Curso de Ingeniería de Software

Unidad 6
Diseño de software

Guadalupe Ibargüengoitia
Hanna Oktaba

Objetivos

- Proponer y documentar el diseño arquitectónico y detallado del software, que corresponde a la especificación de requerimientos de los casos de uso dentro del alcance de la iteración, empleando diagramas de UML.

Entradas a esta unidad

- **Condiciones**
  - Requerimientos de software entendidos
- **Productos de trabajo**
  - *Especificación de requerimientos de software* que incluye:
    - Diagrama general de casos de uso de la iteración
    - Detalle de los casos de uso
    - Prototipos de interfaz de usuario
    - Casos de prueba para los casos de uso
    - Diagramas de navegación

DEFINICIONES DE CONCEPTOS

¿Qué es el diseño de software?

- Diseño de cualquier producto consiste en crear un modelo o representación de lo que se construirá más adelante.

- Diseño de software es el conjunto de actividades creativas mediante las cuales los requerimientos se traducen en una representación del software.

http://losretosdelasociedad.blogspot.com/2017/10/que-es-diseno.html

¿Cuáles son los objetivos del diseño de software?

- Identificar y caracterizar las partes o componentes principales del software

- Definir su interacción e integración en el producto, para llegar al nivel de detalle que permita su mapeo al código en algún lenguaje de programación.

Niveles de abstracción del diseño

- ***Diseño arquitectónico -*** describe cómo se descompone y organiza el software en componentes abstractos partiendo de la especificación de requerimientos.
- ***Diseño detallado -*** describe el comportamiento específico de esos componentes tomando en cuenta el ambiente en el cual se codificará.(SWEBOK 2014)

Diferencia entre Arquitectura de Software y Diseño de Software

- Andrés Mejia Jeferson Vergara

https://youtu.be/8hULWNSPo1w

¿Qué hemos aprendido?

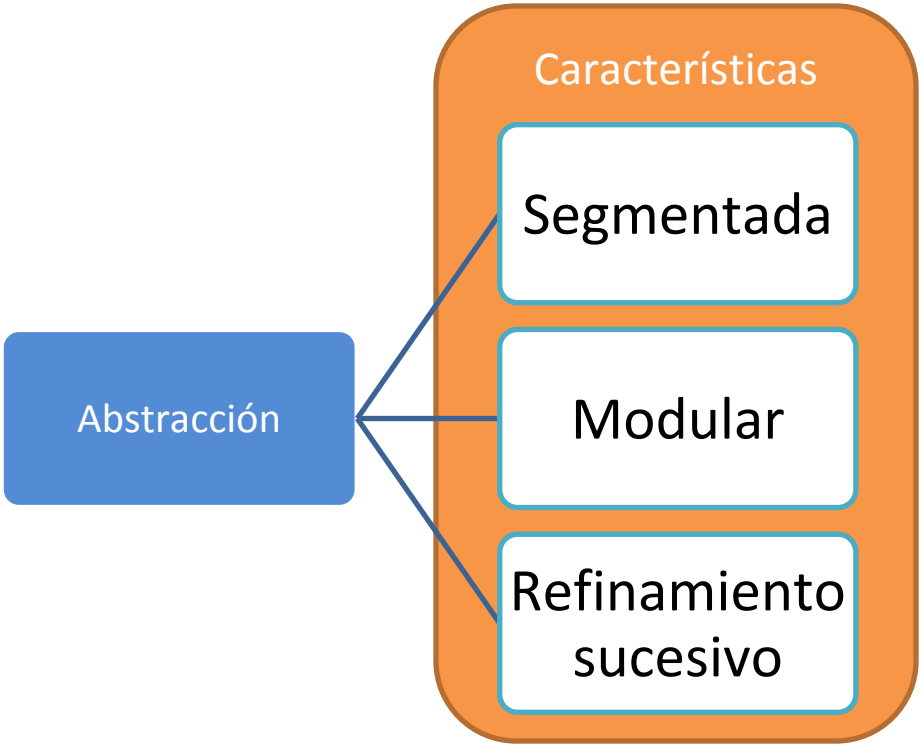
- ¿Qué es y para qué se hace el diseño de software?

- ¿En qué se distingue el diseño arquitectónico del diseño detallado?

Principios para el Diseño de Software (SWEBOK, 2014)

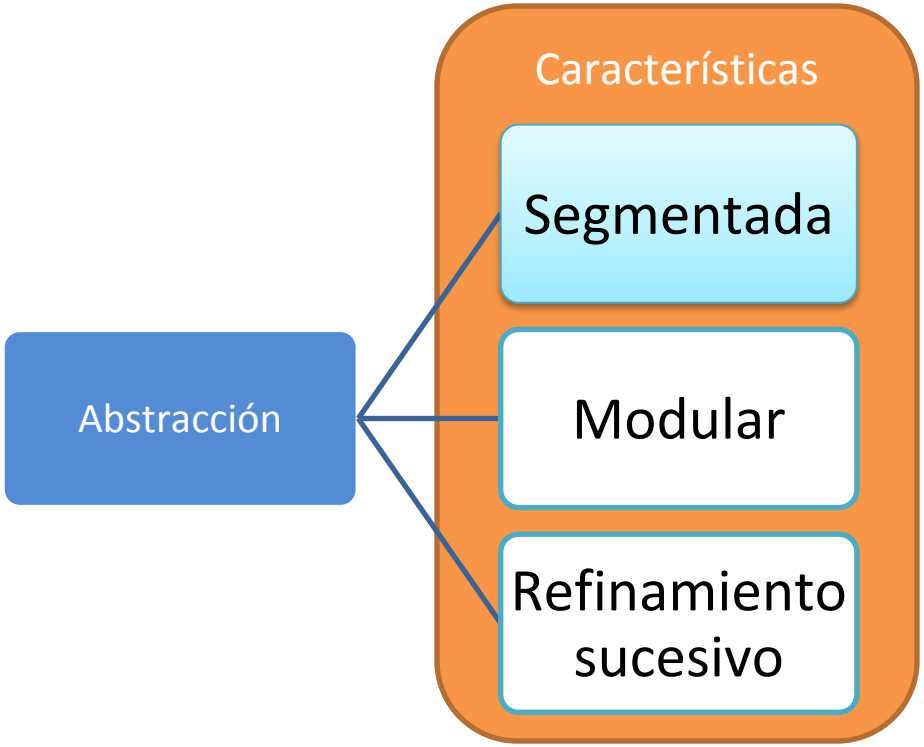
- **Abstracción –** es una vista de un objeto que se centra en la información relevante para un propósito particular e ignora el resto de la información. Para el diseño de software inicialmente se abstraen las propiedades más generales que deberán tener los componentes del software a construir.

...Principios básicos



Se refiere a identificar y modelar de forma estructurada propiedades esenciales de un conjunto de objetos omitiendo detalles no esenciales, según sea el caso.

...Principios básicos

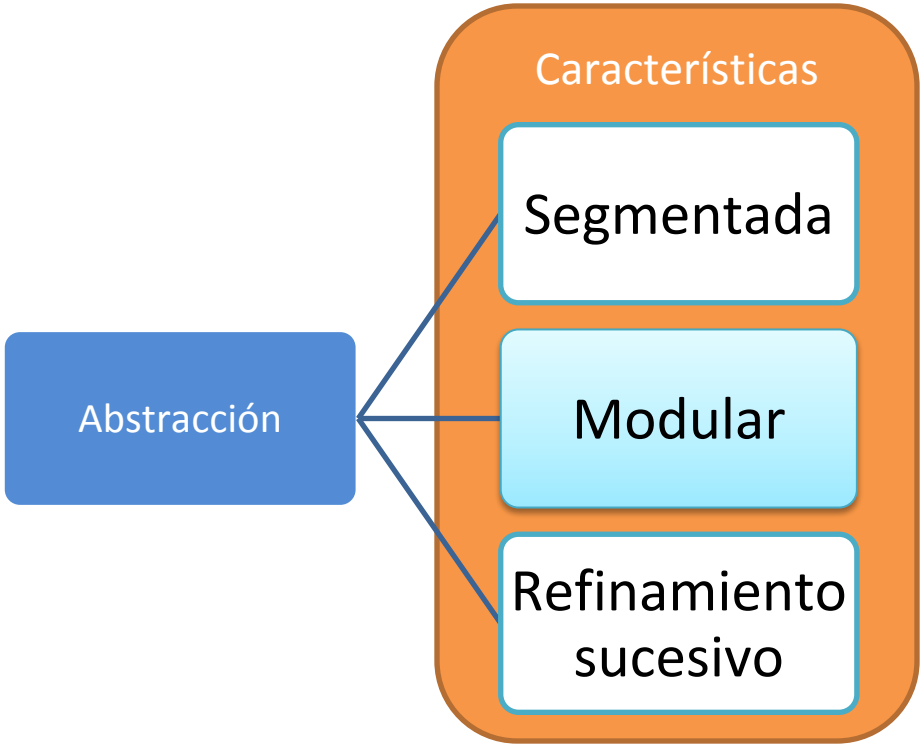


Se refiere a dividir en múltiples elementos que permitan manejar fácilmente un sistema en su conjunto.

Segmentación

- **Segmentación de conceptos** - una preocupación de diseño es la segmentación de elementos relevantes para uno o más de sus involucrados. Cada vista de arquitectura refleja uno o más intereses. Segmentar los conceptos por vistas permite a los interesados cen en una cosa a la vez y de esta manera manejar la complejidad.

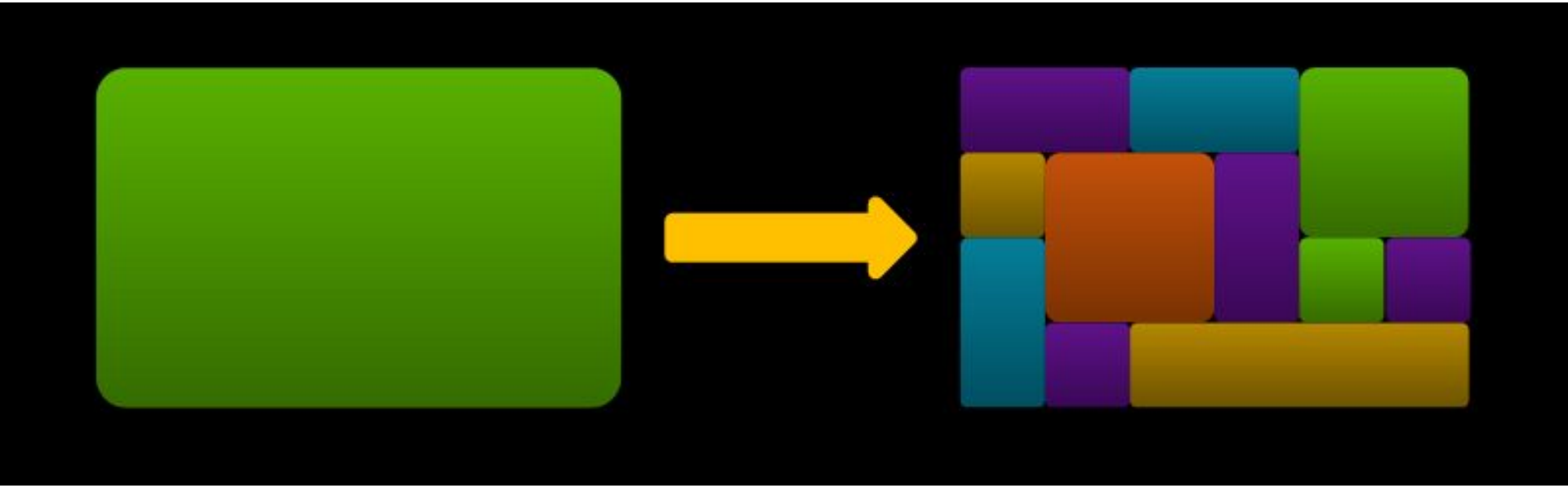
...Principios básicos



Se refiere a que cada elemento cumpla funcionalidades que lo hagan un componente individual.

