matias.rojas46@mail.udp.cl.

RESUMEN SOFTWARE

1. *Ingeniería de Software*

1.1 Historias de usuario

Son la explicación general e informal de una función de software escrita desde la perspectiva del usuario final. Su propósito es articular cómo proporcionará una función de software valor al cliente".
FORMATO: Como [perfil], Quiero [objetivo del software], Para lograr [resultado]"
COMPONENTES A IDENTIFICAR:

- **Perfil:** Se refiere al rol del usuario final. Identificar el perfil del usuario final es necesario
- **Necesidad:** El objetivo que tiene la función del Software para el usuario final. Responde a cómo el usuario final utilizará una funcionalidad del software y por qué.
- **Propósito:** Propuesta de valor que tendrá el Software para el usuario final. Analiza el panorama general de operación del software considerando cómo la función de software se ajusta a tus objetivos internos.

Ejemplo:
Como usuario de una cuenta de correo electrónico, espero que mis datos se puedan guardar en las configuraciones de mi equipo personal para lograr un acceso más rápido a mi bandeja de entrada.

Se pueden escribir con el **formato tabla:**

objetivo

ID_HU	Como < Tipo de usuario>	Quiero <Desarrollar alguna tarea>	De manera de que pueda <lograr un objetivo>
1	Gerente	Ver aporte de estado de cada miembro del equipo	Asegurar que el proyecto avance según lo presupuestado.

Asegurar que no se paguen horas extras.

Se pueden escribir con el **formato escenario:**

- **Supuestos iniciales:** que esperan los usuarios al iniciar el escenario.
- **Normal:** flujo normal del evento del escenario.
- **Que puede fallar:** describir lo que puede fallar y como se maneja.
- **Otras actividades:** descripción de otras actividades que puedan ocurrir mientras sucede el escenario.
- **Estado final:** descripción del sistema cuando el escenario termina.

1.2 Ingeniería de requerimientos

Un requerimiento es la definición de las funciones, capacidades o atributos intrínsecos de un sistema de software.

Criterios de aceptación de las HU. / [Pruebas/ocasiones de pruebas]

dado que <estado previo o contexto> / GIVEN
Cuando <comportamiento o acción> / WHEN
Entonces <cambio o resultado de un comportamiento> / Then.

esenciales para su funcionamiento.

→ son lo que el usuario quiere.
→ forma de validar con el usuario lo que quiere.

- **Requerimientos de usuario**: son descripciones de los requerimientos funcionales y no funcionales de tal forma que sean comprensibles por los usuarios del sistema sin conocimiento técnico detallado.

ID	Descripción	Prioridad	Tipo	Riesgo
1	El sistema debe permitir el registro de los equipos de fútbol.	Alta	F	Alto

→ dan soporte a los requerimientos de usuario.

- **Requerimientos de Sistema:** son versiones extendidas de los requerimientos del usuario dado que agregan detalle y explican cómo el sistema debe proporcionar los requerimientos del usuario. No deben tratar de cómo se debe diseñar o implementar, deben describir el comportamiento externo del sistema y sus restricciones operativas y son requisitos funcionales pues reflejan las operaciones que el sistema llevará a cabo.

ID	Descripción	Prioridad	Tipo	Riesgo
1	El sistema debe guardar todos los datos en una base de datos relacional.	Alta	F	Alto

- **Requerimientos de Dominio**: son aquellos que se derivan del dominio de aplicación del sistema más que de las necesidades específicas de los usuarios. Se pueden considerar "un complemento" para los Requerimientos de Sistema. Se deben satisfacer para que el sistema funcione correctamente, incluyen terminología especializada y pueden restringir los requerimientos existentes o establecer cómo se deben ejecutar dichos requerimientos.

Requerimientos Funcionales → permiten que hacer el sistema. → evitar comas, y, etc.

debe ser atómico completo y coherente.

Se refiere a aquellos requisitos que definen una función del software o de sus componentes y además establecen los comportamientos de software ante input determinados.

¿Qué establecen los Requerimientos Funcionales?

- Servicios o funciones que proveerá el sistema.
- Interacción entre el sistema y su entorno.

Un requerimiento funcional típico contiene: Identificador y/o Nombre, Resumen y Descripción y Prioridad.

Número	Requerimiento	Descripción	Prioridad
RF1	LA ADMINISTRACION LLEVARA UN CONTROL DE NOTAS DE LOS ALUMNOS	El sistema tendrá las notas de los alumnos en cada curso que ha llevado y que esta cursando actualmente,	5
RF2	EL ADMINISTRADOR PODRA MODIFICAR LAS NOTAS	El sistema deberá permitir la modificación de las notas del curso del ciclo vigente	5

El <tipo usuario> <verbo futuro> <descripción>

Ej3: la app debe tener un tiempo de respuesta < 2s.

Requerimientos no Funcionales → limitaciones del sistema.

Ej2: la app debe tener sus contraseñas encriptadas con AES256

Se refiere a criterios que se pueden utilizar para evaluar la operación de un sistema en lugar de sus comportamientos específicos como la disponibilidad, fiabilidad, tiempos de respuesta, seguridad utilizada, etc.

¿Qué establecen los Requerimientos no Funcionales? → Ej: el sistema deberá ejecutarse en dispositivos Android en las últimas versiones.

- Entornos definidos para los servicios o funciones ofrecidas por el sistema.
- Restricciones para la elección en cuanto a la construcción del sistema.

ej: "El (tipo usuario, sistema, componen), (verbo en futuro), (descripción)"

"El administrador podrá crear usuarios del tipo "normal" para que puedan acceder al sistema."

Requerimientos no funcionales:

→ **confiabilidad**: disponible, recuperable, regla de los 5 nueves; 99,999%
- componentes redundantes.
- facil migración de componentes
- bajo acoplamiento.
- control distribuido
- operacion activo-pasivo o pasivo-pasivo.

→ **Integrabilidad**: integracion con otros sistemas.
- Interoperación: interoperando con otro sistema.
- API's: publicación de API's.

→ **Portabilidad**: independencia de plataforma.

→ **mantenibilidad**: integrar nuevos requerimientos funcionales
- componentes auto-contenidos
- especificación detallada
- componentes reemplazables
- evitar datos compartidos.

→ **Verificabilidad**: autotesteo, cheqeo sistema autonomo.
- transacciones dummy
- definir ciclos de procesamiento vacios
- procesos de verificación operativa.

→ **soportabilidad**: diagnostico y corrección de incidencias.
- diseño modular
- Integracion continua (CI/CD)
- Documentacion actualizada, bitacoras de proceso.
- gestion de logs

→ **Seguridad**: el sw debe manejar un cierto nivel de seguridad.
- seguridad del usuario: autenticacion (login) / autorizacion (funciones usuario).
- " en datos: encriptacion (cifrado) / integridad (datos =) / no repudio.

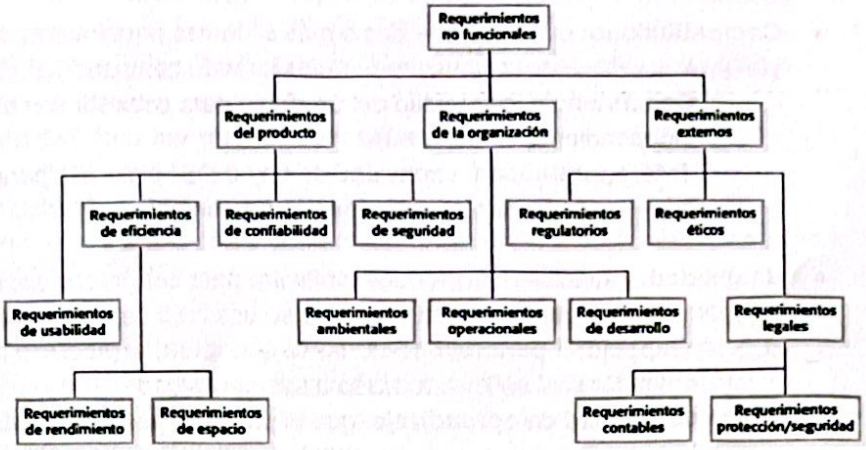
→ **escalabilidad**: capacidad del sw de tratar con multiples usuarios.

→ **performance**: desempeño de la cantidad de recursos utilizados bajo condiciones.

Arquitectura de software:

Ejemplo de requerimientos no funcionales:

- La carga del formulario inicial debe tomar, en condiciones normales de conectividad, un máximo de 3 segundos.
- El sistema deberá estar programado en lenguaje Python 3.



Tipos de requerimientos no funcionales:

SQuaRE (System and Software Quality Requirements and Evaluation): descripción de cómo los modelos de calidad pueden ser usados para los requerimientos de calidad de software.

ISO/IEC 25010

- **Adecuación Funcional:** capacidad del producto para proporcionar funciones que satisfacen las necesidades declaradas e implícitas, cuando el producto se usa en las condiciones especificadas.
  - **Completitud funcional:** grado en el cual el conjunto de funcionalidades cubre todas las tareas del usuario.
  - **Corrección funcional:** capacidad para proveer resultados correctos.
  - **Pertinencia funcional:** capacidad para entregar un conjunto apropiado de funciones para tareas de usuario especificadas.

- **Eficiencia de desempeño:** representa el desempeño relativo a la cantidad de recursos utilizados bajo determinadas condiciones.
  - **Comportamiento temporal:** tiempo de respuesta, procesamiento y ratios de throughput de un sistema cuando lleva a cabo sus funciones bajo condiciones determinadas.

