones comunes**.

·**Diseño de Software** → Son las **diferentes etapas de abstracción por las que pasa una idea hasta convertirse en un sistema técnico detallado**. Son los siguientes:

-Diseño de datos: Para **organización y gestión eficiente a los datos**. Algunos elementos claves son:

1) Estructura de datos: Tablas, relaciones, claves, etc.
2) Normalización: eliminación de datos repetidos.
3) Selección de estructuras eficientes: vectores, listas, etc.

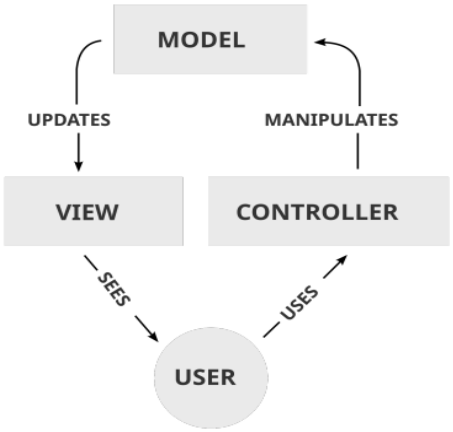
-Diseño arquitectónico: Para **definir la estructura general del sistema**. Algunos elementos claves son:

1) Identificación de módulos principales: Interfaz, lógica de datos…

2) Patrones arquitectónicos: MVC, microservicios, cliente-servido…
3) Diagramas arquitectónicos: Representación gráfica

Para ello se puede usar el patrón Modelo-Vista-Controlador (MVC). Este divide la aplicación en 3 componentes:

1) Modelo: **Gestiona los datos y la lógica del negocio**.
2) Vista: **Presenta los datos de forma interactiv**a.
3) Controlador: **Maneja eventos del usuario**.



Diseño de interfaz: Para **definir como interactúan los módulos**. Algunos elementos claves son:

1) Diseño de interfaces de usuario: UI.
2) Interfaces de programación: API.
3) Interacción con sistemas externos: REST, SOAP.

-Diseño procedimental o de funciones: Para **definir algoritmos y procesos internos del sistema**. Algunos elementos claves son:

1) Diseño de algoritmos: Pasos específicos.
2) Diagramas de flujo: Representación lógica.
3) Pseudocódigo: Descripción textual.

·Patrones de diseño → Son **soluciones reutilizables para problemas comunes en el desarrollo de software**. Un **patrón es una plantilla, no la solución** directa. Categorías:

-Patrones creacionales: **Abstracción del proceso de creación de objetos**, hace el **sistema independiente de cómo se crean y componen los objetos**. Algunos patrones son:

1) Abstract Factory: **Permite producir familias de objetos relacionados sin especificar sus clases concretas**.

Por ejemplo, en vez de crear una clase “Silla” con atributos de ambas, creas unas clase silla con atributos generales y creas dos hijas que sean “Silla antigua” y “Silla moderna” con sus atributos específicos.

2) Singleton: **Garantiza que solo haya una instancia de una clase, controlando la creación de instancias**, y proporciona un **acceso global a ella**.

-Patrones estructurales: Se enfocan en **cómo combinar clases y objetos para formar estructuras más grandes**. Algunos patrones son:

1) Bridge: **Permite dividir una clase grande,** o un grupo de clases estrechamente relacionadas**, en dos jerarquías separadas**.

   Por ejemplo, si quiero añadir colores a las clases “Círculo” y “Cuadrado”, en vez de crear 2 variables para almacenar ese dato, podemos crear un padre, para estas clases, que sea “Figura”, y que estas hereden el color de la clase.

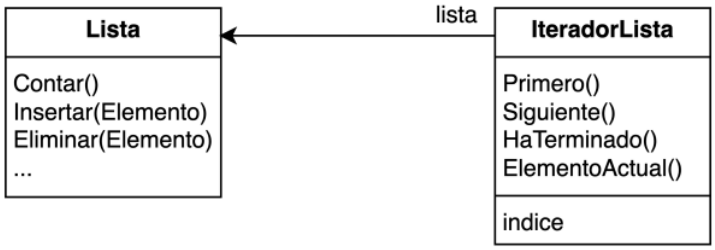
2) Composite: **Permite tratar objetos individuales y composiciones de objetos de manera uniforme**. Organiza los objetos en una jerarquía en forma de árbol, donde los nodos pueden ser objetos simples o compuestos.

   Imagina que tienes un sistema de archivos en tu ordenador: tienes “Archivos” y tienes “Carpetas". Una carpeta puede contener archivos, pero también puede contener otras carpetas. El patrón Composite permite que, si quieres calcular el tamaño total, no te importe si estás tocando un archivo o una carpeta.

-Patrones de comportamiento: Están **relacionados con algoritmos y asignación de responsabilidades**, además, **definen patrones de comunicación entre objetos**. Algunos patrones son:

1) Iterator: **Permite acceder secuencialmente a los elementos de un objeto sin exponer su estructura interna**.

   El iterador es responsable de saber qué elementos se han recorrido y cuál es el actual



2) Observer: **Establece una relación de dependencia uno a muchos entre objetos. Cuando un objeto cambia de estado,**

todos los objetos dependientes, **observers, se actualizan automáticamente.**

TEORÍA TEMA 9

·Técnicas de V&V (Verificación y Validación) → Son procesos de **análisis y pruebas que se encargan de satisfacer la especificación del software** y **entregar la funcionalidad esperada**. Sus objetivos son:

-Mejorar la calidad

-Detectar y corregir errores

-Garantizar características claves: Consistencia, confiabilidad, utilidad y eficacia.

-Asegurar utilidad

-Patrones creacionales

·Verificación → Asegurar que **cumple con los requisitos funcionales y no funcionales** y **confirmar que la implementación es técnicamente correcta**. En resumen, que lo hecho, esta bien hecho.

·Validación → Evaluar si **satisface las necesidades del cliente** y **confirmar que es útil**. En resumen, que lo hecho, es lo que se quería.