Modularización

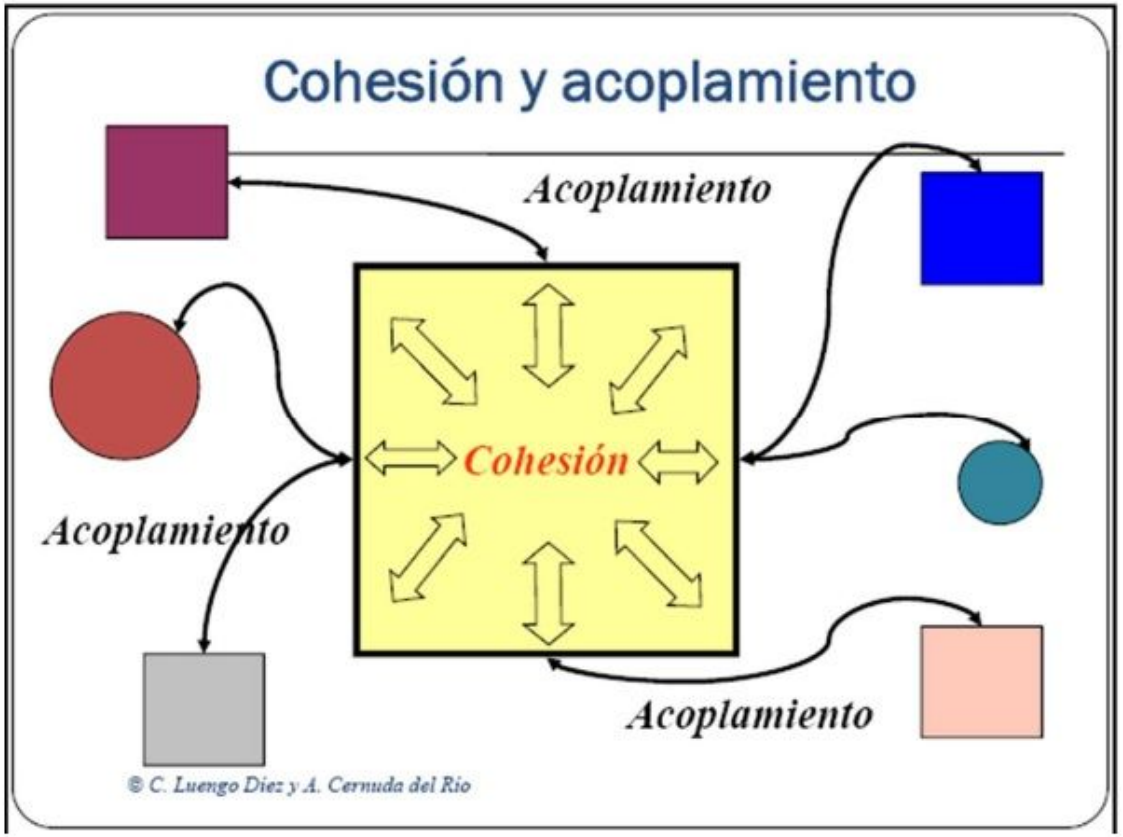
- **Descomposición y modularización -** descomponer y modularizar el software significa dividirlo en una serie de pequeños componentes, que tienen bien definidos interfaces y que describen las interacciones de los componentes. Por lo general, el objetivo es colocar diferentes funcionalidades y responsabilidades en diferentes componentes.



Acoplamiento y Cohesión de los componentes/módulos

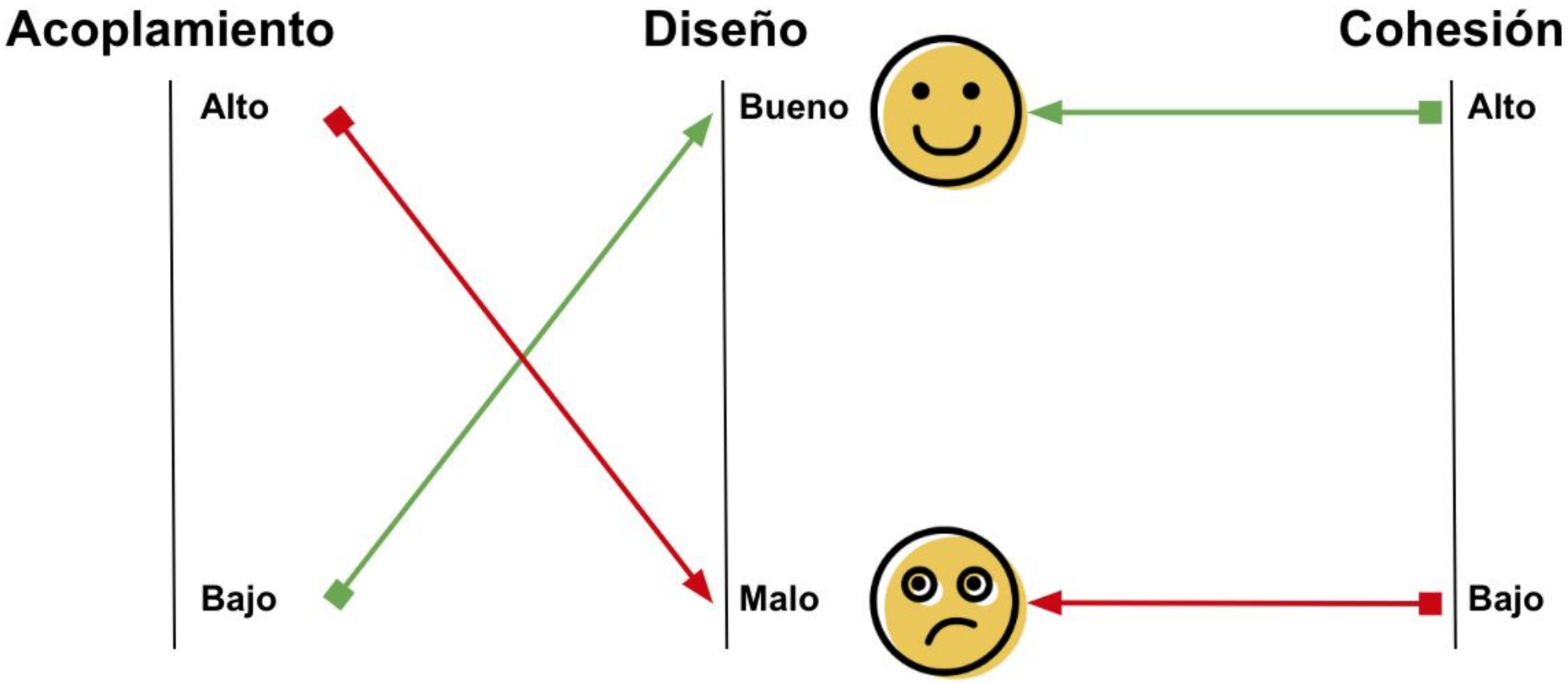
- **Acoplamiento y cohesión** - el acoplamiento está definido como "una medida de la interdependencia entre módulos en un programa de computadora", mientras que cohesión se define como "una medida de la fuerza de asociación de los elementos dentro de un módulo ".
- Un buen diseño busca la cohesión de módulos **alta** y el acoplamiento **débil**.

Cohesión y Acoplamiento



Material Preparado por MARTA SILVIA TABARES B. UdeM

Flujo de Diseño UP


Acoplamiento
Diseño
Cohesión
Alto
Bajo
Bueno
Malo
Alto
Bajo

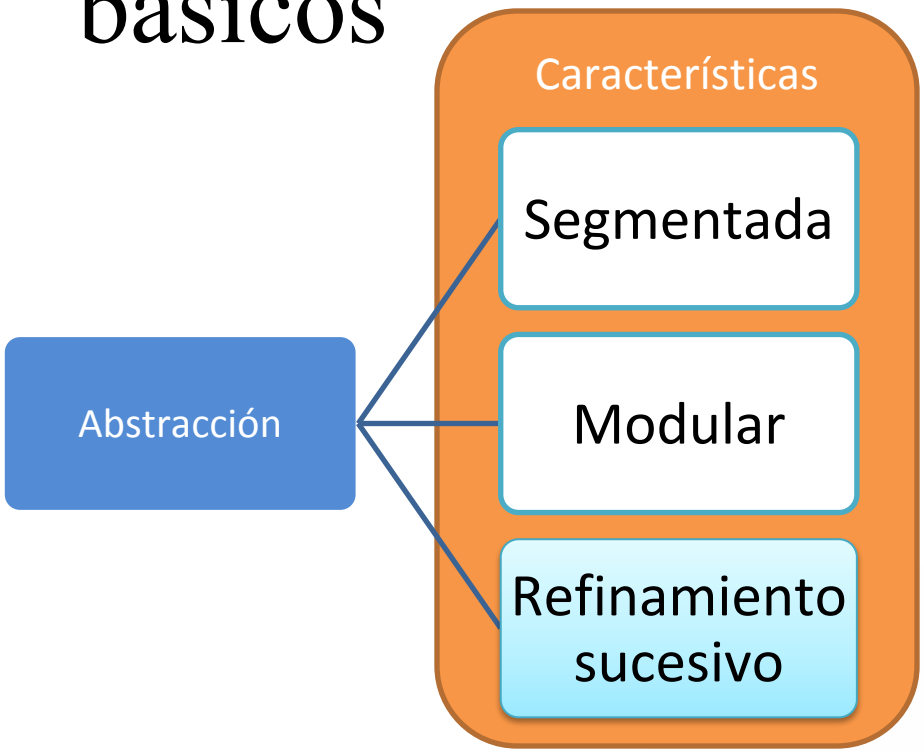
Encapsulación y ocultamiento de la información de un componente/módulo

- **Encapsulación y ocultamiento de la información** - propone agrupar y empaquetar los detalles internos de una abstracción y hacer esos detalles inaccesibles a entidades externas.

Separación de la interfaz y la implementación de un componente/módulo

- **Separación de la interfaz y la implementación** - significa definir un componente especificando una interfaz pública la cual está separada de los detalles de cómo el componente se implementa.

...Principios básicos



Se refiere a extender deductivamente el modelo conceptual de requerimientos en una serie de incrementos precisando cada vez, especificaciones que aumentan el nivel de detalle.

Mantenibilidad

Módulo Consulta

Refinamiento sucesivo del diseño

- **Refinamiento sucesivo -** extender deductivamente el modelo conceptual de requerimientos en una serie de incrementos precisando cada vez más, especificaciones que aumentan el nivel de detalle.

Otros principios

- **Diseñar para el cambio** - significa que el diseño debe ser flexible para permitir cambios con relativa facilidad.
- **Diseñar para facilitar el uso del software** - Considerar algunos escenarios del uso del software y su interfaz puede ayudar en el diseño de los componentes apropiados.