→ performance: respuesta eficiente.
→ disminuir la comunicación entre componentes.
→ sistema debe tener pocos componentes.
→ bajo uso de la red y operaciones de entrada/salida.

- **Utilización de recursos**: cantidad y tipo de recursos utilizados cuando el sw funciona.
- **Capacidad**: grado en que los límites máximos de un parámetro de un producto o sistema software cumplen con los requisitos.

- **Compatibilidad:** capacidad de dos o más sistemas para intercambiar información y/o llevar a cabo sus funciones requeridas cuando comparten el mismo entorno.
  - **Coexistencia**: capacidad del producto para coexistir con otro software independiente.
  - **Interoperabilidad**: capacidad de dos o más sistemas para intercambiar información y utilizar la información intercambiada.

- **Usabilidad:** capacidad del producto software para ser entendido, aprendido, usado y resultar atractivo para el usuario, cuando se usa bajo determinadas condiciones.
  - **Capacidad para reconocer su adecuación**: el producto permite al usuario entender si el sw es adecuado a sus necesidades.
  - **Capacidad de aprendizaje**: que el producto permite al usuario aprender de la app.
  - **Capacidad para ser usado:** permitiendo que el usuario pueda operarlo y controlarlo con facilidad.
  - **Protección contra errores de usuario**: proteger a los usuarios de cometer errores.
  - **Estética de la interfaz de usuario**:capacidad de agradar y satisfacer la interacción con el usuario en cuanto a su interfaz
  - **Accesibilidad**: capacidad del producto que permite que sea utilizado por usuarios con determinadas características y discapacidades.
- **Fiabilidad:** capacidad de un sistema para desempeñar las funciones especificadas bajo condiciones y periodo de tiempo determinados.
  - **Disponibilidad**, capacidad del sistema para estar operativo cuando este se requiera.
  - **Capacidad de recuperación:** capacidad de recuperar datos afectados o restablecer el estado deseado del sistema en caso de interrupción/fallo.
  - **Tolerancia a fallos**: capacidad del sistema para operar según lo previsto en presencia de fallos.
  - **Madurez**: sastifacción de las necesidades de fiabilidad en condiciones normales.
- **Seguridad:** capacidad de protección de la información y los datos de manera que personas o sistemas no autorizados no los puedan leer y/o modificar.
  - **Confidencialidad:** capacidad de protección contra el acceso de datos e información no autorizados.
  - **Autenticidad:** capacidad de demostrar la identidad de un sujeto o un recurso.
  - **Integridad**: capacidad del sistema para prevenir accesos o modificaciones no autorizados a datos.
  - **Responsabilidad:** capacidad de rastrear de forma inequívoca las acciones de una entidad.
  - **No repudio:** capacidad de demostrar las acciones o eventos que han tenido lugar, de manera de no poder ser repudiadas.

Para lograr esto
• componentes auto-contenidos
• especificacion detallada
• componentes reemplazables.
• evitar datos compartidos.
→ Integrar nuevos requerimientos funcionales.

- **Mantenibilidad:** capacidad del producto software para ser modificado efectiva y eficientemente debido a necesidades.
  - **Modularidad:** capacidad de un sistema que permite que un cambio en un componente tenga un impacto mínimo en los demás.
  - **Reusabilidad:** capacidad de un activo que permite que sea utilizado en más de un sistema software.
  - **Capacidad para ser probado:** facilidad con la que se pueden establecer criterios de prueba para un sistema.
  - **Capacidad para ser modificado:** capacidad del producto que permite que sea modificado de forma efectiva.
  - **Analizabilidad:** facilidad con la que se puede evaluar el impacto de un determinado cambio sobre el sw, diagnosticar deficiencias o identificar partes a modificar.

- **Portabilidad:** capacidad del producto de ser transferido de forma efectiva y eficiente de un entorno hardware, software, operacional o de utilización a otro.
  - **Adaptabilidad:** capacidad del producto que le permite ser adaptado de forma efectiva a diferentes entornos.
  - **Capacidad para ser instalado:** facilidad con la que el producto se puede instalar y/o desinstalar de forma exitosa en un determinado entorno.
  - **Capacidad para ser reemplazado:** capacidad del producto para ser utilizado en lugar de otro producto software determinado con el mismo propósito y en el mismo entorno.

Número	Requerimiento	Descripción	Prioridad
RNF1	USABILIDAD	Debe ser fácil de usar. Con ayudas e interfaces intuitivas.	5
RNF2	SEGURIDAD	El ingreso al sistema estará restringido bajo contraseñas cifradas y usuarios definidos.	5
RNF3	PORTABILIDAD	El sistema debe brindar comodidad al usuario y a otras áreas que trabajan o necesitan del Área de personal. Por ejemplo El Sistema de Pago y Planillas no debe tener problemas en acceder al Sistema de Personal.	5
RNF4	MULTIPLATAFORMA	El sistema deberá funcionar en distintos tipos de sistemas operativos y plataformas de hardware.	3
RNF5	RENDIMIENTO	El sistema debe soportar el manejo de gran cantidad de información durante su proceso.	3
RNF6	DESEMPEÑO	El sistema no presentara problemas para su manejo e implementación.	1

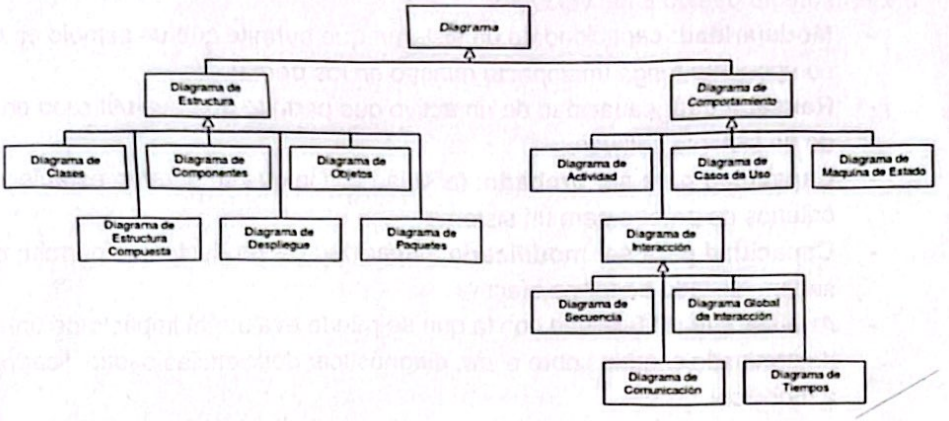
1.3 Modelamiento de software

¿Qué podemos modelar con UML? (categorías de modelamiento)

- **Diagramas estructurales:** muestran la estructura estática del sistema y sus partes en diferentes niveles de abstracción.
- **Diagramas de comportamiento:** muestran el comportamiento dinámico de los objetos en el sistema.

INGENIERIA DE SW

TIPOS DE DIAGRAMAS



- **Diagrama de Casos de Uso** PRECUNTABLE

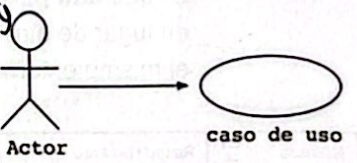
Nos permite representar la forma en que un Cliente (Actor) opera con el sistema en desarrollo y el tipo, orden en como los elementos interactúan (operaciones o casos de uso).

Elementos:

- **Actor:** Se refiere a la persona o sistema externo que realiza acciones en el sistema.
- **Caso de uso:** Se refiere a una acción que se puede realizar en el sistema.

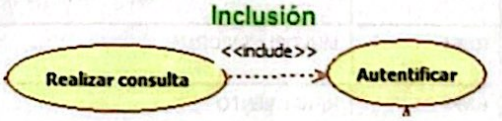
representa una tarea específica y acotada que involucra una interacción con un sistema

descripción de lo que un usuario espera de un sistema.



- **Relación directa:** forma de representar los comportamientos comunes, acciones directas que se representan mediante una flecha. ——————>
- **Relación de inclusión:** forma de interacción donde un caso de uso dado puede "depender" otro caso de uso. En simple, un caso de uso "requiere" de uno o más casos de uso. Se representa con una flecha desde el caso de uso "que ocupa" hacía los casos de uso "utilizados".

parte del otro, necesario para el otro, siempre.



- **Relación de extensión:** un caso de uso se puede "especializar" en otros casos de uso. Se representa con una flecha que comienza en el caso de uso "especializado" y termina en el caso de uso "general".

extension especificación

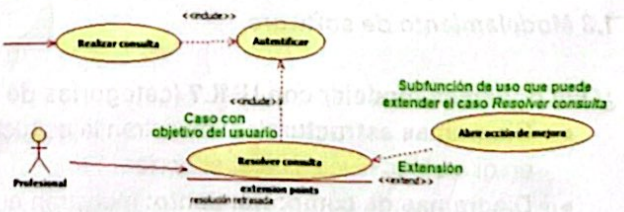
enviar mail

Actores

Emitir receta

[Verbo + sustantivo]

herencia.



- **Diagrama de estados**

Muestra la secuencia correcta de un caso de uso e indica los eventos para pasar de un estado a otro y las acciones que se generan con ello.