·Inspecciones del software → Son **técnicas estáticas de validación para revisar diagramas basados en requisitos, diseño y/o código fuente**. Su **objetivo es detectar errores, omisiones o anomalías**. No requieren de ejecución del código, aplicables a cualquier etapa del sistema y muy efectivas. A su vez, no garantizan que el software sea operativo, no evalúan fiabilidad y detectan fallos a nivel de módulo, no del sistema.

·Pruebas del software → Son **técnicas dinámicas de validación**, donde se ejecuta el código. Su **objetivo es detectar fallos y evaluar funcionalidad comparando el comportamiento real con el esperado**. Tipos de prueba:

-Pruebas de validación: **Confirman que el software satisface requisitos y evalúan el rendimiento y fiabilidad**.

-Pruebas de defectos: **Detectan inconsistencias con la especificación y revelan errores fuera del uso esperado**.

·**Depuración del software** → Es el **proceso de localizar y corregir errores detectados en V&V**. Tiene **herramientas que ayudan a analizar y solucionar problemas**, pero los **errores pueden manifestarse lejos de su origen**.

·**Pruebas** → Ejecución de un sistema bajo condiciones especificadas para observar y evaluar resultados.

·**Caso de prueba** → Conjunto de entradas, condiciones y resultados esperados para un objetivo específico.

·**Defecto** → Error en el software causado por una acción humana, como un proceso o paso incorrecto.

·**Fallo** → Incapacidad de un sistema para cumplir funciones según los requisitos. Es un desvío del comportamiento esperado, visto por el usuario.

·**Error** → Acción humana que lleva a un resultado incorrecto.

·**Objetivo de las pruebas software** → Ya que no se puede probar todas las posibles combinaciones en un sistema, el **objetivo en estas es, en base a pruebas, encontrar errores y defectos en el programa**.

·**Principios claves de las pruebas software**:

-Definir resultados esperados: **Cada caso de prueba debe especificar qué resultado se espera**, para compararlo con el real.

-Evitar auto-pruebas: **El programador no debe probar su propio código**, ya que puede estar sesgado.

-Inspección cuidadosa: **Revisar a fondo los resultados para detectar posibles defectos**.

-Cobertura de casos: **Incluir datos de entrada válidos e inválidos en los casos de prueba**.

-Centrarse en: **Probar si el software no hace lo que debe de hacer** y **si el software hace lo que no debe de hacer**.

-Recursos adecuados: No asumir que no hay defectos, **siempre asignar suficientes recursos a las pruebas**.

-Encontrar múltiples defectos: **Si se encuentra un error, es probable que haya más**.

-Creatividad en las pruebas: Las pruebas son tan creativas como el desarrollo del software.

-Planificación sistemática: **Diseñar pruebas de manera organizada para detectar la mayor cantidad de defectos con el menor esfuerzo**.

·Pruebas funcionales → Se centran en **verificar que el software cumpla con los requisitos funcionales** (lo que el **sistema debe hacer**). Pruebas funcionales aseguran que el software **cumpla con las expectativas definidas sin considerar su implementación interna**. Tipos:

-Pruebas unitarias: **Prueban funciones individuales**.

-Pruebas de componentes: **Prueban módulos o unidades completas**.

-Pruebas de integración: **Comprueba que los módulos interactúan correctamente**.

-Pruebas del sistema: **Evalúan el sistema en su entorno**.

- Pruebas de Aceptación (UAT): **: Confirman que el sistema satisface al cliente o usuario final**.

-Pruebas de regresión: **Verifican que los cambios no afecten las funciones existentes**.

-Pruebas de UI: **Evalúan la interacción con la interfaz gráfica.**

- Pruebas Alfa y Beta: **Pruebas que están hechas internamente antes de lanzar el producto** (Alfa) y **que están hechas por usuarios finales** (Betas).

·Pruebas NO funcionales → **Evalúan aspectos no relacionados con funcionalidades específicas, sino con atributos de calidad del sistema**, como el rendimiento, la seguridad y la usabilidad (como el sistema realiza las funciones). Tipos:

-Pruebas de rendimiento: **Prueban los límites del sistema**. Tipos:

1) Pruebas de carga: **Comportamiento del sistema con aumento de usuarios** o transacciones.
2) Pruebas de estrés: **Evaluación de límites del sistema bajo condiciones extremas**.
3) Pruebas de estabilidad: **Funcionamiento continuo durante largos períodos**.
4) Pruebas de escalabilidad: **Capacidad para manejar incrementos en la carga**.

-Pruebas de seguridad: **Evalúan las vulnerabilidades del sistema frente a ataques**.

-Pruebas de usabilidad: **Evalúan la facilidad de uso e intuitividad de la interfaz para el usuario final**.

-Pruebas de compatibilidad: **Verifican el funcionamiento correcto en diferentes dispositivos, sistemas operativos y navegadores**.

-Pruebas de localización: **Aseguran que el software funcione correctamente en diferentes idiomas**, regiones y formatos de datos.

·Desarrollo guiado por pruebas (TDD) → Es una **metodología que pone las pruebas en el centro del desarrollo de software**. Este sistema **hace pruebas antes del código**, sirviendo para guiar el desarrollo, **y usan código mínimo**, para evitar complejidad y superar las pruebas. Ventajas:

-Diseño enfocado en requisitos: **Asegura el cumplimiento de los requerimientos funcionales**.

-Mayor calidad del código: **Reduce errores y facilita el mantenimiento**.

-Documentación implícita: **Pruebas describen el comportamiento esperado del sistema**.

-Confianza en cambios: **Las pruebas automatizadas detectan regresiones**.