Otros principios

- **Diseñar para facilitar la prueba** - Los componentes del sistema deben estar diseñados como unidades que se pueden probar sin depender de la implementación de otros componentes.
- **Diseñar para la reutilización** –Consiste en definir partes genéricas que puedan volver a usarse. Para aplicar este principio se deben identificar los componentes comunes que se podrán reutilizar. El reuso incluye no solo el nivel del diseño, sino de código, casos de prueba, modelos o diagramas.

Otros principios

- **Diseñar para facilitar la comprensión humana**
  – Usar los estándares acordados por las comunidades de desarrolladores para el nombramiento y estructura de los diferentes elementos de la arquitectura para facilitar que lo comprendan otros desarrolladores. Esto facilitará la labor de mantenimiento.

Cualidades del diseño de software

- Al diseñar se deben considerar las cualidades:
  - De la arquitectura: integridad, corrección, facilidad de construcción y completitud.
  - De uso: seguridad, eficiencia, funcionalidad y usabilidad.
  - De evolución: facilidad de modificación, de prueba, reusabilidad y portabilidad.

El diseño es una actividad creativa por lo que no existe el mejor diseño.

¿Qué hemos aprendido?

- ¿En qué nos ayudan los principios de diseño de software?

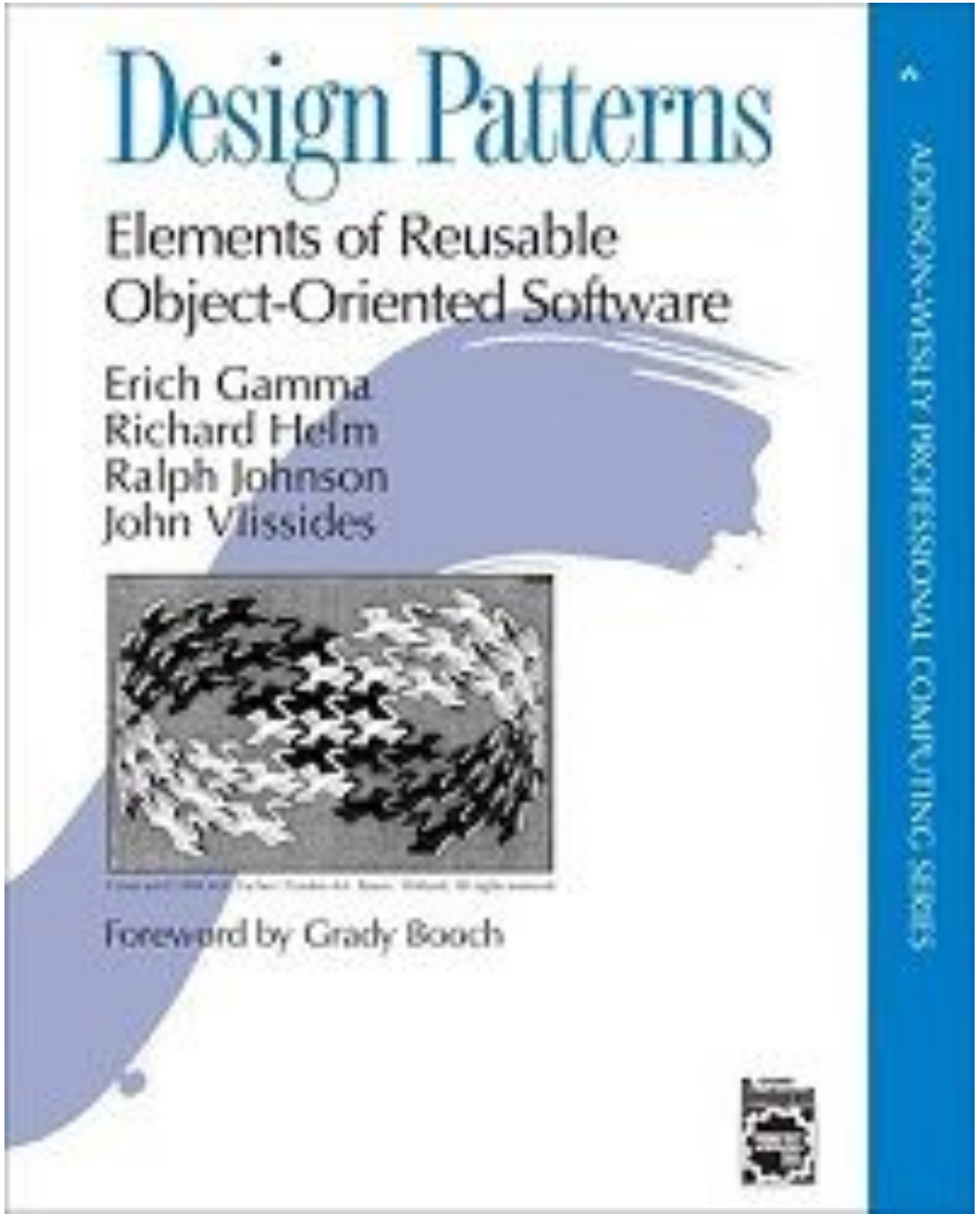
- ¿Cuál de los principios has aplicado anteriormente?

Actividades del diseño

1. Diseñar la arquitectura del software
2. Seleccionar el ambiente de desarrollo
3. Diseñar los componentes
4. Diseñar la base de datos

Diseñar la arquitectura de software

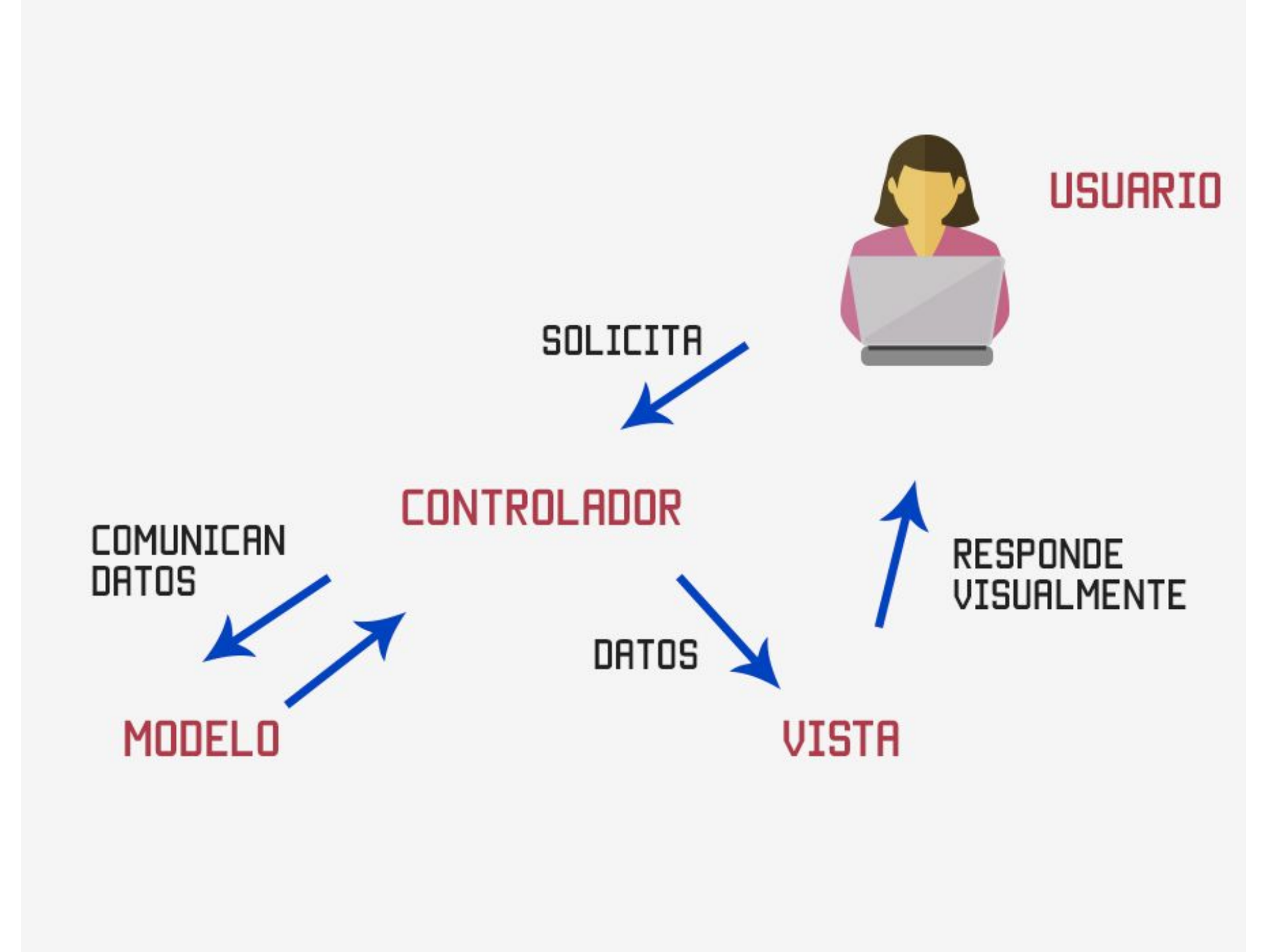
- Seleccionar el **tipo de arquitectura** del software según el tipo de la aplicación a desarrollar.
- Identificar los **componentes** básicos que conformarán la **arquitectura** del software.
- Documentar la **vista estática** de la arquitectura con un **diagrama de paquetes de UML**.
- Usar estándares de nombramiento.


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

1995

Arquitectura *Modelo Vista Controlador (MVC)*

- Para aplicaciones web tradicionales hay un tipo de arquitectura recomendado conocido como patrón arquitectónico *Modelo Vista Controlador (MVC)*.
- MVC tiene varias cualidades como son modularidad, sencillez, facilidad de construcción y adaptabilidad al cambio, separa los datos de una aplicación, la interfaz del usuario y la lógica de control en tres componentes relacionados.
- El patrón MVC es utilizado por los principales marcos de trabajo para construir aplicaciones web como Netbeans, Eclipse, entre otros.