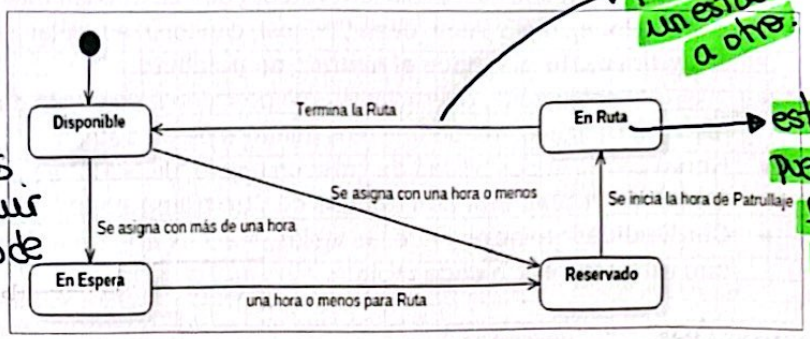
Elementos:

Recuadros: estado |círculo relleno: inicio | círculo con círculo exterior: final | líneas: eventos.

→ muestran los estados del sistema y eventos que causaron transiciones de uno a otro estado.

→ no muestran flujos de datos dentro del sistema, pero puede incluir info adicional del procesamiento de estos en cada estado.

Diagrama de Estados para un objeto "Patrullero"



pasa de un estado a otro.

estados, pueden tener funciones dentro

- **Diagrama de actividades**

Es considerado una extensión del diagrama de estados y se utiliza para modelar una secuencia de acciones y condiciones dentro de un proceso.

- **Branch:** se utiliza cuando existe la posibilidad de más de una transición (similar a un if/else). **Está representado por un rombo.**
- **Join y Fork**

Join ⇒ Ocurre al fusionar dos o más transiciones. Se representa con una **línea negra sólida, perpendicular a las líneas de transición.**

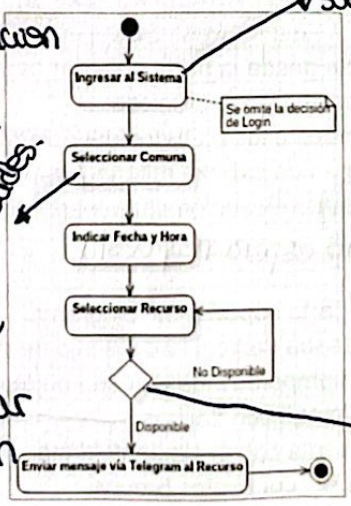
Fork ⇒ Representa una ramificación en dos o más transiciones obligadas. **Se representa igual que un Join.**

Elementos:

Los rombos representan condicionales, dos caminos, funciona como un if/else. | Línea negra sólida representa unión de actividades.

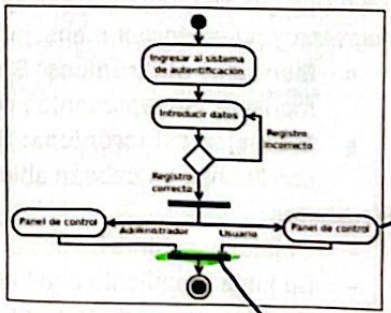
→ actividad: operacion de un usuario en el sistema, una operacion del sistema o accion realizada por alguien que contextualiza una operacion con el sistema.

→ consiste en un modelo de flujo de actividades, considerando que estas actividades se pueden ordenar secuencialmente y pueden producir resultados y/o outputs.



solo parte en uno

actividades



puede terminar en varios

toma de decisiones, se deben etiquetar.

unir varias actividades que apuntan a un mismo elemento dentro del diagrama

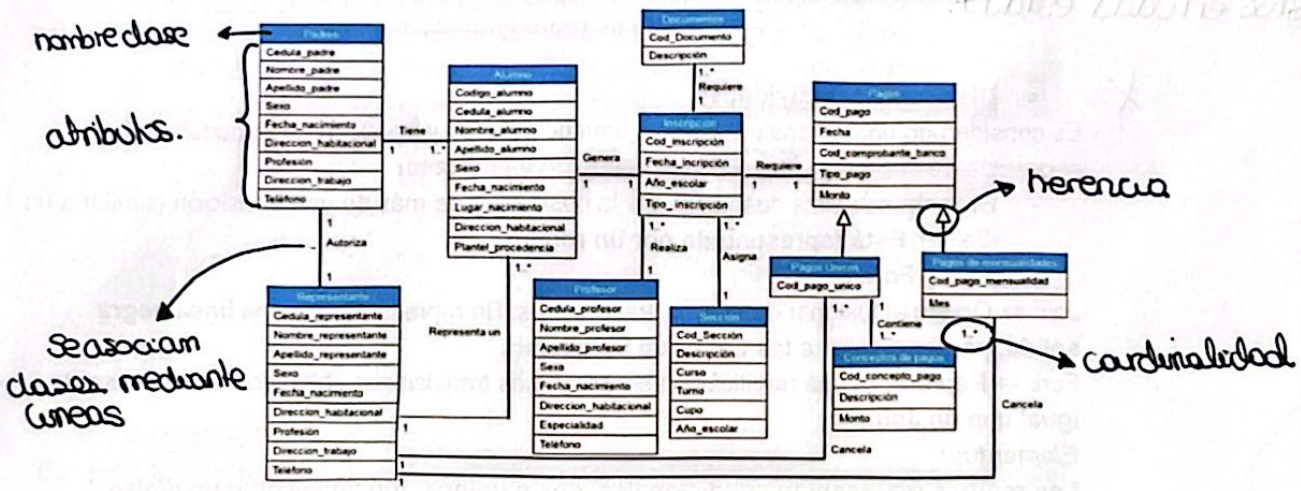
- **Diagrama de clases**

Diagrama estático de análisis y diseño que sirve para visualizar las relaciones entre las clases que involucran el sistema. Muestran los atributos, métodos y restricciones a que se ven sujetas las clases según la forma en que se conectan.

Elementos:

- **Atributos:** son características que corresponden a un objeto. **Representado en los recuadros debajo del nombre en cada cartilla, se puede poner el tipo de dato.**
- **Métodos:** son aquellas acciones (verbos) que se pueden realizar con y para un objeto. Por ejemplo: Abrir, cerrar, buscar, cancelar, acreditar, cargar, etc. **Se escribe entre líneas, lo que hace el método en palabras.**
- **Interfaz:** conjunto de operaciones que permiten a un objeto comportarse de cierta manera. Define los requerimientos mínimos de un sistema.
- **Herencia:** es la posibilidad de crear un objeto "derivado" a través de un objeto "base". **Se representa con flechas con un triángulo blanco hacia arriba.**
- **Cardinalidad:** relación entre las tablas, 1 a 1, N a 1, 1 a N, N a M. **Se representa con números entre cada tabla.**

cardinalidades.



- **Diagrama de secuencia** / diagrama poderoso.

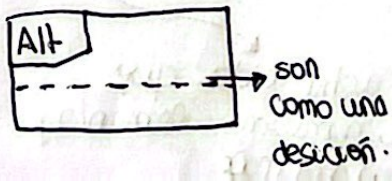
Es la "particularización" de cada caso de uso y se utiliza para modelar interacción entre objetos en un sistema ya que se basa en el "intercambio de mensajes" entre objetos.

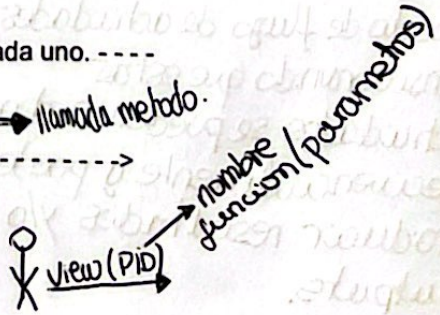
Los mensajes se dibujan cronológicamente desde la parte superior hasta la inferior del diagrama y puede incluir mensajes sincrónicos y asincrónicos:

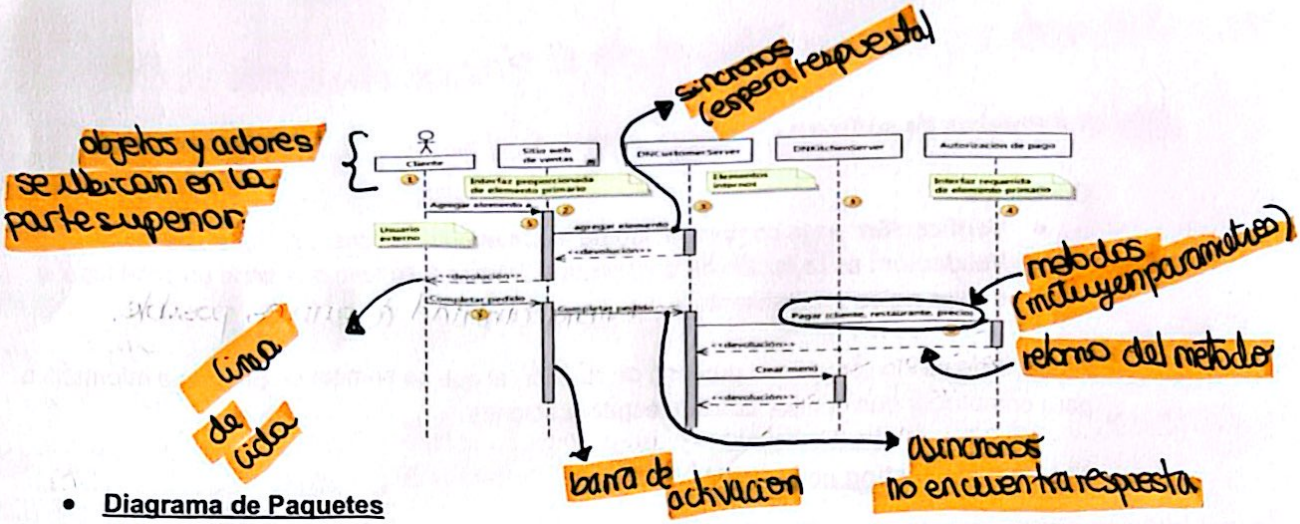
- **Mensajes sincrónicos:** Son llamadas a los métodos del objeto que recibe el mensaje. Se representan con flechas de cabeza rellena. / espera respuesta →
- **Mensajes asincrónicos:** Son hilos de ejecución en una secuencia. Se representan con flechas de cabeza abierta. / no espera respuesta. →

Elementos:

- Objetos y actores se ubican en la parte superior del diagrama.
- Se juntan mediante una línea punteada vertical hacia abajo de cada uno. - - - -
- Rectángulos verticales, indican el tiempo de vida de cada objeto.
- Métodos con parámetros se determinan con flechas. ——→ llamada método.
- Retorno de cada método se representa con la siguiente línea. - - - - - - - - - ->
- Se puede escribir entre interacciones, conceptos breves.