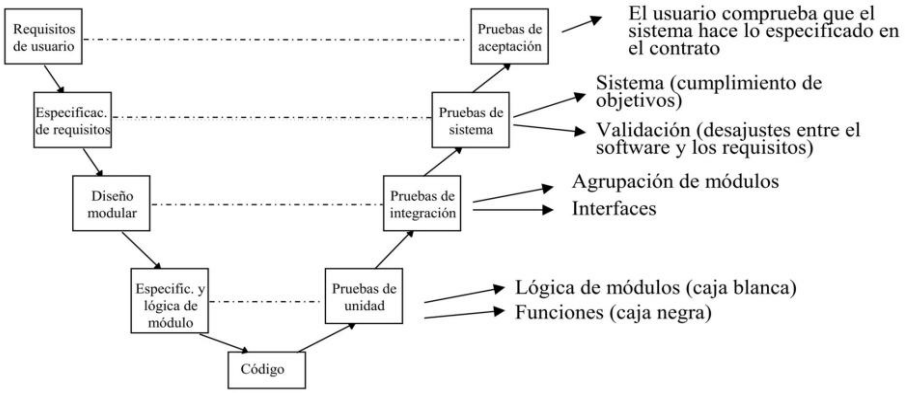
Flujo de trabajo del TDD:

-RED: **Se escribe una prueba que falla**, para garantizar que la prueba es válida y la funcionalidad no existe.

-GREEN: **Se arregla el sistema para que pase la prueba**. Se implementa el código mínimo para pasarla.

-REFACTOR: **Mejorar el código para introducirlo en el sistema**. Optimiza el código manteniendo el cumplimiento de las pruebas.

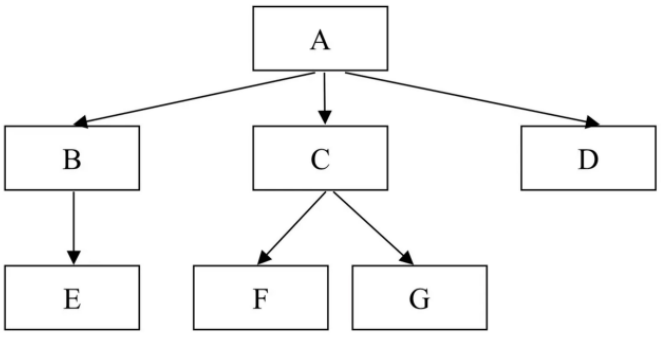
·Estrategia aplicación de pruebas → Pretende **integrar el diseño de los casos de prueba en pasos bien coordinado**s, creando niveles de prueba con objetivos específicos. **También permite la coordinación del personal del desarrollo**.



-Prueba unitaria: **Pruebas formales que permiten declarar que un módulo está listo y terminado**. Se puede probar más de un módulo a la vez. Estas pruebas suelen realizarlas el propio personal de desarrollo, **evitando que sea el creador de estos**.

-Prueba integración: **Están relacionadas con la integración de los componentes del software hasta obtener con el producto global que debe entregarse**. Implican una secuencia ordenada de pruebas, desde los módulos hasta el sistema completo. Su **objetivo es la prueba del flujo de datos módulos**. Tipos:

1) Integración incremental: Se **añade un módulo a la vez al conjunto ya probado,** aumentando progresivamente el número de módulos, **hasta formar el sistema completo**. Tipos:
   1) Ascendente: **Comienza desde los módulos más bajos**. Se agrupan los módulos de bajo nivel que realizan funciones específicas. Se **crea un módulo impulsor (driver) para cada grupo** que **introduce datos de prueba y recoge los resultados**.
   2) Descendente: **Comienza desde los módulos principal**. Se comienza haciendo pruebas con módulos ficticios. Poco a poco, se van introduciendo módulos reales. Se puede hacer **en profundidad**, completando cada rama, o **en anchura**, completando cada nivel.



2) Integración no incremental: Se **prueba cada módulo individualmente y luego todos se integran y prueban de una vez**. Cada módulo necesita un módulo impulsor y X módulos ficticios par hacerle las pruebas. **Una vez probado todos los módulos, se juntan todos y se prueba**.

-Prueba del sistema: Es el proceso de **prueba de un sistema integrado de hardware y software para comprobar el cumplimiento de los requisitos funcionales**, el **funcionamiento y rendimiento en las interfaces** hardware, software, de usuario y de operador, **adecuación de la documentación** y **ejecución y rendimiento en condiciones límite** y de sobrecarga.

-Prueba de aceptación: Es una prueba planificada y organizada formalmente para **determinar si se cumplen los requisitos de aceptación marcados por el cliente**. Se caracteriza por la **participación del cliente**, estar **enfocada hacia la prueba de requisitos de usuario** y **estar considerada la fase final del proyecto**.

·Técnica diseño de casos de pruebas → Su **objetivo es garantizar la detección de defectos con una cantidad eficiente de recursos**. La idea fundamental consiste en **seleccionar los casos más representativos que cubran el máximo posible de escenarios**. Si no se detectan errores, se puede decir que esta libre de defectos. Existen varios enfoques:

-Basado en requisitos: Se **diseñan pruebas para verificar el cumplimiento de los requisitos del sistema**. **Útil para pruebas del sistema**, ya que un requisito puede involucrar varios módulos.

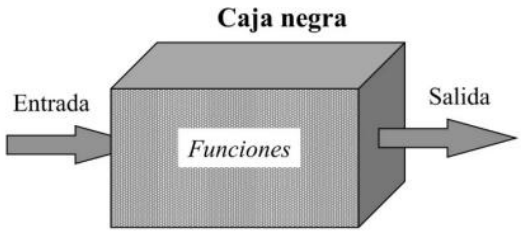
Ejemplo de Requisitos

1. Permitir búsquedas en todas las bases de datos o en un subconjunto
2. Mostrar vistas adecuadas para leer documentos
3. Cada solicitud debe incluir un identificador único (ORDER_ID)

Ejemplo de Pruebas (Requisito 1)

- Buscar en una base de datos: elementos presentes y ausentes
- Buscar en dos bases de datos: elementos presentes y ausentes
- Buscar en más de dos bases de datos: elementos presentes y ausentes
- Seleccionar una base y buscar elementos presentes y ausentes
- Seleccionar múltiples bases y buscar elementos presentes y ausentes

-Funcional (caja negra): Identifica grupos de datos (particiones) de entrada y salida con características comunes. **Diseña pruebas para cubrir todas las entradas y salidas posibles**.



Debemos **reducir el numero de prueba posibles, intentando abarcar los máximo posible**.