USUARIO
SOLICITA
CONTROLADOR
COMUNICAN
DATOS
RESPONDE
VISUALMENTE
DATOS
MODELO
VISTA

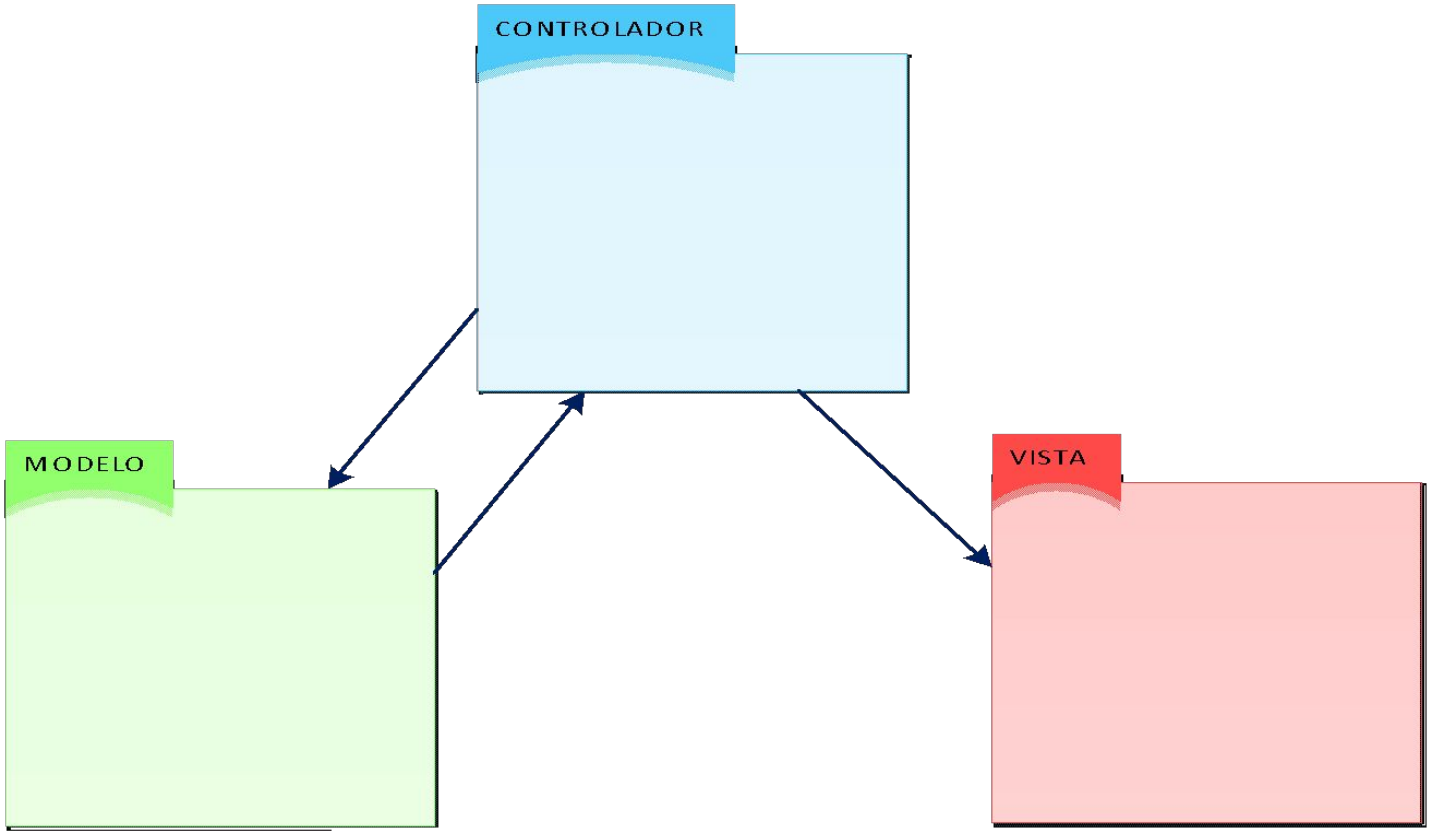
Patrón arquitectónico *Modelo Vista Controlador (MVC)*

- **Modelo** que se encarga representar a los datos de la aplicación y su estructura agrupando todos los componentes que tienen que ver con la persistencia de los datos. Se implementa con apoyo de algún manejador de bases de datos.
- **Vista**, agrupa todos los componentes que permiten la interacción con el usuario. La implementación frecuentemente es a través de páginas HTML.
- **Controlador**, establece la comunicación entre los dos componentes anteriores, agrupando los componentes que manejarán las reglas del dominio de la aplicación. El controlador recibe los eventos solocitados por el usuario, efectúa las operaciones necesarias solicitando los servicios al modelo y refleja los cambios en la vista.

Diagrama de paquetes

- Paquete es un mecanismo general de UML para organizar elementos en grupos.
- Un paquete se representa con una carpeta con su nombre.
- Los paquetes pueden contener otros paquetes, clases u otros elementos.
- Los elementos de cada paquete deben tener una alta cohesión y bajo acoplamiento con los elementos de otros paquetes.

Diagrama de paquetes de Modelo/Vista/Controlador



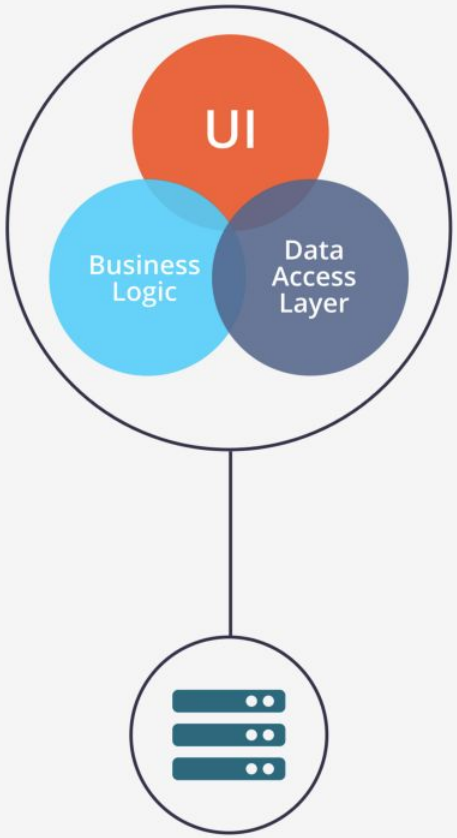
Desventajas de la arquitectura MVC

- Diseño arquitectónico **MVC** es una **arquitectura en la que todos los aspectos funcionales quedan acoplados e integrados** en un mismo **programa**. Esto **impacta al mantenimiento evolutivo** (cambio o agregación) por la creciente **complejidad e interdependencia** de los **componentes** del software.
- Este tipo de sistemas **está alojado en un servidor**, lo que dificulta la escalabilidad y disponibilidad.

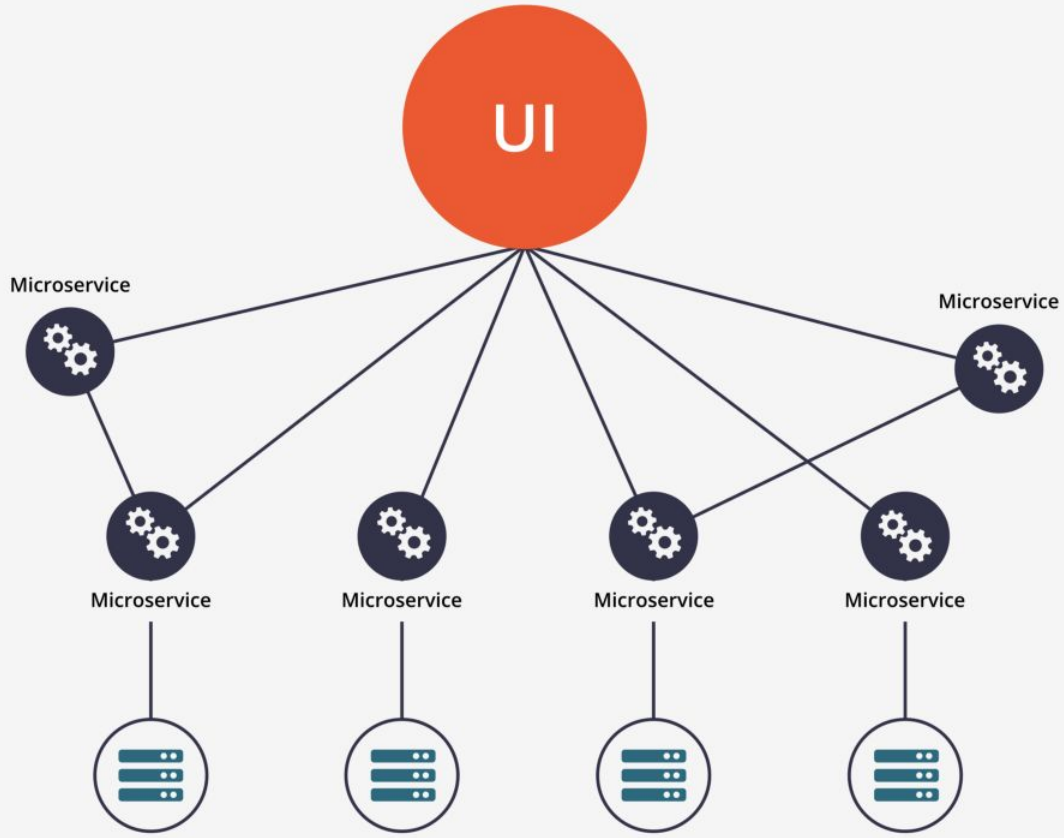
Arquitectura de Microservicios (MSA)

- **Arquitectura de microservicios** responde a la necesidad de facilitar la **realización de cambios** en el software e implementarlos de forma fácil y rápida.

- La idea es **dividir los sistemas en partes individuales**, permitiendo que se puedan **tratar y abordar los problemas de manera independiente** sin afectar al resto.