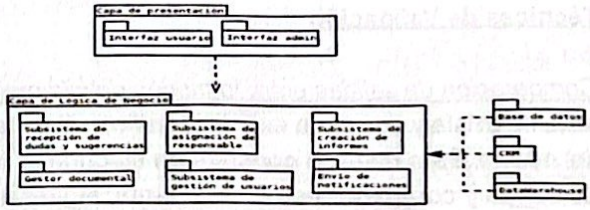


- **Diagrama de Paquetes**

Diagrama estático que representa las dependencias entre los paquetes que componen un modelo. Es decir, muestra cómo un sistema está dividido en agrupaciones lógicas (paquetes) y las dependencias entre esas agrupaciones.

Elementos:

- **Paquete:** espacio de nombres encapsulado dentro del cual todos los nombres deben ser únicos. **Visualmente se representa como una carpeta.**
- **Dependencia:** Indica que un elemento de un paquete requiere a otro de un paquete distinto. **Visualmente con una flecha que parte del elemento de origen y apunta hacia el elemento destino.**

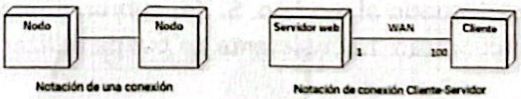


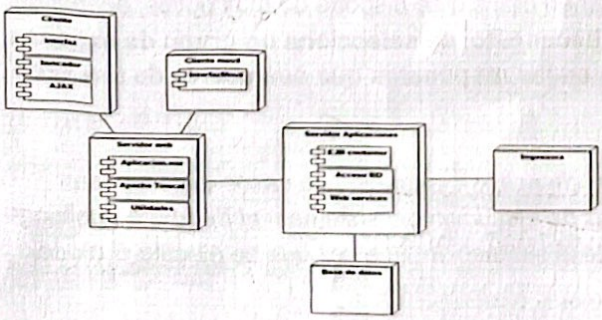
- **Diagrama de Despliegue**

Es un diagrama estático que representa la disposición física de los artefactos del sistema. Esto incluye tanto el Hardware como el Software.

Elementos:

- Los elementos usados por este tipo de diagrama son **nodos (representados como un prisma), componentes (representados como una caja rectangular con dos protuberancias del lado izquierdo) y asociaciones.**





1.4 Pruebas de software

Conceptos

- **Verificación:** es la comprobación de algún evento, proceso, material u otro.
- **Validación:** es la acción de dar fuerza o firmeza a aquello que tiene un peso legal o que es rígido y subsistente. → menor cantidad de errores posible.

Pruebas de validación: son el proceso de revisión al que se somete un programa informático para comprobar que cumple con sus especificaciones.

Técnicas de Testing como V&V

El Testing consiste en realizar una serie de pruebas controladas sobre el sistema o conjunto de programas.
Los **dos tipos** de prueba más comunes son:
Ascendentes: en estos se comienza por comprobar los módulos básicos y luego se pasa a probar las interrelaciones entre ellos hasta que el modelo ha sido probado como un único sistema.
Descendentes: las pruebas comienzan con el módulo principal y van bajando hasta los módulos básicos.

Técnicas de Validación

Comparación de salidas entre lo modelado y lo real: Se aplica en casos en los que el sistema exista y se pueda experimentar con él de forma que se obtengan datos de salida del mismo. **Este método consiste en ejecutar el modelo y obtener una serie de datos de salida y comparar éstos, mediante algún método estadístico, con resultados que se tengan del sistema.**

Método de Turing: En este test, a un experto o grupo de expertos, se le presentan resúmenes o informes de resultados de ejecución del sistema y del modelo, a los que se les ha dado el mismo formato. Estos informes se reparten aleatoriamente a los ingenieros y administradores del sistema, para ver si son capaces de discernir cuáles son los reales del sistema y cuales la imitación resultado de la simulación. **Si los expertos no son capaces de distinguir entre ambos, se puede concluir que no hay evidencias para considerar inadecuado al modelo. Si descubren diferencias las respuestas sobre lo que encuentran inconsistente se puede utilizar para realizar mejoras en el modelo.**

Método Delphi: Se utiliza cuando no se tienen datos o se dispone de muy pocos, acerca del sistema que se está considerando. **Para hacer esto, se selecciona un grupo de expertos los cuales deben llegar a un consenso en las respuestas que den acerca de una serie de preguntas que se les plantean.**

Comportamientos en casos extremos (test de estrés): situación de casos extremos son ideales para recoger datos de las medidas de ejecución del sistema real de forma que luego se puedan comparar con los resultados de la simulación, una vez que se ejecute el modelo bajo situaciones similares.

Testing:

→ 3 etapas:

1. **Development Testing:** se ejecuta durante la etapa de desarrollo por desarrolladores. Cuenta con 3 etapas:
   → **Unit testing:** unidades individuales del programa son testeadas. Su enfoque es en funcionalidades.
   - **Tipos:**
     - flujo normal y que muestre lo que el componente debe hacer.
     - condiciones anormales del programa.
   - **Ej:**
     - forzar inputs invalidos o computos muy grandes o muy pequeños.
     - repetir inputs o elegir inputs invalidos forzando al sistema a generar error.
   - **Estrategias para saber cual usar:**
     - **partition testing:** identificar inputs con caracteristicas en comun y se procesen de igual manera.
     - **Guideline-base testing:** se usan lineamientos para elegir los casos de prueba., es decir, experiencias previas de tipos de errores.
   - **PRUEBA CAJA NEGRA:** divide la entrada de un problema en clases de equivalencia, de las cuales se predicen las clases de valor de entrada que es probable que detecten errores. Particiones; mayor a 10, menor a 10, etc.
   - **PRUEBA CAJA BLANCA:** requieren funcionamiento interno del sistema, particiones son diferentes rangos donde se debe aplicar el mismo manejo de excepción.
     → **Component testing:** se enfoca en el testeo de interfaces ~~que~~ de componentes que dan acceso a funcionalidades especificas.
     → **system testing:** sistema se prueba como un todo, para ver bugs del sistema completo. Pruebas basadas en HU.
2. **Release testing:** un equipo de testeo, testea una version completa del sistema antes de ser lanzado a los usuarios. Se puede separar en perspectivas de testing basado en requerimientos, en escenarios o performance testing.
3. **User testing:** usuarios testean el sistema en su propio ambiente.

Existen 3 tipos:

- **Alpha Testing:** grupo selecto de usuarios trabajan con el equipo de desarrollo para versiones preliminares del sistema.
- **Beta Testing:** grupo mas grande de usuarios para permitirles experimentar y levantar problemas que puedan ir descubriendo.
- **Acceptance testing:** grupo de usuarios deciden si esta o no listo para ser lanzado.

Pruebas de software

- ***Pruebas unitarias:*** forma de comprobar el correcto funcionamiento de un módulo de código, con el fin de asegurar que cada módulo funcione por separado.

Prueba unitaria tenga calidad debe cumplir lo siguientes:

- Automatizable → Sin intervención humana (o mínima).
- Completas → Deben cubrir la mayor cantidad de código.
- Repetibles o Reciclables → Las pruebas deben poder ser ejecutadas varias veces.
- Independientes → La ejecución de una prueba no afecta a la ejecución de otra.
- Profesionales → Se deben documentar.

- ***Pruebas de integración***: que se realizan en el ámbito del desarrollo de software una vez que se han aprobado las pruebas unitarias. Se refieren a las pruebas hechas en conjunto de una sola vez, con el fin de evidenciar que los sistemas integrados funcionan.

- ***Pruebas Modulares:*** estas pruebas nos permiten determinar si un módulo del programa está listo y correctamente terminado.

- ***Pruebas de Aceptación:*** son pruebas formales que verifican si un sistema satisface los requisitos empresariales. Requieren que se esté ejecutando toda la aplicación durante las pruebas y se centran en replicar las conductas de los usuarios.

1.5 Definición de un Plan de Pruebas de SW

El plan de pruebas de software se elabora para atender los objetivos de calidad en un desarrollo de sistemas, encargándose de definir aspectos como los módulos o funcionalidades sujeto de verificación, tipos de pruebas, entornos, recursos asignados, etc.