Técnica: Prueba de partición o Clases de equivalencia

Cada caso debe cumplir máx numero de entradas, y debe tratarse el **dominio de valores de entrada dividido en un número finito de clases de equivalencia**. **Pasos** para identificar una clase de equivalencia:

1) **Identificación de las condiciones de las entradas** del programa.
2) Se **identifican clases de equivalencia** que pueden ser de datos válidos o de datos no válidos o erróneos.
3) **Asignar un número identificador único** a cada clase de equivalencia
4) **Crear casos de prueba que incluyan todas las clases de equivalencia válidas posibles**.
5) **Crear un caso de prueba por cada clase no válida**, para asegurar que **cada condición no válida sea probada de manera aislada**.

Reglas para creación de clases:

1) Regla 1: **Si se especifica un rango de valores para los datos de entrada**, se creará **una clase válida y dos clases no validas**.
2) Regla 2: **Si se especifica un número de valores finito y consecutivo**, se creará **una clase válida y dos no válidas**.
3) Regla 3: **Si se especifica una situación del tipo «debe ser» o booleana**, se identifica **una clase válida y una no válida**.
4) Regla 4: **Si se especifica un conjunto de valores admitidos y se sabe que el programa trata de forma diferente cada uno de ellos**, se identifica **una clase válida por cada valor y una no válida**.

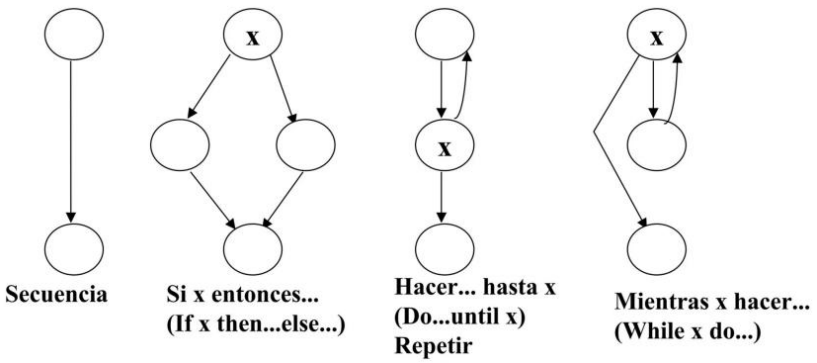
5) Regla 5: **En cualquier caso, si se sospecha que ciertos elementos de una clase no se tratan igual que el resto** de la misma, **deben dividirse en clases menores**.

Ejemplo: Aplicación bancaria en la que el operador debe proporcionar un código, un nombre y una operación

Información de entrada	Regla	Clases Válidas		Clases No Válidas	
		ID	Dominio	ID	Dominio
Código área Nº de 3 dígitos que no empieza con 0 ni 1)	3			1	no es número
	5, 1	2	200 ≤ código ≤ 999	3 4	código < 200 código > 999
Nombre para identificar la operación	2	5	6 caracteres	6 7	menos de 6 caracteres más de 6 caracteres
Orden Una de las siguientes →	4	8 9 10 11	cheque depósito pago factura retirada de fondos	12	ninguna orden válida

-Estructurales (Caja blanca): **Basado en la estructura interna del programa** y **asegura que cada parte del código se ejecute al menos una vez**. Se usan para **diseñar casos de prueba basados en caminos importantes para garantizar un nivel aceptable de detección de defectos**. Para representarlos, se usan como base los **Diagramas de Flujo de Control**. Estos se realizan siguiendo los siguientes pasos:

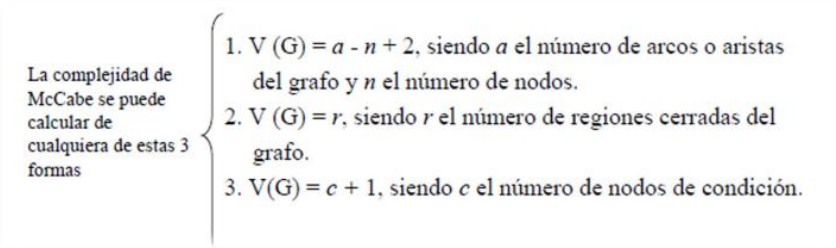
1) **Identificar y marcar cada condición en decisiones en el código** (IFTHEN, CASE-OF, WHILE, UNTIL).
2) **Agrupar las sentencias entre condiciones según estructuras** básicas
3) **Numerar condiciones y grupos de sentencias con identificadores únicos**.
4) **Reordenar condiciones en decisiones multicondicionales en orden decreciente de restricción** para facilitar el diseño de pruebas.



Técnica: Prueba del camino básico

La cobertura de caminos es una secuencia de sentencias desde el inicio hasta el final del programa**.** Es el **criterio principal en pruebas de caja blanca**. Como hay muchos caminos en un programa se propone usar **caminos de prueba, que recorren cada buble como máximo una vez**.

Mide los caminos independientes usando la **Complejidad Ciclomática V(G)**. Esta **aporta el límite superior que indica el número máximo de pruebas necesarias para ejecutar cada sentencia** al menos una vez. **Camino independiente**: **Recorrido que introduce nuevas sentencias o condiciones**.



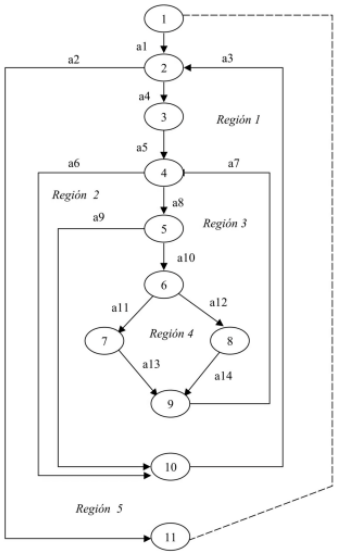
Un **nodo predicado contiene una condición y tiene dos o más salidas**.

Criterio de prueba de McCabe: Usar **tantos casos de prueba como caminos independientes**.

La experiencia en este campo asegura que: **V(G) es el mínimo de casos de prueba necesarios** y si **V(G) > 10, aumenta la probabilidad de defectos** en el módulo, **puede ser útil dividir el módulo**.