Arquitectura MVC

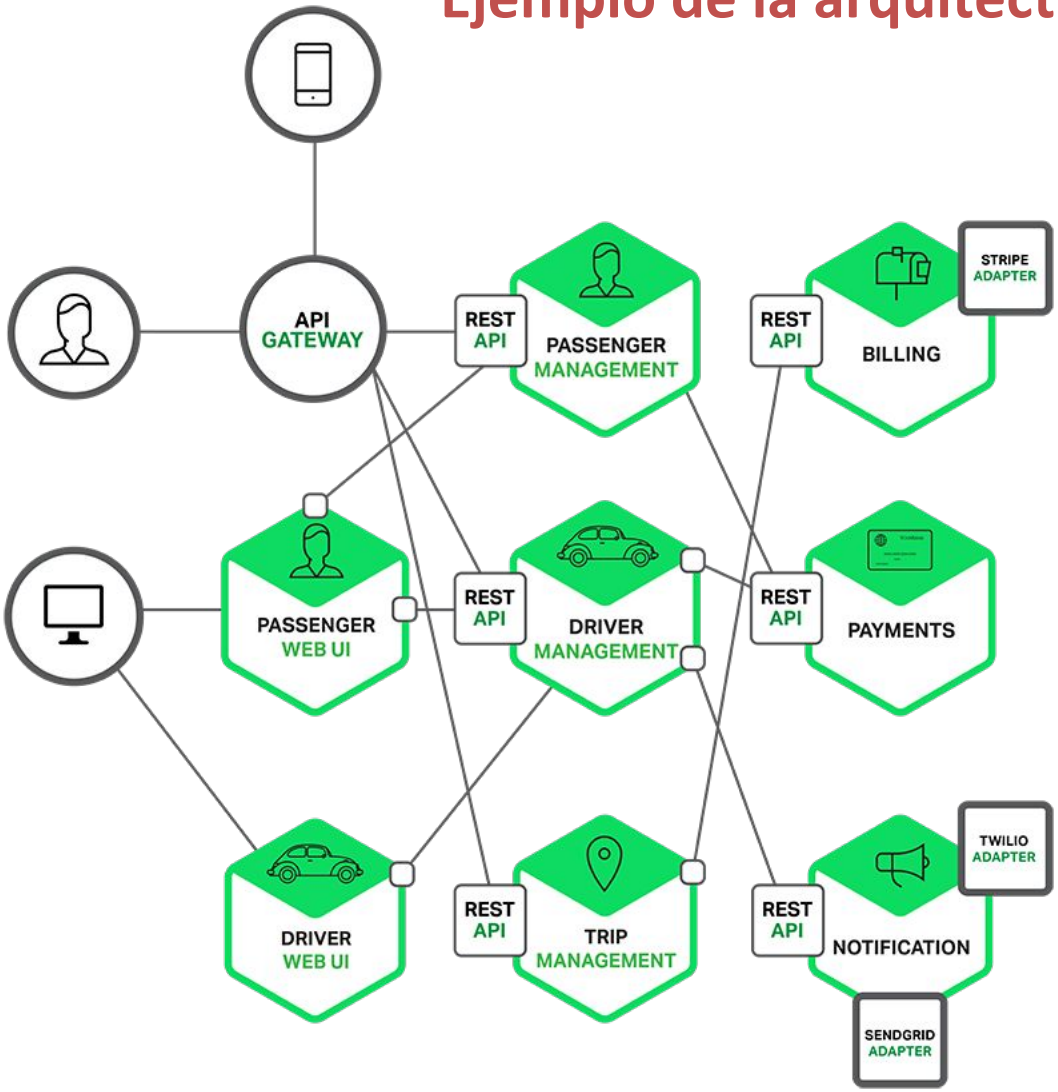


Arquitectura MSA

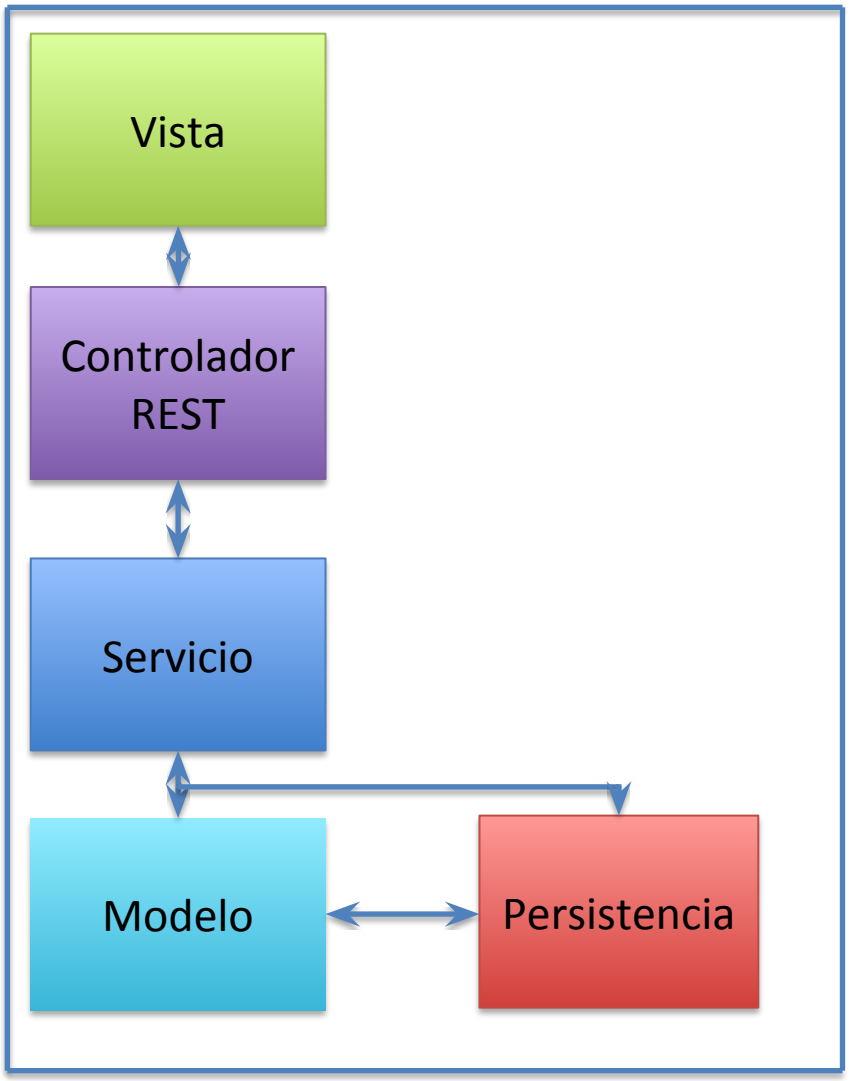
Arquitectura de Microservicios (MSA)

- Consiste en **construir una aplicación como un conjunto de pequeños servicios**, los cuales se ejecutan en su propio proceso y se comunican con mecanismos ligeros, normalmente una Application Programming Inteface (**API**) de recursos HTTP.
- Cada **servicio** se encarga de **implementar una funcionalidad completa del negocio**.
- Cada **servicio** es desplegado **de forma independiente,** puede estar **desarrollado en un lenguaje distinto al resto de servicios** y puede **usar diferentes tecnologías de almacenamiento de datos.**

Ejemplo de la arquitectura de MSA



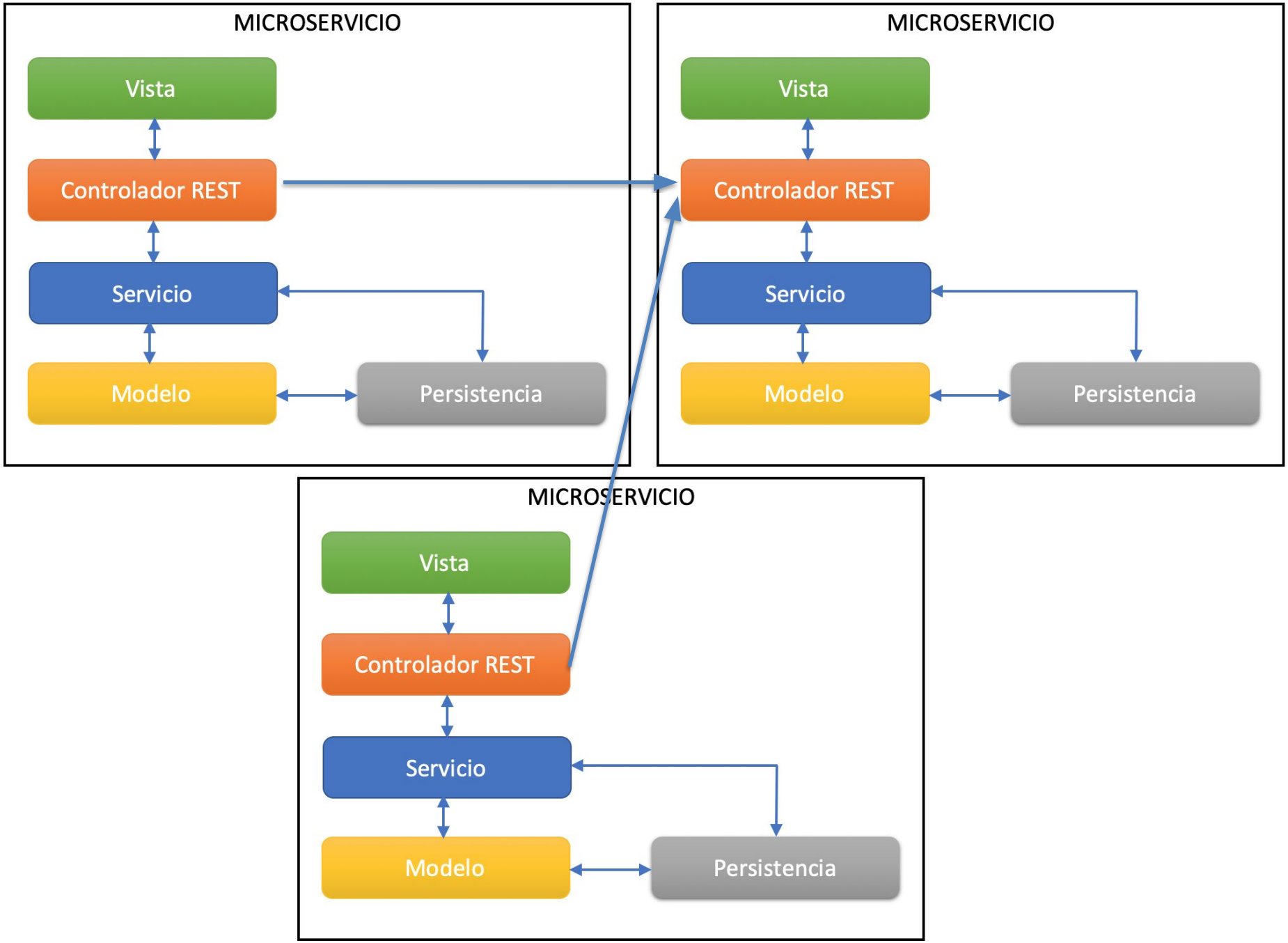
Microservicio



Contenedor

Microservicio

- **Vista** – interfaz visual y la interacción con el usuario
- **Controlador REST** -interfaz operativa entre sistemas que utiliza el protocolo HTTP para solicitar la ejecución de operaciones sobre los datos enviados y recibidos en formato JSON.
- **Servicio** – implementación de las funcionalidades
- **Modelo** – Abstracción de los datos que modelan el negocio (típicamente POJOS en Java)
- **Persistencia** – registro, actualización, borrado y la recuperación de los datos
- **Contenedor**- Elemento de infraestructura que agrupa los aspectos anteriores para lograr una ejecución independiente del servicio.


MICROSERVICIO
Vista
Controlador REST
Servicio
Modelo
Persistencia
MICROSERVICIO
Vista
Controlador REST
Servicio
Modelo
Persistencia
MICROSERVICIO
Vista
Controlador REST
Servicio
Modelo
Persistencia

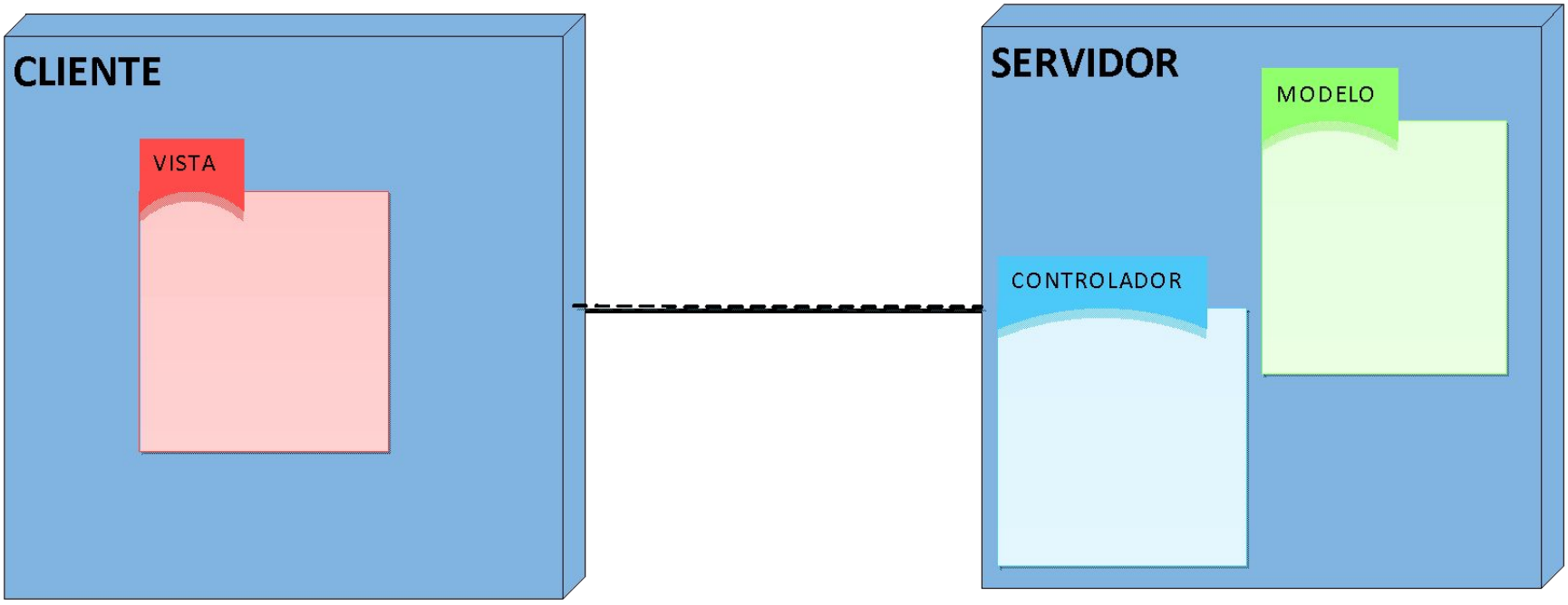
Diseñar la arquitectura de software

- Otra vista importante de arquitectura de software identifica la lógica del despliegue del software, es decir la asignación de los componentes a los distintos **nodos de hardware.**
- Esta vista se puede documentar con el **diagrama de despliegue** de UML.