- **PASO 1: Analizar los requerimientos de desarrollo de software:**

Lo primero es comprender los requerimientos de usuarios, teniendo eso se analiza toda la información de la documentación como la matriz de trazabilidad, especificaciones y diseño funcional, requisitos no funcionales, casos de uso, historias de usuario,etc.

- **PASO 2: Identificar las funcionalidades nuevas a probar**

A partir del paso anterior se debe identificar e incluir en el plan de pruebas de software en la lista de las funcionalidades si el sistema es nuevo. Si el desarrollo de software está integrado a un sistema existente es necesario revisar las funcionalidades que forman parte del desarrollo de software, en todas las capas de la arquitectura.

- **PASO 3: Identificar las funcionalidades de sistemas existentes que se deben probar**

Se debe identificar las funcionalidades existentes que están siendo impactadas por el desarrollo de alguna forma, considerando todos los componentes afectados en todas las capas de la arquitectura de software. Existen dos tipos de funcionalidades modificadas de cara al usuario y modificadas en sus componentes internos, en donde, está siendo modificada agregando más pantallas y modificando componentes internos que comparten con otras funcionalidades del sistema respectivamente.

- **PASO 4: Definir la estrategia de pruebas**

Consiste en seleccionar cuáles son los tipos de pruebas de software que se deben realizar.

- **PASO 5: Definir criterios de inicio, aceptación y suspensión de pruebas**

Criterios de aceptación o rechazo: para definir los criterios de aceptación o rechazo, es necesario definir el nivel de tolerancia a fallos de calidad. **Si la tolerancia a fallos es muy baja se puede definir como criterio de aceptación que el 100% de los casos de prueba estén sin incidencias.**

Criterios de suspensión: las condiciones van a depender de los acuerdos de nivel de servicio (SLA) internos de la organización y también de los acuerdos establecidos en cada proyecto individual.

Criterios de inicio o reanudación: definen las condiciones que se deben cumplir para dar inicio o reanudar las pruebas.

1.5 Arquitecturas de software

Arquitectura de sw.

FASES DE DISEÑAR SISTEMA

PASO 0:
- Analizar contexto. → Req fun y no fun

PASO 1:
- Estructuración → Arquitectura

PASO 2:
- Modelo de control

PASO 3:
- Descomposición modular.

La arquitectura de software se refiere a las partes que vamos a construir y la forma en la que las vamos a combinar y juntar para poder trabajar con ellas.

El estilo **arquitectónico monolítico** consiste en crear una aplicación autosuficiente que contenga absolutamente toda la funcionalidad necesaria para realizar la tarea para la cual fue diseñada.

Características:

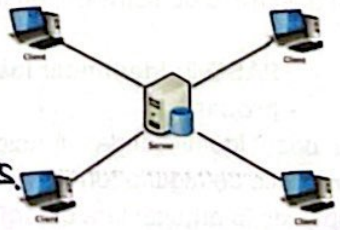
- **Componentes integrados:** los componentes de la aplicación trabajan de manera conjunta y comparten recursos y memoria, lo que significa que están altamente integrados.
- **Unidad cohesiva de código:** se presenta como una única y cohesiva unidad de código, donde todas las partes de la aplicación están combinadas en un solo programa o sistema.

- **Arquitectura Cliente - Servidor** → Servidores ofrecen servicios, en donde clientes requieren servicios, se comunican por redes.

Está compuesto por dos componentes, el proveedor (es un servidor que brinda una serie de servicios o recursos) y el consumidor (cliente que consume dichos servicios o recursos).

Existe un servidor y múltiples clientes que se conectan al servidor para recuperar todos los recursos necesarios para funcionar, en este sentido, el cliente solo es una capa para representar los datos y se detonan acciones para modificar el estado del servidor, mientras que el servidor es el que hace todo el trabajo pesado.

- como sabemos si esta disponible ono, un servicio, solo esperando su respuesta.



→ proactivo: igual a pasivo + definición de intereses.
→ pasivo: guarda y recupera lo que quiere, esperando.

- modelo de repositorio: → V: escalable, mantenible; D: punto unico de falla, dificil cambio de modelo de datos.
→ util cuando hay grandes cantidades de datos que deben ser compartidos por varias aplicaciones
→ los proovedores acceden a la BBD, los otros piden permiso a los proveedores para ejecutar una accion
→ permiten que apps accedan a una BBD mediante proovedores.

VENTAJAS	DESVENTAJAS
Centralización: el servidor es la fuente de verdad, por ende, los clientes siempre tendrán información actualizada.	**Punto único de fallo:** servidor se cae, clientes quedan inoperantes. • Performance deteriorada
Seguridad: servidor protegido por firewall o subredes que impiden que los atacantes puedan acceder a la base de datos o los recursos sin pasar por el servidor.	**Actualizaciones (clientes):** se debe asegurar que todos los clientes estén actualizados al momento que aparezca alguna actualización.
Fácil de instalar, cliente es una app simple sin dependencias.	**Concurrencia:** cantidad inesperada de usuarios concurrentes pueden generar problemas para que el servidor pueda atender todas las peticiones.
Separación de responsabilidades • Datos Distribuidos.	**Protocolos de bajo nivel:** comunicación entre el cliente y el servidor suelen ser de bajo nivel, como Sockets, HTTP, RPC, etc
Portabilidad: se desarrolla cada parte para correr en diferentes plataformas.	**Depuración:** dificil análisis de error, debido a que los clientes están distribuidos en diferentes máquinas, incluso, equipos a los cuales no se tiene acceso

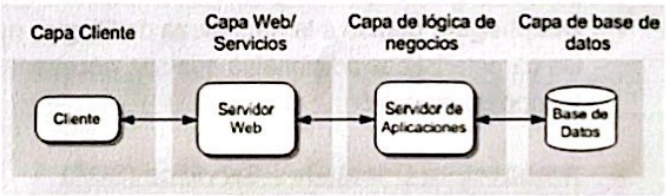
• escalable horizontalmente.

• no hay registro centralizado de los servicios.

- **Arquitectura de Capas**

La arquitectura en capas consta en dividir la aplicación en capas, con la intención de que cada capa tenga un rol muy definido: Capa de presentación (UI) Capa de reglas de negocio (servicios) Capa de acceso a datos (DAO).

→ conjunto de capas que ofrecen servicios específicos cada capa tiene una interfaz bien definida.



VENTAJAS	DESVENTAJAS
Separación de responsabilidades: cada capa tiene su responsabilidad. • desarrollo incremental.	**Tolerancia a los fallos:** si una capa falla, todas las capas superiores comienzan a fallar en cascada.
Seguridad: aislamiento de los servidores en subredes diferentes, ya que existe una separación de capas, haciendo ataques más difíciles.	**Rendimiento:** la comunicación por capas, genera un degrado significativo en el performance ya que se debe comunicar capa a capa.
Fácil de probar: posibilidad de probar funcionalidad capa por capa.	**Complejidad de despliegue:** se despliegan los componentes de abajo arriba, creando una dependencia de despliegue.

• Arquitectura objetos distribuidos / componentes se instancian cuando se requieren.
→ cada componente define los datos y metodos que pueden ser invocados
cada objeto provee y recibe servicios y se comunican a traves del sist ORB.
→ ventajas: flexible, facil de agregar objetos, configuracion dinamica, escalable.
→ desventaja: bajo rendimiento.

- **Arquitectura MicroKernel**

Arquitectura que permite crear aplicaciones extensibles, mediante la cual es posible agregar nueva funcionalidad mediante la adición de pequeños plugins que extienden la funcionalidad inicial del sistema.

- Usualmente se ocupa en la implementación de aplicaciones basadas en productos, es decir, aquellas que están empaquetadas y disponibles para su posterior descarga.
- Permite añadir características adicionales como plug-ins y es muy flexible y extensible.
- Se usa para aplicaciones que precisan datos de diferentes fuentes y para aplicaciones de flujo de trabajo.

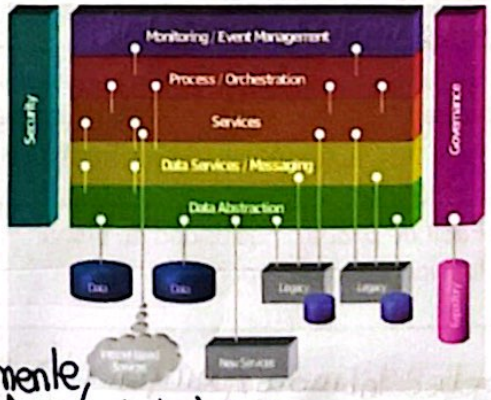
Ventajas:

- **Construcción modular:** el sistema de Plugins permite que diferentes equipos puedan trabajar en paralelo para desarrollar los diferentes Plugins.
- **Reutilización:** debido a que los Plugins pueden ser instalados en cualquier instancia del sistema Core, es posible reutilizar los módulos en varias instancias.
- **Testabilidad:** debido a que los Plugins y el sistema Core son desarrollados de forma por separado, es posible probarlos de forma aislada.
- **Despliegue:** debido a la naturaleza de Plugins es posible instalar fácilmente todas las características adicionales que sea necesarias, incluso, pueden ser agregar en tiempo de ejecución.

- **Arquitectura Orientada a Servicios (SOA)**

Es un método de desarrollo de software que utiliza componentes de software llamados servicios para crear aplicaciones (usualmente para empresas).

Cada uno de estos servicios brinda una capacidad empresarial y, además, se puede comunicar con el resto de los servicios mediante diferentes plataformas y lenguajes.