Procedimiento para derivar casos de prueba:

1. Dibujar el grafo de flujo de control
2. Calcular la complejidad ciclomática de dicho grafo
3. Identificar caminos linealmente independientes
4. Preparar casos de prueba para ejecutar cada camino básico



a) V(G) = 14-11+2 = 5
b) V(G) = 5 regiones cerradas
c) V(G) = 5 condiciones

```
   PROCEDURE imprime_media (VAR x, y : real;)
   VAR resultado : real;
1  resultado := 0;
2  IF (x < 0 OR y < 0)
3  THEN
4          WRITELN("x e y deben ser positivos");
5  ELSE
6          resultado := (x + y)/2
7          WRITELN("La media es: ", resultado);
8  ENDIF
9  END imprime_media
```

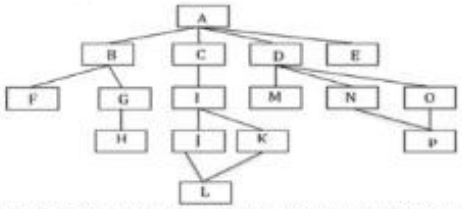
- Para mecanizar la determinación del conjunto básico de caminos se utilizará una matriz de conexiones:
  - Matriz cuadrada basada en el número de nodos
  - Las entradas representan las conexiones entre nodos (aristas)
- Cálculo de V(G):
  1. Añadir una columna con el sumatorio de cada fila menos 1
  2. La suma de esta columna más 1 es la complejidad ciclomática
- Usos adicionales de la matriz:
  - Probabilidad de ejecución de una arista
  - Tiempo, memoria o recursos requeridos por un enlace
- Una vez calculada la complejidad ciclomática por los distintos métodos se procede a preparar los casos de prueba que recorran cada camino independiente del conjunto básico

EJERCICIOS TEMA 9

9.

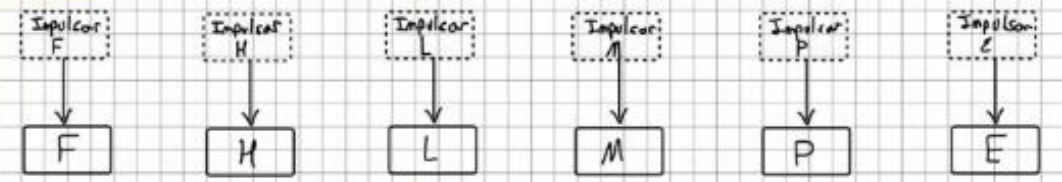
Problema 1 – Prueba de Integración

Dado el siguiente Esquema Modular, que podría representar la arquitectura de un sistema, aplicar la Prueba de Integración Incremental Ascendente para los distintos módulos que la componen, realizando las siguientes tareas:

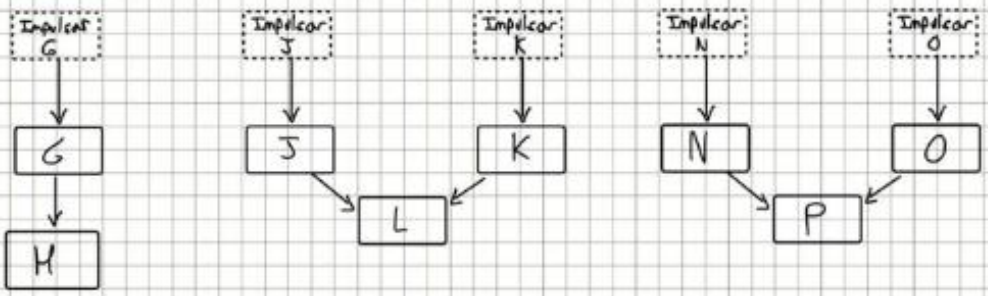


a) Indicación del número de fases necesarias para hacer la prueba completa.
b) Representar de manera grafica la secuencia de pasos/fases para realizar la integración incremental ascendente, indicando módulo e impulsor/driver en cada una de las fases.
c) Indicar los siguientes parámetros:
   - Profundidad y Anchura de dicha arquitectura.
   - Abanico de Entrada y Salida de los Módulos B, D, I y P