Diagrama de despliegue

- Los diagramas de despliegue representan los componentes que se instalarán en cada elemento de hardware y las conexiones entre ellos.
- Los elementos gráficos de estos diagramas son: los *nodos* y las *conexiones* entre ellos.

Diagrama de despliegue para un sistema web



Frontend **Backend**

Seleccionar el ambiente de desarrollo

- Otra decisión importante del diseño es definir el ambiente de implementación.
- Dependiendo del tipo de aplicación a desarrollar, se selecciona el (o los) lenguaje(s) de programación, el ambiente y herramientas con que se hará la construcción.
- Esta decisión va tener mucha importancia para la duración del ciclo de vida de software y el costo de su mantenimiento.

Seleccionar el ambiente de desarrollo

- Seleccionar (o aceptar en el caso de que lo define el Cliente) el lenguaje (o lenguajes) de programación y las herramientas que se utilizarán para la construcción del software.

- Es importante registrar los nombres y las versiones de los elementos del ambiente de desarrollo para que todos miembros del equipo utilicen las mismas versiones y para fines del mantenimiento.

Diseñar los componentes

- Una vez identificados los **componentes** más abstractos de la arquitectura, se continua con la **identificación** de los **elementos a nivel mas detallado del diseño,** que son **clases**.
- Para detallar los componentes se construyen diagramas de clases y de secuencia de UML que muestran las vistas estáticas y dinámicas de los componentes.

Diagrama de clases

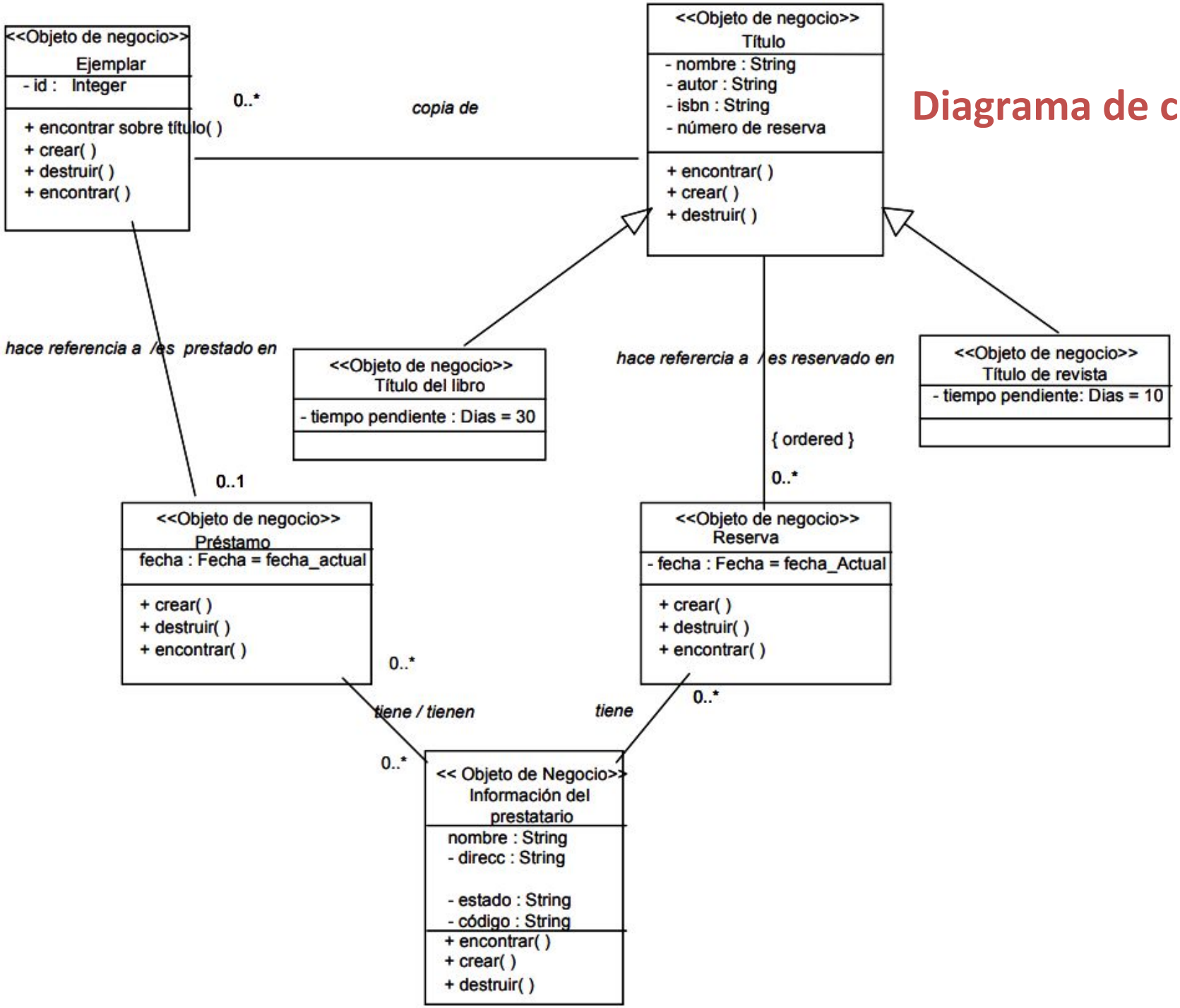
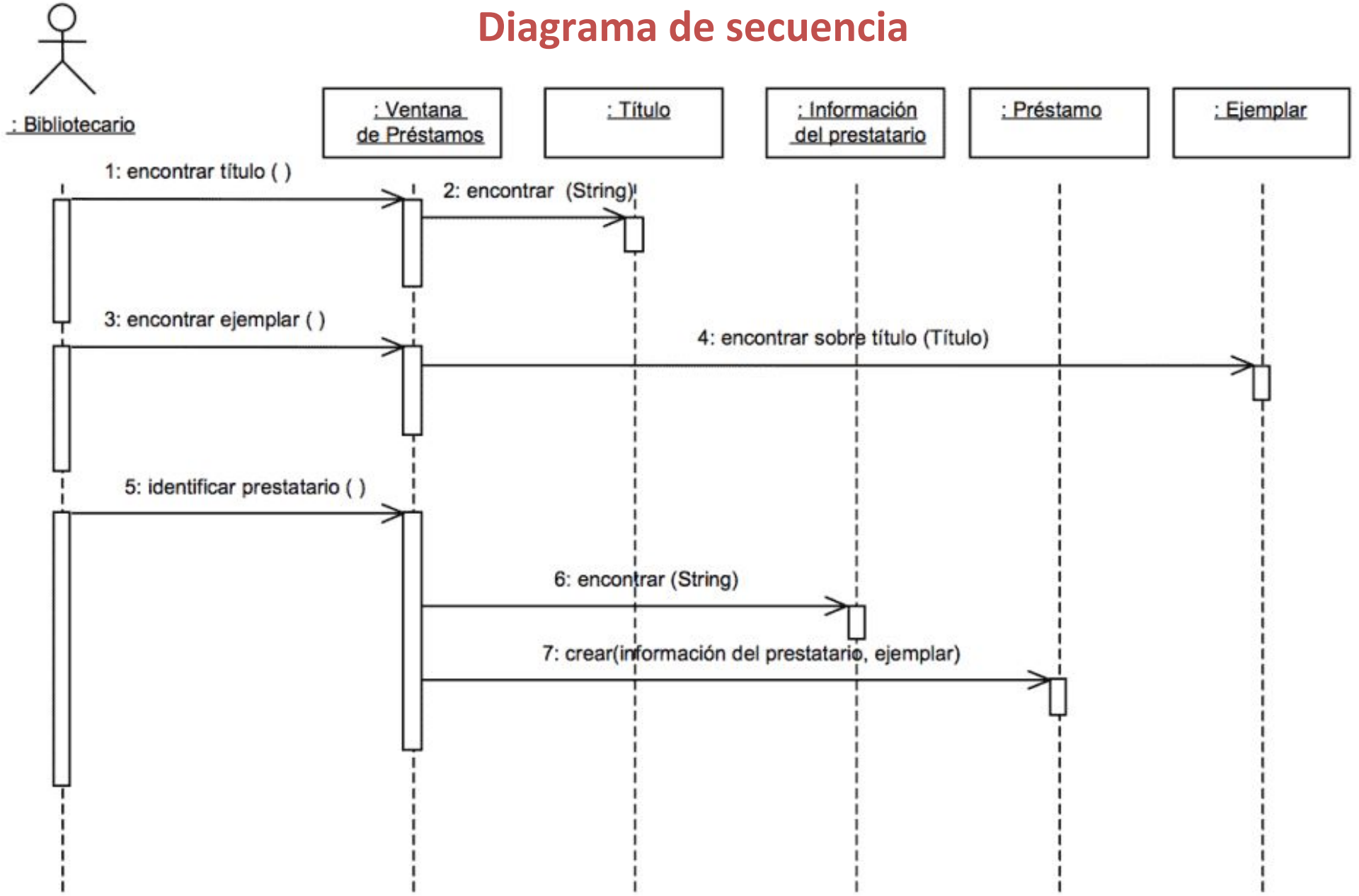


Diagrama de secuencia



Diseñar la base de datos

- A partir de las **clases** se identifica la información que debe ser **persistente** y se diseña la **base de datos.**
- Uno de los tipos de diagramas más utilizados para diseñar las bases de datos, es el de **Entidad-Relación**.
- Las clases y sus atributos se convierten en las Entidades y sus atributos.
- Las relaciones entre las Entidades correspoden a las relaciones entre clases.