- servicios siempre operativos, para que el bus los use en base a los requerimientos del cliente.

→ tres componente: bus, servicios, cliente

→ clientes y servicios hablan con el bus esperando que pase algo.

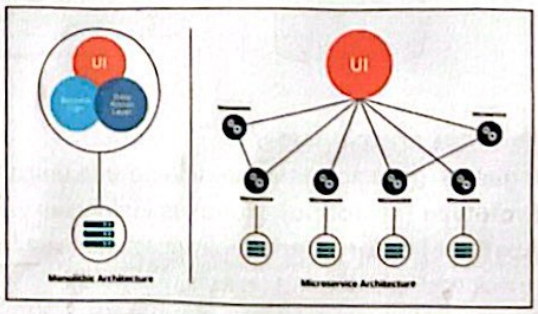
→ servicios administrados centralizadamente, conecta aplicaciones como servicios, flujo de datos controlados.

VENTAJAS	DESVENTAJAS
• **Reduce el acoplamiento:** ya que no existen referencias a la aplicación, si no a un servicio que puede estar en cualquier parte.	• **Rendimiento:** la arquitectura SOA no se caracteriza por el performance, pues la comunicación por la red y la distribución de los servicios agrega una latencia significativa en la comunicación.
• **Testabilidad:** fáciles de probar, pues es posible probar de forma individual cada servicio.	• **Volumen de peticiones:** SOA no se lleva con grandes volúmenes por petición, por lo que para procesar mucha información es recomendable utilizar otras tecnologías.
• **Reutilización:** Los servicios SOA son desarrollados para hacer una sola tarea, por lo que facilita que otras aplicaciones utilicen los servicios que existen.	• **Más puntos de falla:** debido a que SOA es una arquitectura distribuida, es necesario implementar estrategias de falla, pues se incrementa el número de puntos de falla.
• **Agilidad:** permite crear rápidamente nuevos servicios a partir de otros existentes.	• punto unico de falla: bus.
• **Escalabilidad:** los servicios SOA son muy fáciles de escalar, sobre todo porque permite agregar varias instancias de un servicio.	• alta dependencia de estandares
• **Interoperable:**permite que aplicaciones heterogéneas se comuniquen de forma homogénea mediante el uso de estándares abiertos.	

• reutilizacion de sistemas existentes

- **Arquitectura de microservicios**

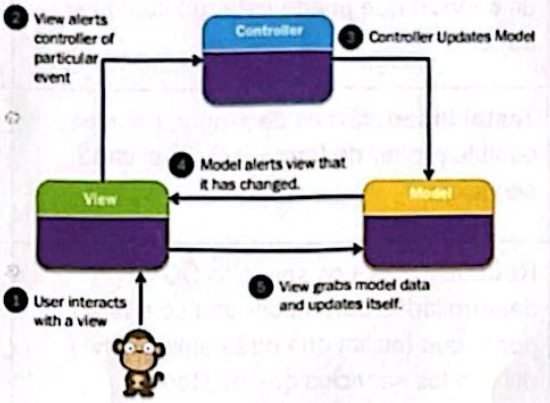
Lo que hace este patrón es escribir multitud de solicitudes que van a funcionar juntas.
Su principal ventaja es la posibilidad de escribir, mantener y desplegar cada microservicio de forma individual.
Se utiliza, sobre todo, para webapps con pequeños componentes, centros de datos no muy grandes y el desarrollo rápido de aplicaciones web.



• Arquitectura Cloud:
→ externalizacion de servicios computacionales.
→ IaaS = infraestructura, Paas = plataforma, Saas = aplicacion.
→ ventaja: se usa lo que se paga / FALSO SEGUN PROFE.
  • disponibilidad: replicas; escalabilidad: pido mas espacio; elasticidad:
→ desventaja: dependencia de este externo: seguridad/confidencialidad/[illegible]

- **Modelo Vista Controlador - MVC**

La arquitectura de software basada en el patrón Modelo-Vista-Controlador (MVC), divide el desarrollo del sistema en conceptos. Su objetivo es separar la lógica del negocio de la presentación. No pasa por alguno de manera obligatoria como el modelo de capas.



- **Patrón CQRS (Segregación de Responsabilidades de Comandos y Consultas)**

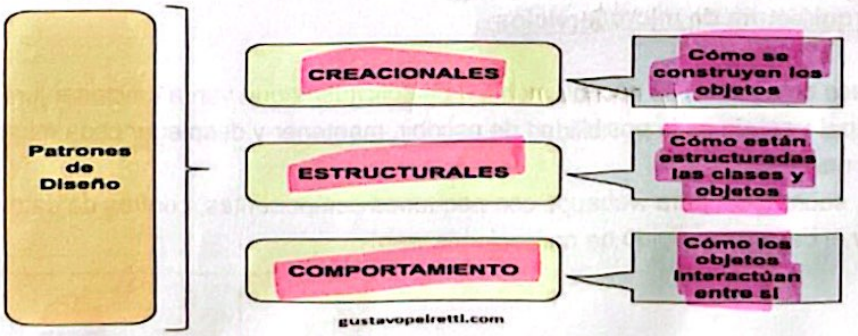
Separa las operaciones de lectura y actualización de un almacén de datos.
Se utiliza para maximizar el rendimiento, la escalabilidad y la seguridad de una aplicación.
Permite que el sistema evolucione mejor con el tiempo y no se vea dañado por los comandos de actualización.

1.6 Patrones de diseño de software

ING. SOFTWARE.

Los patrones de diseño son unas técnicas para resolver problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

TIPOS DE PATRONES



Patrones creacionales:

Ingleton: garantiza la existencia de una única instancia para una clase.
Prototype (prototipo): clona las instancias ya existentes.

- **Abstract Factory:** permite proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.
- **Builder:** ayuda a crear objetos complejos de manera sencilla, legible y escalable. Se utiliza en situaciones en las que debe construirse un objeto repetidas veces.
- **Factory Method:** nos ayuda a tener instancias de un objeto dado el tipo.

Public (Interface Abstract
CreateFish()
CreateAlga()
Public crear clases

Solo un objeto. pero igual que abstract Factory.

→ Uso de patrones:

Ejemplos:

- Patron Observador: decirles a varios objetos que el estado de un objeto ha cambiado.
- Patron Cascada: ordenar la interfaz a numero de objetos relaciones que muchas veces se han desarrollado incrementalmente.
- Patron Herador: provee una manera estandar de acceso a elementos en una coleccion, independiente de como se implemento esa coleccion.
- Patron Decorador: permite la posibilidad de extender la funcionalidad de una clase existente en tiempo de ejecucion.

permite empaquetar objetos, con el fin de modificar sus responsabilidades y comportamientos existentes

Patrones Estructurales:

Bridge (Puente): separa la abstracción de la implementación.

Decorator (Decorador): agrega funcionalidades a una clase de forma dinámica.

Facade (Fachada): nos provee una interfaz unificada y simple para acceder a un sistema más complejo.

permite a clases distintas, interactuar creando un objeto en comun en el que puedan comunicarse.

Adapter: cuando dos clases no se entienden, el adapter es mediador y adapta una clase para que la otra la entienda. → adaptar una clase; es decir, heredar sobre alguna clase.

Composite: ayuda a construir objetos complejos a partir de otros más simples.

Flyweight: se refiere a los objetos que queremos reutilizar para crear objetos más ligeros.

Proxy: es un elemento que se encarga de introducir un nivel de acceso a una clase. Ese nivel de acceso puede ser por seguridad o por complejidad.

Patrones de Comportamiento:

Observer (Observador): un objeto le pasa el estado interno a muchos objetos que están interesados. → separa la visualización del estado de un objeto del objeto propio y permite visualizaciones alternativas.

Chain of Responsibility: simplifica las interconexiones de objetos.

Command: separa acciones que pueden ser ejecutadas desde varios puntos diferentes de la aplicación a través de una interfaz sencilla.

Iterator: este se utiliza en relación a objetos que almacenan colecciones de otros objetos.

Mediator: define un objeto que media entre otros objetos.

Memento: se utiliza para restaurar el estado de un objeto a un estado anterior.

State: permite a un objeto alterar su comportamiento cuando su estado interno cambia.

Strategy: permite que un objeto tenga parte o todo su comportamiento definido en términos de otro objeto que sigue una interfaz particular.

Template Method: se centra en la reutilización del código para implementar pasos para resolver problemas.

Visitor: se utiliza para separar la lógica y las operaciones realizadas sobre una estructura compleja.

1.7 Siete etapas del proceso de desarrollo de software

Etapa 1: Análisis de Requisitos

Es la etapa en donde se obtienen los requisitos de un producto de software a través de indagación de necesidades del cliente.

Esto se realiza mediante entrevistas con clientes para saber cuales son las expectativas, detección y corrección de errores, retroalimentación, documentación, etc.

Los problemas en esta etapa es que los clientes piensan o creen que saben lo que el software debe hacer, requerimientos incompletos o contradictorios.

Etapa 2: Especificación

Es la etapa en donde se realiza la transformación del análisis en documentos que determinan el comportamiento esperado de las distintas funcionalidades y características inherentes del software. Se busca definir el qué y el cómo del software. Se formaliza a través del documento de SRS (Software Requirements Specification), este documento tiene su nivel de aprobación según lo completado que esté, para esto puede estar correcto, equivoco, completo, consistente, justificable, comprobable, modificable e identificable.