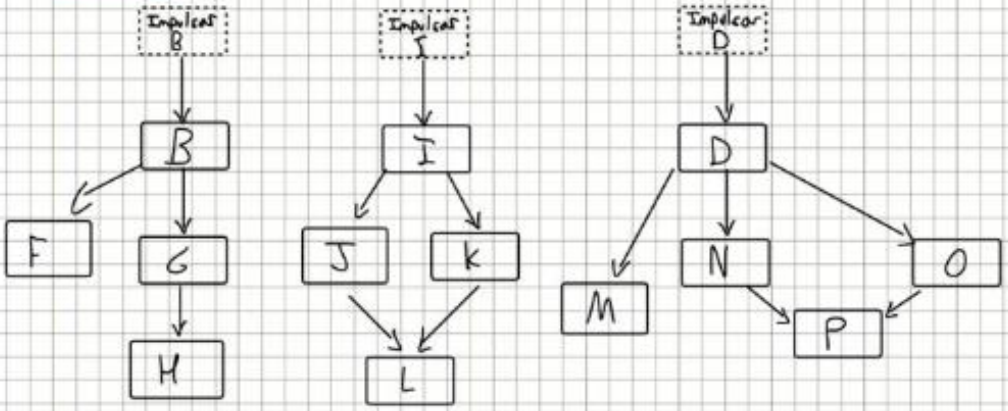
b) Fase 1



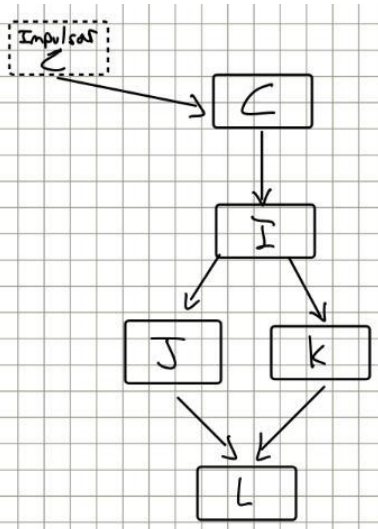
Fase 2



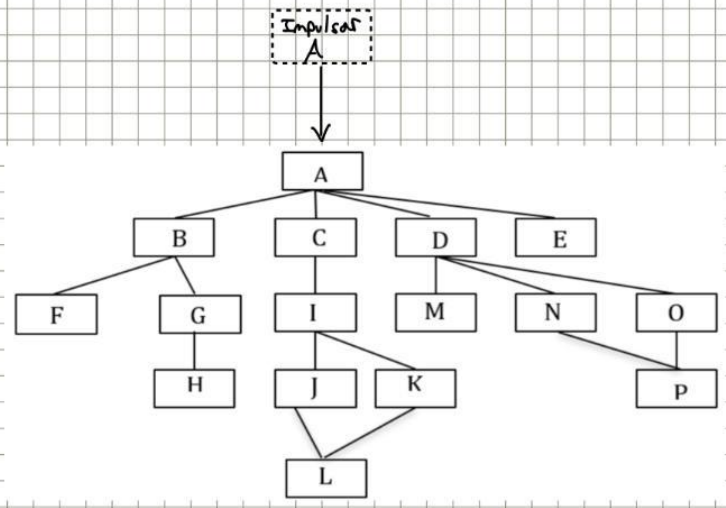
Fase 3



Fase 4



Fase 5



a) 5 fases

c) Profundidad: 5

Anchura: 6

Modulo B → Entradas: 1 Salidas 2

Modulo D → Entradas: 1 Salidas 3

Modulo I → Entradas: 1 Salidas 2

Modulo P → Entradas: 2 Salidas 0

Problema 2 – Clases de Equivalencia

Partimos de un registro que tiene los siguientes campos:

- **Codigo_Empresa**: es un campo de números positivos de 6 dígitos y que no empiezan por 0.
- **Nombre_Empresa**: es un campo alfanumérico entre 4 y 30 caracteres.
- **Actividad_Empresa**: es un campo que puede tomar cuatro posibles valores "confección", "hostelería", "limpieza", "alimentación".
- **Numero_Empleado**: es un campo numérico de tres dígitos que debe tomar valores entre 100 y 500.

Información de entrada	Regla	Clases válidas		Clases no válidas	
		ID	Dominio	ID	Dominio
Código_Empresa Nº positivos 6 dígitos (no empiezan por 0)	1	1	$100000 \leq n \leq 999999$	2 3	$n < 100000$ $n > 999999$
	3			4	no es un nº
Nombre_Empresa alfanumérico entre 4 y 30 caracteres.	2	5	$4 \leq$ caracts ≤ 30	6 7	caracts < 4 caracts > 30
Actividad_Empresa	4	8 9 10 11	"confección" "hostelería" "limpieza" "alimentación"	12	Actividad no válida
Nº_Empleados Nº positivos 3 dígitos entre 100 y 500	1	13	$100 \leq num \leq 500$	14 15	$num < 100$ $num > 500$
	3			16	no núm

Caso	(V/F)	Valores del dominio	ID Clases
1	V	135789, "Deza", "alimentación", 115	1, 5, 11, 13
2	V	879789, "NH Hotel", "limpieza", 336	1, 5, 10, 13
3	V	237769, "100 Montaditos", "hostelería", 115	1, 5, 9, 13
4	F	078935, "Bar", "hostelería", 007	2, 6, 9, 14
5	F	587169, "100 Montaditos", "chef", 115	1, 5, 12, 15
6	F	CBA769, "100 Montaditos", "hostelería", 118	4, 5, 9, 13

Problema 3 – Clases de Equivalencia

Considérese una aplicación bancaria, donde el usuario puede conectarse al banco por Internet y realizar una serie de operaciones bancarias. Una vez accedido al banco con las consiguientes medidas de seguridad (clave de acceso y demás), la información de entrada del procedimiento que gestión a las operaciones concretas a realizar por el usuario requiere la siguiente entrada:

- **Código del banco**: En blanco o número de tres dígitos. En este último caso, el primer dígito tiene que ser igual o mayor que 1.
- **Código de sucursal**: Un número de cuatro dígitos. El primero de ellos mayor de 0.
- **Número de cuenta**: Número de cinco dígitos.
- **Clave personal**: Valor alfanumérico de cinco posiciones.
- **Orden**: Este valor se introducirá según la orden que se desee realizar. Puede estar en blanco o ser una de las dos cadenas siguientes:
  - "Talonario"
  - "Movimientos"

Las clases de equivalencia derivadas para este programa. Cada una de las clases ha sido numerada para facilitar después la realización de los casos de prueba.

Información de entrada	Regla	Clases válidas		Clases no válidas	
		ID	Dominio	ID	Dominio
Código_Banco	1	1	$100 \leq cod \leq 999$	2	$cod < 100$
				3	$cod > 999$
Nº 3 dígitos que no empiece en 0, o nada		4	en blanco		
	3			5	no es número
Código_Sucursal	1	6	$1000 \leq cod \leq 9999$	7	$cod < 100$
				8	$cod > 999$
Nº 4 dígitos que no empiece en 0	3			9	no es número
Nº_Cuenta	1	10	$0 \leq n \leq 99999$	11	$n < 0$
				12	$n > 99999$
Nº 5 dígitos	3			13	no es número
Clave_Personal	2	14	5 caracts.	15	caracts < 5
alfanumérico 5 caracts.				16	caracts > 5
Orden	4	17	Talonario		
		18	movimientos	20	orden no válida
		19	en blanco		