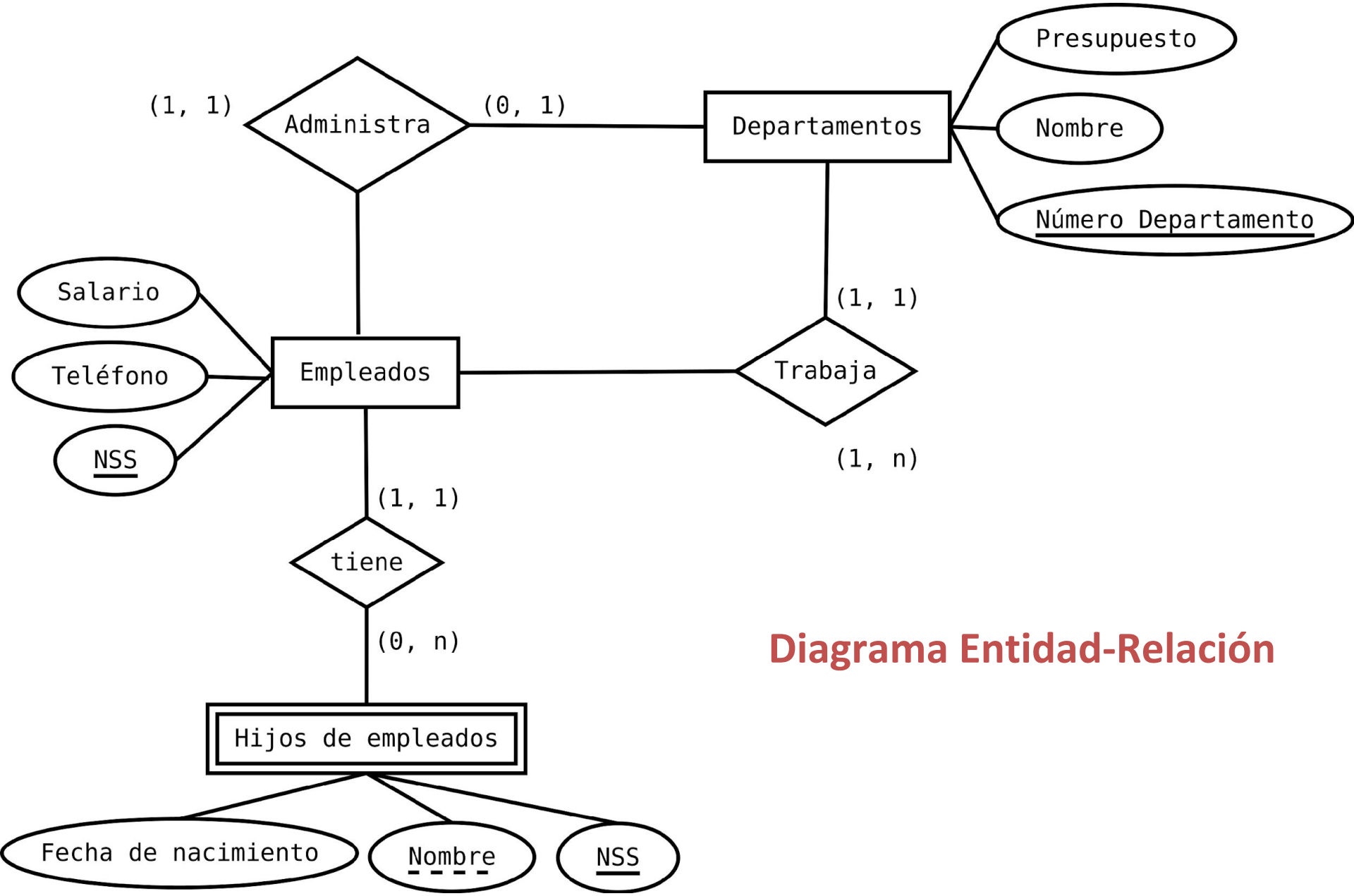


Diagrama Entidad-Relación

¿Qué hemos aprendido?

- ¿Cuál es la diferencia entre la arquitectura MVC y la de Microservicios?
- ¿Cuáles son las características de un microservicio?
- ¿Cuál es el papel del Controlador REST?
- ¿Qué diagramas de UML ayudan a definir la vista estática y la dinámica de los componentes?

PRACTICA PD2 DISEÑO DE SOFTWARE

PD2 Diseño de software

Objetivos

- Seleccionar la arquitectura del software y diseñar los componentes para poder cumplir con la *Especificación de requerimientos de software*.

PD2 Diseño de software

Entradas

Condiciones

- Requerimientos de software entendidos

Productos de trabajo

- *Especificación de requerimientos de software* de cada equipo
- Plantilla *Diseño General de Arquitectura de Microservicios y de la BD de la Iteración*
- Plantilla *Diseño detallado del Microservicio*
- Herramienta(s) colaborativa(s)

PD2 Diseño de software

Actividades

1. Establecer la arquitectura general del software.
2. Especificar el ambiente y las herramientas para el desarrollo de software.
3. Diseñar los componentes principales del software para cada caso de uso dentro del alcance de la iteración.
4. Diseñar la base de datos.
5. Integrar los documentos del *Diseño General de Arquitectura de Microservicios y de la BD de la Iteración* y *del Diseño detallado del Microservicio* (por equipo).

TÉCNICAS PARA REALIZAR LAS ACTIVIDADES DE PD2

Actividad 1 Establecer la arquitectura general del software.

- Los representantes técnicos de los equipos hacen el diagrama de paquetes que describe los microservicios principales del software de la iteración y sus relaciones.
- Un representante de los equipos genera el documento *Diseño General de Arquitectura de Microservicios y de la BD de la Iteración* *e* integra el diagrama de paquetes de microservicios.

Actividad 2 Especificar el ambiente y las herramientas para el desarrollo de software. **(1/2)**

Un representante de los equipos registra en el documento *Diseño General de Arquitectura de Microservicios y de la BD de la Iteración* los siguientes elementos del ambiente de implementación con su nombre y versión:

- Lenguaje (o lenguajes) de Programación
- IDE (Editor de desarrollo)
- Gestor de dependencias
- Framework de frontend
- Framework de backend
- Herramienta para Control de versiones
- Servidor de integración continua

Actividad 2 Especificar el ambiente y las herramientas para el desarrollo de software. **(2/2)**

- Repositorio de bibliotecas
- Herramienta para Control de calidad del código
- Diagramador UML
- Herramienta para Prototipado de interfaz de usuario
- Manejador de bases de datos
- Visualizador de la BD
- Tecnología de contenerización
- Sistema operativo

Actividad 3. Diseñar los componentes principales del software para cada caso de uso dentro del alcance de la iteración.

- Para el diseño detallado de los componentes usaremos los diagramas de clases, de paquetes y de secuencia de UML.
- Los diagramas de clases y de paquetes servirán para definir la vista estática de los componentes.
- Los diagramas de secuencia servirán para definir la vista dinámica de los componentes.

Identificación de clases en una Arquitectura orientada a Microserivicios MSA (1/6)

En una arquitectura orientada a Microservicios (MSA) los únicos elementos a identificar son los siguientes 5 (cinco) elementos:

1) **Vista** - Interfaz de usuario. Se refinan los prototipos de pantalla incluidos en la *Especificación de requerimientos de software*. Se modelan como clases las pantallas con atributos que representan los datos que se despliegan o capturan y los métodos que representan las posibilidades de interacción que tenga el usuario.

Identificación de clases en una Arquitectura orientada a Microserivicios MSA (2/6)

2) Controladores REST. Son clases que indican qué funciones de microservicio van a estar disponibles para su consumo.

Cada función tendrá un nombre (url compatible con la convención http

: https://tools.ietf.org/pdf/rfc2616.pdf) y podrá ser de tipo GET, POST, PUT o DELETE, que significan: selección, inserción, actualización o borrado respectivamente.

Cada Controlador REST define el tipo de parámetros que recibe y el tipo de datos que regresa. Internamente, un Controlador REST típicamente convertirá y des-convertirá todo a formato JSON.

.

Identificación de clases en una Arquitectura orientada a Microserivicios MSA (3/6)

2) Controladores REST.

Continua:

Un Controlador REST NO contiene lógica de negocio. Siempre invoca a un "Plain service" para proveer la funcionalidad ofrecida.

Para este curso, la seguridad de acceso a los "Plane Services" si será responsabilidad de estos controladores. Se recomienda usar este estándar de nombramiento:

- https://restfulapi.net/resource-naming/,
- Y las mejores prácticas:
- https://florimond.dev/blog/articles/2018/08/restful-api-design-13-best-practices-to-make-your-users-happy/

NOTA: Una clase de Controlador REST típicamente contiene varios métodos (o endpoints) y **varios controladores hacen un microservcio**.

Identificación de clases en una Arquitectura orientada a Microserivicios MSA (4/6)

3) Servicio. Clases de servicio, también conocidas como "Plane services", son clases que contienen la lógica de negocio y no saben nada acerca de cómo serán consumdas ni por quién. Tampoco incluyen cuestiones de seguridad, ni de escalabilidad o tolerancia a fallos.

Se comunican con clases de modelo, con elementos de persistencia e invocan a otros servicios para ofrecer funcionalidad pura de negocio.

Identificación de clases en una Arquitectura orientada a Microserivicios MSA (5/6)

3) Servicio.

Continua: Para nombramiento de clases, atributos y métodos se debe usar el estándar:

https://www.oracle.com/java/technologies/javase/codeconventions-programmingpractices.html

Nota. Este tipo de clases son las que deben ser cubiertas por las pruebas unitarias de ser posible, en un 100%. Deben ser algorítmicamente eficientes en términos de complejidad O(n) y de uso de memoria.

Identificación de clases en una Arquitectura orientada a Microserivicios MSA (6/6)

4) **Modelo**. Clases de modelo, también conocidas como DTO o bien como VO (Value Objects / Data Transfer Object). Son típicamente POJOS. (Plane Old Java Objects)
5) **Persistencia**. Se modelan como interfaces de Java y definen operaciones de persistencia al asociar el nombre de ciertos métodos con cadenas SQL.

Ejemplo Login simple (1/2)

- **Vista**: HTML con 2 cajas de texto y un botón de “Aceptar”. La primer caja de texto es para capturar el nombre y la segunda caja de texto es para capturar la clave de acceso.

- **Controlador REST**: Clase Java que define un método (o endpoint) “login”. Este controlador invoca al “Plain service” login y regresará un objeto de tipo POJO “ResponseLogin”.

- **Servicio**: Clase Java que implementa el método “login” haciendo uso del método findById de la interfaz de persistencia, de la clase de modelo Usuario y del POJO RequestLogin. Contiene lógica para validar la clave de acceso y tiene la capacidad de indicar si el intento de login fue exitoso o no. La respuesta será de tipo POJO “ResponseLogin”.

Ejemplo Login simple (2/2)

- **Modelo**:
  - POJO “RequestLogin” que contiene dos atributos de tipo “String”, a saber: usuario y clave.
  - POJO “Usuario” que modela un objeto de tipo Usuario. Contiene todos los atibutos asociados a un Usuario.
  - POJO “ResponseLogin” que contienen tres atributos: succeed:boolean, message:String, code:int
  - Cada POJO contiene setters y getters para cada atributo, un método equals, un método hash y un método toString.
- **Persistencia**: Interfaz Java con 5 métodos: select all, findById, insert, update y delete. Cada método recibe y regresa objetos de tipo Usuario.

Identificación de clases (1)

- Para identificar las clases, se analiza cada caso de uso para imaginar qué clases se necesitan para modelar a los componentes de la Vista del Controlador, del Servicio y del Modelo.

- El responsable técnico coordina la identificación de las clases solicitando a cada miembro del equipo que ayude a identificar las clases para cada una de las capas del microservicio.

Identificación de clases (2)

- **Clases para la Vista** (interfaz con el usuario). Son los elementos con los que interactúa directamente el usuario.
- **Clases de Controlador**. Son los elementos que definen la interfaz del servicio, es decir la descripción de entradas y los resultados de todas las funcionalidades que implementa un servicio acorde con la interfaz del servicio del Controlador.
- **Clases de Servicio**. Son los elementos que implementan las funcionalidades del servicio,.
- **Clases para el Modelo**. Son los elementos que representan los conjuntos de datos de la aplicación y manejan su persistencia.