Etapa 3: Diseño y Arquitectura
Etapa en donde se definen las distintas interacciones, capacidades y trazado de solución. Se refiere a cómo se llevará a cabo la integración de infraestructura, el desarrollo de aplicaciones y cómo será el modelo y el uso de la base de datos.

Etapa 4: Programación
En esta etapa se realiza la codificación del Software en los distintos ambientes de un desarrollo. Se tiene como objetivos explícitos realizar la programación del sistema y lograr la interacción con los sistemas legados o interconectados y como objetivos implícitos optimizar la líneas y estructuras del código, lograr que se respeten los plazos, obtener la menor cantidad de bugs posibles y la realización de pruebas unitarias.

Etapa 5: Pruebas
En esta etapa, interconectada a distintas fases, comprobamos que el software hace las tareas que se definieron en su especificación y cumple con sus parámetros no funcionales. Se tiene pruebas del tipo unitarias, de integración, de sistema e interconectadas. Las pruebas debe realizarlas, por lo general, un equipo destinado a ello y no relacionadas con el equipo de desarrollo.

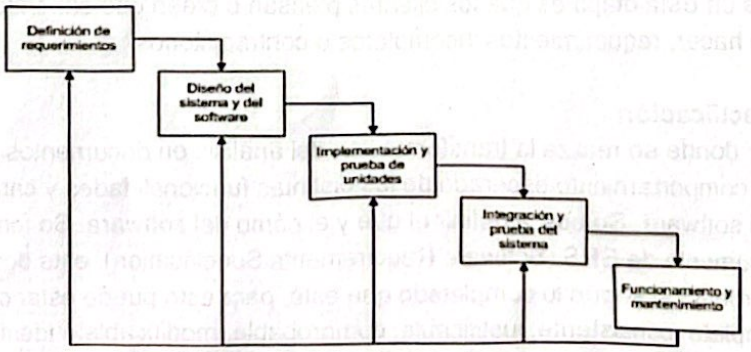
Etapa 6: Documentación
Etapa en donde se recopila todo lo realizado como parte del proceso de desarrollo de SW. Existen distintos tipos de documentos como los de requerimientos, gestión del proyecto, especificación (UML), diseño y arquitecturas, pruebas, manuales técnicos de usuario, etc.

Etapa 7: Mantenimiento
Etapa en donde el software ya está en producción y se ocupa de distintos mantenimientos. (mantenimientos de sistema, mantenimientos de mejoras del sistema, adicionales al sistema o eliminación de errores).

1.8 Modelos de desarrollo de software

- **Modelo de desarrollo en cascada:** /estructura lineal.

Ordena las etapas de modo tal que una etapa no comienza sino hasta que termina la anterior.
Considera las actividades fundamentales del proceso de especificación, desarrollo, validación y evolución, y los representa como fases separadas del proceso.



Etapas Modelo en Cascada

Etapa 1: Definición de Requerimientos

Se detallan y especifican los requerimientos como servicios, restricciones y metas a partir de las consultas con los usuarios.

Etapa 2: Diseño del Sistema y del Software

Proceso que divide los requerimientos en sistemas de hardware o software, establece una arquitectura del sistema e identifica y describe las abstracciones y relaciones fundamentales del sistema.

Etapa 3: Implementación y Prueba de unidades

Es llevar el diseño del software, a través de programación, a un conjunto o unidades de programas. La prueba de unidades implica verificar que cada una cumpla su especificación.

Etapa 4: Integración y Pruebas de Sistema

Las unidades individuales de programas se integran y se prueban como un sistema completo para asegurar que se cumplan los requerimientos del software.

Etapa 5: Funcionamiento y Mantenimiento

Fase más larga del ciclo de vida. El sistema se instala y se pone en funcionamiento práctico. El mantenimiento implica corregir errores no descubiertos en las etapas anteriores y mejorar la implementación de las unidades del sistema y resaltar los servicios del sistema una vez que se descubren nuevos requerimientos.

VENTAJAS	DESVENTAJAS
Se tiene todo bien organizado y no se mezclan las fases.	Proyectos en la vida real, raramente siguen una forma lineal. / inpracticable.
Perfecto para proyectos que son rígidos, y donde se especifiquen muy bien los requerimientos de la herramienta a utilizar.	Su mala implementación lleva al fracaso.

- Modelo de desarrollo en espiral:

Modelo en donde se representa como una espiral el proceso de de desarrollo de software, Cada ciclo en el espiral representa una fase del ~~proyecto~~ proceso del software, el ciclo mas interno se podría referir a la viabilidad del sistema, el sgte ciclo a la definicion de requerimientos y asi sucesivamente.

- Ademas cada ciclo de espiral, se divide en 4 sectores:

1. Definición de objetivos: se definen los objetivos específicos, se identifican las restricciones del proceso y el producto, se traza un plan detallado de gestión, se identifican los riesgos del proyecto.
2. Evaluación y Reducción de Riesgo: se lleva a cabo un analisis detallado para cada riesgo identificado y se definen los pasos para reducir dichos riesgos.
3. Desarrollo y validación: despues de la evaluacion de riesgo, se elige un modelo para el desarrollo del sistema.
4. Planificación: se revisa el proyecto y se toma la desicion de si se debe continuar con un ciclo posterior de la espiral. Si se continua, se desarrollan los planes para la sgte fase.

VENTAJAS	DESVENTAJAS
El análisis del riesgo se hace de forma explícita y clara.	Genera mucho tiempo en el desarrollo del sistema.
Integra el desarrollo con el mantenimiento.	El desarrollo puede tornarse muy costoso.
Incorpora objetivos de calidad.	Requiere experiencia en la identificación de riesgos.

- **Modelo de desarrollo Iterativo - Incremental** → se parte de un diseño y se modifica ese mismo.

Modelo de desarrollo donde la especificación, el diseño y la implementación del software se dividen en una serie de incrementos, los cuales se desarrollan por turnos.
En el **enfoque incremental, no existe una especificación completa del sistema hasta que el incremento final se especifica.**

Etapas Modelo de desarrollo Iterativo - Incremental

Etapa 1: Definir el esbozo de Requerimientos
Se detallan y se especifican los servicios, restricciones y metas del sistema.

Etapa 2: Asignar requerimientos a los incrementos
Etapa en donde se separan los requerimientos y se forman conjuntos interrelacionados se prioriza de acuerdo a algún parámetro y se agrupan y asocian a un número de incremento particular.

Etapa 3: Diseñar la Arquitectura del Sistema
En esta etapa se lleva el cúmulo de incrementos del software a un estado de necesidades y, finalmente, definir sus interacciones, tecnología para soportarlos y estándares de desarrollo.

Etapa 4, Cíclica: Desarrollar los incrementos
En esta etapa se lleva a cabo el diseño de software a través de la programación para cumplir con lo que se debe construir en cada iteración.

Etapa 5, Cíclica: Validar el incremento
En esta etapa se revisará, idealmente junto al usuario y/o cliente, que el incremento cumple sus expectativas.

Etapa 6, Cíclica: Integrar los incrementos
En esta etapa se reúne el incremento a los otros incrementos.

Etapa 7, Cíclica/Final: Validar el Sistema
En esta etapa se reúne el conjunto de incrementos al sistema completo que "interactúa" con él.

VENTAJAS	DESVENTAJAS
Reducción de tiempo del desarrollo inicial.	No es recomendable para casos de sistemas de tiempo real, de alto nivel de seguridad.
Facilidad de acomodar cambios al acotar el tamaño de los incrementos	Requiere de mucha planeación y necesita metas claras para conocer el estado del proyecto.

Tiene un impacto frente al cliente por la entrega temprana de partes operativas del software.	

- **Modelo de Desarrollo por Etapas** / incremental: desarrollar por partes.

Modelo del tipo "evolutivo" que considera que los requerimientos no son del todo conocidos al comenzar el proyecto. Esto implica que se van desarrollando las especificaciones a medida que se está trabajando en una versión y existen actividades recurrentes que requieren interacción directa con el cliente para el feedback.

Cuenta con un desarrollo exploratorio, en donde se trabaja con el cliente para explorar sus requerimientos y entregar un sistema final. El desarrollo empieza con las partes del sistema que se comprenden mejor y el sistema evoluciona agregando nuevos atributos propuestos por el cliente.

VENTAJAS	DESVENTAJAS
Constante intercambio de información con el cliente.	No facilita la integración de aplicaciones que han sido desarrolladas como sistemas independientes
Entregas periódicas del Sistema.	Algunas deficiencias del software se hacen inmodificables junto a la evolución
Priorizan las necesidades del negocio de acuerdo a cómo las enfrenta el cliente.	

1.9 Metodologías de desarrollo de software

Programación Extrema (XP): es una metodología de desarrollo de software que está destinada a mejorar la calidad del software y capacidad de respuesta a los cambios en los requisitos del cliente.

Scrum: método ágil basado en el Modelo Iterativo/Incremental que se enfoca en la gestión de procesos de desarrollo de software y que propone un conjunto de prácticas y roles para definir el proceso de desarrollo que se ejecutará durante un proyecto.

EJEMPLO PRÁCTICO DE LA METODOLOGÍA SCRUM

Paso 1: Un cliente se pone en contacto con una empresa que fabrica Robots y realiza el pedido.

Paso 2: El cliente se reúne con el Product Owner quien toma nota de lo que desea (tiene pensado) el cliente.

Paso 3: El producto se divide en historias (partes o entregables) que definen la Pila del Producto.