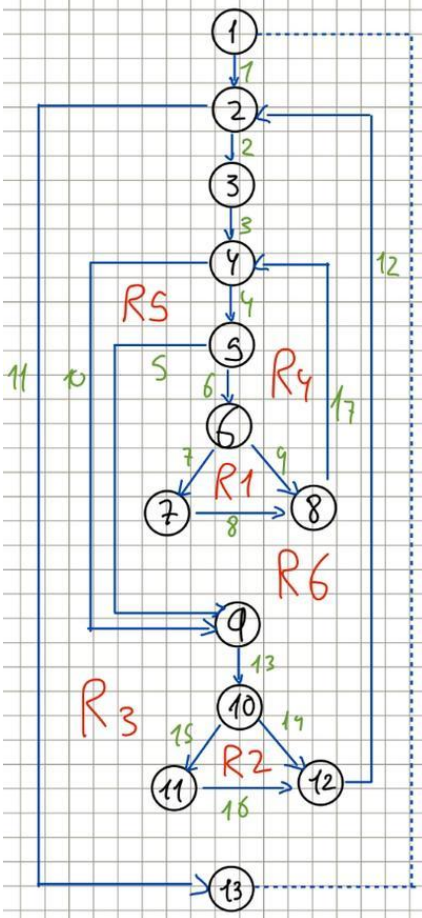
Caso	(V/F)	Valores del dominio	ID Clases
1	V	139, 4532, 99853, "GATO6", talonario	1, 6, 10, 14, 17
2	F	051, 100 000, 105793, "12345 6", bizum	2, 8, 12, 16, 20

Problema 4 – Prueba del Camino Básico

Partiendo del procedimiento denominado "Eliminar Repetidos" realizar las siguientes actividades:

- Diagrama de Flujo de Control
- Calcular la complejidad ciclomática de todas las formas posibles.
- Calcular la matriz de conexiones.
- Obtener los caminos linealmente independientes.

```
1   PROCEDIMIENTO Eliminar_Repetidos
    ENTRADAS:
    a: ARRAY DE ENTEROS con valores repetidos
    n: entero. Tamaño del array
    SALIDAS:
    b: ARRAY DE ENTEROS sin repetidos
    m: entero. Número de elementos en b
    VARIABLES:
    i, j, k : enteros
    repetido : booleano
    INICIO
2       b(1) ← a(1)
3       j ← 2
        i ← 2
4   MIENTRAS (i <= n) HACER
5       k ← 1
6       repetido ← falso
7       MIENTRAS (k <= j) Y (no repetido) HACER
8           SI b (k) = a(i) ENTONCES
9               repetido ← verdadero
10          FIN SI
11          k ← k + 1
12      FIN MIENTRAS
13      SI (no repetido) ENTONCES
14          b(j) ← a(i)
15          j ← j + 1
16      FIN SI
17      i ← i +1
18  FIN MIENTRAS
19  m ← m - 1
20  FIN
```



$V(G) = 6$

$V(G) = 5 + 1 = 6$

$V(G) = 17 - 13 + 2 = 6$

$C_1 = 1, 2, 13$

$C_2 = 1, 2, 3, 4, 9, 10, 12, 2, 13$

$C_3 = 1, 2, 3, 4, 9, 10, 11, 12, 2, 13$

$C_4 = 1, 2, 3, 4, 5, 9, 10, 12, 2, 13$

$C_5 = 1, 2, 3, 4, 5, 6, 8, 4, 9, 10, 12, 2, 13$

$C_6 = 1, 2, 3, 4, 5, 6, 7, 8, 4, 9, 10, 12, 2, 13$

$V(G) = 5 + 1 = 6$

	1	2	3	4	5	6	7	8	9	10	11	12	13	0
1		1												0
2			1										1	1
3				1										0
4					1				1					1
5						1			1					1
6							1	1						1
7								1						0
8					1									0
9										1				0
10											1	1		1
11												1		0
12		1												0
13	1													0

TEORÍA TEMA 10

·Inicio de actividades clave del ciclo de vida Software→ Son las **actividades responsables de garantizar su utilidad y sostenibilidad**. Estas son:

-Puesta en marcha: **Instalación, configuración y la capacitación**.

-Nuevas funcionalidades: **Ampliación y mejora del sistema**.

-Corrección de errores

-Adaptación a nuevos entornos: **Ajustes ante tecnologías o infraestructura**.

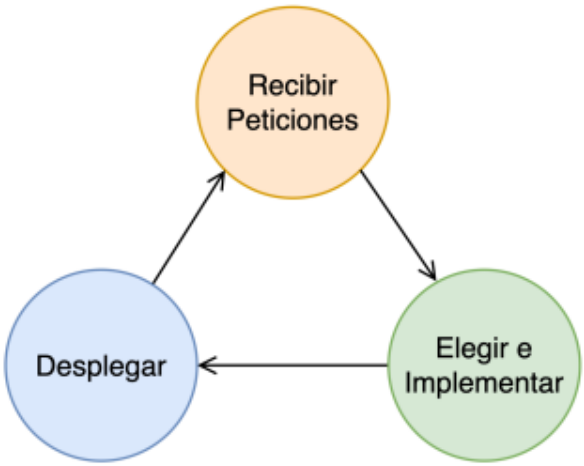
-Cambios del negocio: **Adaptación a nuevos procesos, normativas o expectativas del mercado**.

·Despliegue→ Es el **proceso de poner el software** (o sus actualizaciones) **a disposición de los usuarios en un entorno real**. Su **objetivo es combinar procesos manuales y automáticos para distribuir actualizaciones y parches** de forma eficiente. Incluye:

-Preparación y lanzamiento de nuevas versiones

-Configuración e instalación del sistema

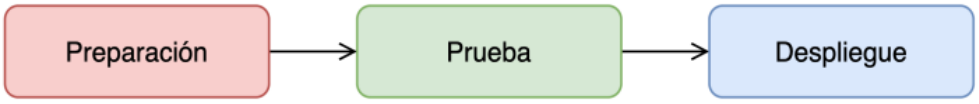
-Monitoreo del rendimiento tras la puesta en marcha



-Estados esenciales del despliegue:

1. Preparación: **Recopilación del código y recursos necesarios para la integración**
2. Pruebas: **Evaluación del código en un entorno de pruebas** y **uso de prueba** para prevenir errores.

3. Despliegue: **Integración en el entorno de producción** y **disponibilidad de nuevas funcionalidades** junto con las previas.



Existen dos tipos de despliegue:

-Despliegue Continuo (CD): Se **publican nuevas funcionalidades de forma actualizada y frecuente**.

- Integración Continua (CI): Los **cambios de código se integran constantemente en la rama principal**.