Paso 4: El Product Owner se reúne con el ScrumMaster para entregar la Pila de Producto.

Paso 5: El equipo se reúne para estimar el costo (esfuerzo) de capa historia de la pila.

Paso 6: El cliente, una vez aprobado el presupuesto, reordena la pila de acuerdo a sus prioridades para el desarrollo (de mayor a menor o menor a mayor urgencia, pero con claridad).

Paso 7: El equipo comienza su trabajo desglosando la primera historia en tareas menores para crear el Sprint y su pila respectiva.

Paso 8: El Product Owner, después de consultar al cliente, prioriza las tareas antes de comenzar el Sprint.

Paso 9: El equipo comienza el Sprint tomando las tareas priorizadas. Una vez terminada una tarea, se toma la siguiente. Así mismo, a diario se generan reuniones para ver el avance del equipo.

Paso 10: Al finalizar el sprint, el Product Owner se reúne con el cliente para mostrar el avance. Con ello, el cliente tiene la primera entrega y puede, si lo desea, reordenar la Pila para que comience otro Sprint.

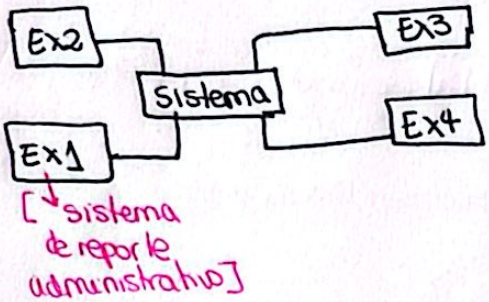
Paso 11: El equipo se reúne para hacer la revisión del Sprint y su retrospectiva. Se recomienda un ambiente distendido (fuera de la oficina) para esta reunión.

modelo de contexto:

→ da pie para un modelo de arquitectura

→ recuadros de afuera son externos, servicios, API, BBD, etc.

→ sistema a desarrollar al medio y sistemas externos afuera.



- muestra interacciones con otros sistemas ya existentes.
- sistemas externos puede ser una API, BBD, servidores web, etc.
- muestra tipos de relaciones con sistemas del ambiente, I/O datos, automatizaciones, interacciones con usuarios, etc.

¿Como realizar un diagrama de contexto?

- Hacer una lista de todos los sistemas externos que se conectaran con el sistema que vamos a desarrollar.
- el sistema se ubica en el centro y los externos alrededor.
- Los sistemas se modelan con rectangulos, se conectan con lineas, son cardinalidad o tipos de relacion.

→ sistema externo: aquel ya desarrollado y solo podria necesitar modificaciones menores para conectarse con nuestro sistema.

→ Los sistemas externos son sistemas fisicos como sensores, ademas de sw, servicios cloud.

→ tambien se considera un sub-sistema de un sistema.

• Modelo de control: como controlamos el sistema. 3°
→ control del flujo de procesamiento entre los componentes.

1. Centralizado: un componente controla la ejecución del sistema.
→ modelo call-return: cadena de procesos, llega uno lo atiendo, luego otro / invoco a alguien termina y vuelve sgte
→ modelo administrador: malla de procesos, el admin administra mas de una cadena a la vez.

2. Descentralizado basado en eventos.
→ procesa eventos generados asincronicamente / eventos = alertas generados por procesos.
→ transmision multiple: modelo publicador / suscriptor
los programas actuan cuando se produzca algun evento, nadie controla a quien le toca. Procesos esperan a que se produzca un evento, por ende, si el proceso esta suscrito lo ejecuta.
→ manejo de interrupciones: eventos de alta prioridad.
interrupcion: evento de alta prioridad, en donde el suscriptor debe tener todo para priorizar eso rapido. Se da en sistemas en tiempo real, como un piloto automatico; procesos de extraccion de cobre.

• Descomposicion modular 4°
→ es describir los componentes, es decir, el detalle de lo necesario y como lo haremos.

1. Modelo de objetos:
→ conjunto de clases debilmente acopladas.
→ crear objetos a partir de clases.

2. Modelo flujo de datos:
→ descomposicion en procesos funcionales
→ flujo predeterminado.
→ ej: compra en un ecommerce → compra, pago, avisar compra, confirmar, despacho, etc.

• Arquitectura de dominio especifico:
→ Arquitecturas genericas: sistema ya existente, generalizacion de sistemas reales, resuelven requerimientos especificos.
→ Arquitectura de referencia: conjunto de recomendaciones para crear un sistema, son mas de estudio.

• Patrones de Arquitectura:
→ nos dice que hacer segun la arquitectura elegida, componentes a usar, recomendaciones aprobadas.
→ Para definirlo se analiza el contexto, los componentes (procesos a construir), la responsabilidad de los componentes y como se relacionan entre ellos.
→ descripcion de un patron: nombre del patron, contexto (situacion que origina la necesidad), los requerimientos (necesidad a solucionar) y la solucion (estructura, comportamiento), etc.

→ Clasificación:

- Patrones simples: Arquitectura que hacer, patron como hacerlo.

1. Patron Capas: estructura multi-capa descompone c/u en tareas con diferentes niveles.

- se tiene una capa base en donde se completa la transaccion y esta invoca a componentes que hace algo mas generico y asi sucesivamente.
- se empieza con la interfaz de usuario y cada capa expone una interfaz con los servicios que ofrece para poder utilizarla.
- capa base nivel de abstraccion mas bajo.
- Para implementarlo en la arquitectura, definimos la cantidad de capas, asignamos responsabilidades, especificamos servicios de cada capa (UI, validacion, seguridad, etc), luego ~~que servicios provee~~ la estructura de cada capa, su metodo de comunicación y el esquema de manejo de errores.
  ↳ (HTTP, RES, etc).

- ventaja: componentes estandarizados, cambios afectan a nivel local
- desventaja: baja performan, afecta en cascada los cambios, complejo de definir.

2. Patron Tubos y Filtros: procesamiento de flujo de datos.

- Se tienen 2 elementos:
  → filtros: procesos que transforman datos de entrada en datos de salida, son independientes (no comparten su estado y desconocen a los otros filtros).
  → Tubos: medio de comunicacion entre 2 filtros.
- Para implementarlo se divide el sistema en una secuencia de procesos ordenados e independientes, se define un formato de los datos transmitido por los tubos(si es igual el formato mejor rendimiento, si son distintos formatos mas flexibilidad) luego se especifica el procesamiento de cada filtro, se construyen y se define el esquema para el manejo de errores.
- ventaja: arquitectura flexible, filtros reutilizables, procesamiento paralelo.
- desventaja: informacion no compartido (filtros independientes), conversion de datos, errores afectan al flujo de procesamiento.

3. Repositorio:

- Grandes cantidades de datos en diferentes plataformas que deben ser compartidos. Se comporta como una capa intermedia entre los apps y los datos.
- Separa la lógica del negocio de las fuentes de datos → datos independientes.
- ventajas: arquitectura flexible, independencia de datos, desarrollo independiente, reutilizable.
- desventajas: sobrecarga de procesos, duplicacion de codigo, agrega capa adicional.

Patrones para sistemas interactivos

1. MVC: modelo vista controlador:

- sistema se divide en 3: modelo (contiene datos y funcionalidades esenciales), vista (comunicación con el usuario), controlador (controla los cambios que se le efectuan al modelo).

→ interfaz de usuario: (V + C) ; logica del negocio (C + M).

→ **Contexto**: sistemas interactivos con interfaz flexible.

→ **problema**: interfaz con diferentes representaciones posee una interfaz cambiante, facil de modificar y si se quiere agregar una nueva funcionalidad, implica modificar la interfaz.

→ **solucion**: la solucion consta de 3 componentes.
La comunicacion (V) envia datos y despliega al usuario los datos en respuesta al sistema.
administracion (C): define el comportamiento del sistema, recibe eventos de la interfaz y solicita los servicios respectivos al modelo.
procesamiento (M): provee funcionalidades por la vista, encapsula al modelo de datos

→ **implementación**: separar la funcionalidad del sistema de la interaccion del usuario, se tiene que diseñar: MVC y las interacciones entre ellas.

→ **Ventajas**.
- soporta multiples vistas, flexible, mantenible.

→ **desventajas**:
- complejidad, vista no hay acceso a datos

2. PAC: Presentación Abstracción Control:

(P)(A)(C) → [presentacion: interfaz; Abstraccion: funcionalidad; Contpl: Comunicacion con agentes]

→ patron que tiene una estructura jerarquica con agentes que cooperan entre ellos para dar una solucion, los agentes proveen la funcionalidad de la aplicación.

→ **contexto**: sistema interactivo desarrollados utilizando agentes.

→ **problema**: tener un sistema interactivo mediante agente funcionando en el sistema de manera integrada. Los agentes deberan tener interfaces propias de comunicacion, estados y datos privados.

→ **solución**: definir la estructura jerarquica del sistema, alto nivel (funcionalidad central), intermedio (relaciona agentes de bajo nivel), bajo nivel (manejo interfaz de usuario). Cada agente es responsable de parte del funcionamiento en donde la presentacion, es el aspecto visible, la abstraccion es el modelo de datos y las operaciones sobre ellos y control conexion entre agentes.

→ **implementacion**: se definen las funcionalidades del sistema, sus estructuras jerarquicas a utilizar y definir cada agente, su funcionalidad, interfaz etc.

→ **Ventajas**
- soporta multitarea, asignacion de responsabilidades especificas

→ **desventajas**
- baja eficiencia, complejo control