- Prácticas recomendadas para CI/CD
  - Mantener un repositorio único de código
  - Automatizar la construcción (*build*)
  - Pruebas automáticas
  - Todo el mundo hace commits diarios a la rama principal
  - Los commits deben ser correctamente integrados en la versión actual
  - Todo "bug" corregido debe ir acompañado del caso de prueba que lo encontró
  - El proceso de construcción debe permanecer rápido
  - Haz pruebas en un entorno específico (staging)
  - Acceso rápido a nuevas funcionalidades para facilitar las pruebas
  - Todo el mundo debe poder ver si la construcción falla y por qué

·**Evolución del Software**→ Es un **proceso inevitable y esencial para mantener su relevancia y utilidad en un entorno dinámico**. Los cambios pueden deberse a:

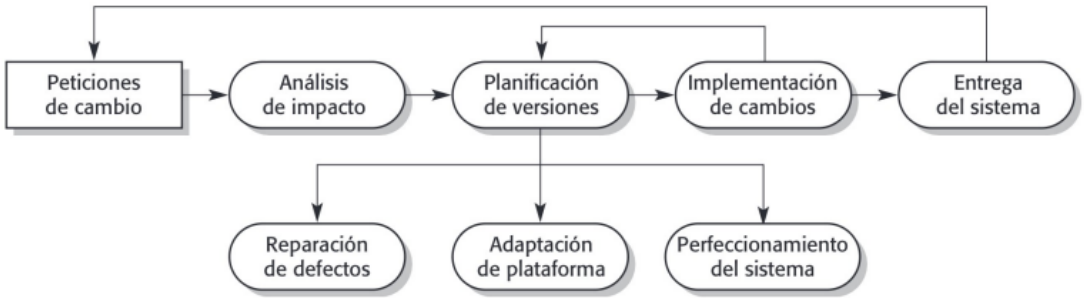
-Nuevas situaciones en el negocio

-Cambios en las expectativas de los clientes

-Necesidad de adaptarse a nuevo hardware o software

-Mejoras en el rendimiento

- Una vez aceptada la propuesta de cambios:
  - Se deben identificar las partes del sistema que se verán afectadas
  - Se debe analizar el coste y el impacto del cambio
  - Se planifica la nueva versión con los cambios a considerar (reparación de defectos, adaptaciones y nuevas funcionalidades)
  - Se implementan los cambios y se valida la nueva versión del sistema
  - Se entrega la nueva versión del sistema
- En situaciones como vulnerabilidades de seguridad o fallos graves, el proceso puede simplificarse para acelerar los cambios



·**Mantenimiento del Software**→ **Abarca los cambios realizados después de entregar el sistema**. Es **especialmente relevante en sistemas a medida, donde desarrollo y mantenimiento suelen estar separados**. Su **objetivo es la reparación de fallos**, la **adaptación a nuevas plataformas** y la **incorporación de nuevas funcionalidades o soporte para nuevos requisitos**. Tipos de cambios en el mantenimiento:

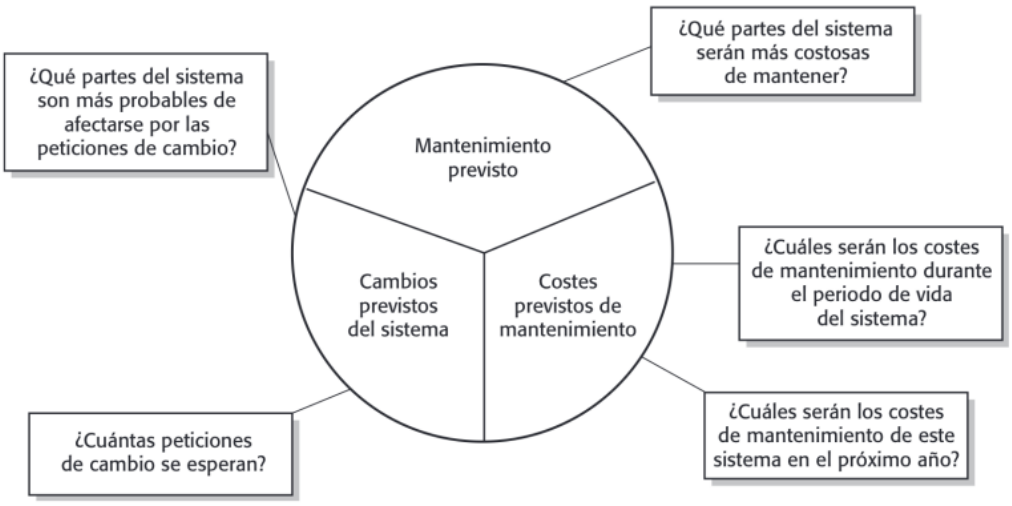
-Correcciones simples: **Solución de errores en el código**.

-Modificaciones profundas**: Rediseños o incorporación de nuevos requisitos**.

Tipos de Mantenimiento

1. Mantenimiento Correctivo
   - Corrige errores o vulnerabilidades tras el despliegue
2. Mantenimiento Adaptativo
   - Ajusta el software a nuevos entornos, plataformas o tecnologías
3. Mantenimiento Perfectivo
   - Mejora el rendimiento o añade funciones según nuevas necesidades del negocio
4. Mantenimiento Preventivo
   - Anticipa problemas mediante refactorización, mejor documentación y revisiones de arquitectura

Predicción del Mantenimiento



·**Reeingeniería** → Se aplica a sistemas heredados que**, por su antigüedad, son difíciles de actualizar**. **Se usa cuando modificar un sistema directamente no es viable** o reemplazarlo es demasiado costoso o riesgoso. Su **propósito es mejorar y reestructurar el sistema para optimizar su rendimiento, adaptabilidad y sostenibilidad**. Algunas actividades de esta es:

-Traducción del código: **Migrar el código a un lenguaje más moderno**.

-Ingeniería inversa: **Analizar el sistema para comprender su estructura y funciones**, sobre todo si no hay documentación.

-Mejora estructura: **Reorganizar y refactorizar el código para hacerlo más claro y mantenible**.

-Modularización del programa: **Dividir el sistema en módulos independientes** para mejorar la flexibilidad y escalabilidad.

-Traducción del código: **Optimizar bases de datos para mejorar el rendimiento y compatibilidad** con tecnologías actuales