Identificar clases y sus relaciones

Ejemplo:

- Para ingresar al sistema, se necesita alguna interfaz de usuario que reciba los datos de ingreso, esto se puede implementar a través de código HTML y se modela como una clase en la Vista.
- Para almacenar las claves de ingreso, se construye una clase correspondiente en el componente del Modelo y se diseña una tabla en la base de datos.
- En el Controlador se construye una clase llamada *Ingreso* con métodos para revisar si los datos dados por la Vista corresponden con los almacenados en el Modelo.

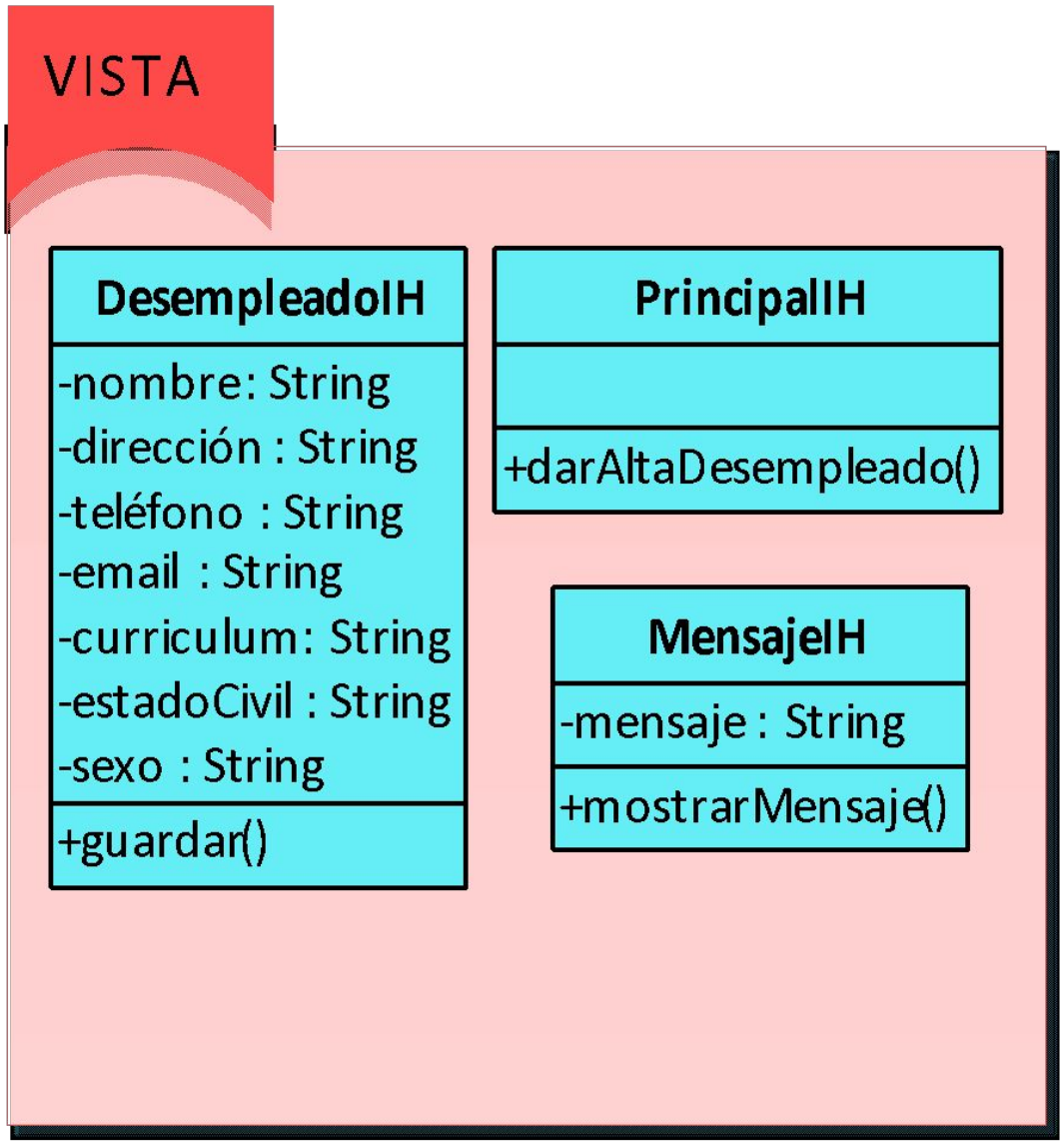
Identificación de clases del Controlador

- Los miembros del equipo revisan el detalle de sus casos de uso y crean una clase (o mas) para el componente del Controlador que se responsabilizará por realizar la funcionalidad básica del caso de uso.
- A esta clase se le da el nombre similar al caso de uso pero usando uno o mas sustantivos.
- Esta clase va a tener por lo menos un método que corresponda a la funcionalidad del caso de uso.
- Un controlador lleva nombres especiales para la exposición de un endopint.
- VER: https://restfulapi.net/resource-naming/

```java
@RestController
@Api(value = "administracion")
@RequestMapping(value = "/api")
public class UsuarioRest {
    private UsuarioService usuarioService;
    private UsuarioDetallesService usuarioDetallesService;

    public UsuarioRest(UsuarioService usuarioService, UsuarioDetallesService usuarioDetallesServ
        this.usuarioService = usuarioService;
        this.usuarioDetallesService = usuarioDetallesService;
    }

    @GetMapping(
            value = "/usuarios.json",
            produces = "application/json; charset=utf-8")
    public List<Usuario> getAllUsuario() throws BusinessException {
        return usuarioService.getAll();
    }

    @GetMapping(
            value = "/usuario.json",
            produces = "application/json; charset=utf-8")
    public Usuario getUsuario(@RequestParam long id) throws BusinessException {
        return usuarioService.getById(id);
    }

    @PostMapping(
                value = "/usuario.json",
                produces = "application/json; charset=utf-8")
    public int insert(@Valid @RequestBody Usuario usr) throws BusinessException {
        return usuarioService.insert(usr);
    }

    @PutMapping(
                value = "/usuario.json",
                produces = "application/json; charset=utf-8")
    public int update(@RequestBody Usuario usr) throws BusinessException {
        return usuarioService.update(usr);
    }

    @PostMapping(
```

Identificación de clases de Vista

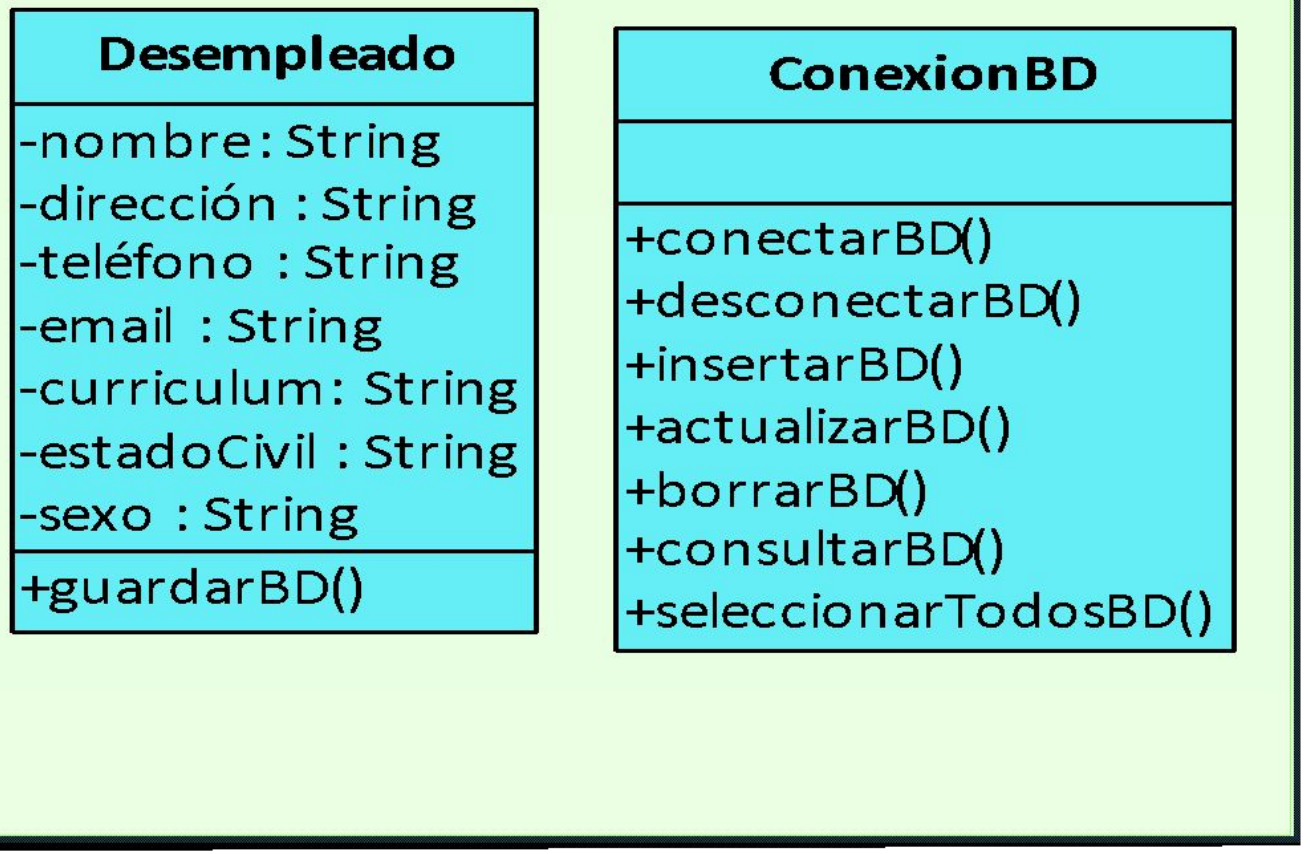
- Para identificar las clases que representarán la interfaz con el usuario se revisa el prototipo de interfaz de cada caso de uso en el documento de requerimientos y se dibuja una clase por cada pantalla.
- A la clase se da el nombre de la pantalla.
- Las clases de interfaz son las abstracciones que podrán ser implementadas con diversas tecnologías como por ejemplo código HTML o jsp.


VISTA
DesempleadoIH
-nombre: String
-dirección : String
-teléfono : String
-email : String
-curriculum: String
-estadoCivil : String
-sexo : String
+guardar()
PrincipalIH
+darAltaDesempleado()
MensajeIH
-mensaje : String
+mostrarMensaje()

Identificación de clases de Modelo

- Para identificar las clases del Modelo se analizan los detalles de casos de uso buscando conjuntos de datos que representan cierto tipo de información que manejará el sistema y la cual va requerir de ser resguardada y/o persitente.

MODELO


Desempleado
-nombre: String
-dirección : String
-teléfono : String
-email : String
-curriculum: String
-estadoCivil : String
-sexo : String
+guardarBD()
ConexionBD
+conectarBD()
+desconectarBD()
+insertarBD()
+actualizarBD()
+borrarBD()
+consultarBD()
+seleccionarTodosBD()

Integración de los diagramas de clases

- Cuando todos los miembros del equipo han analizado sus casos de uso y han identificado las clases de cada componente de MSA, el **Responsable técnico** dirige la integración de clases de cada componente MSA en un diagrama de clases siguiendo la práctica PS3.
- En esta integración se toman las decisiones sobre la unificación de los atributos y métodos de las clases comunes y se establecen las relaciones entre clases.

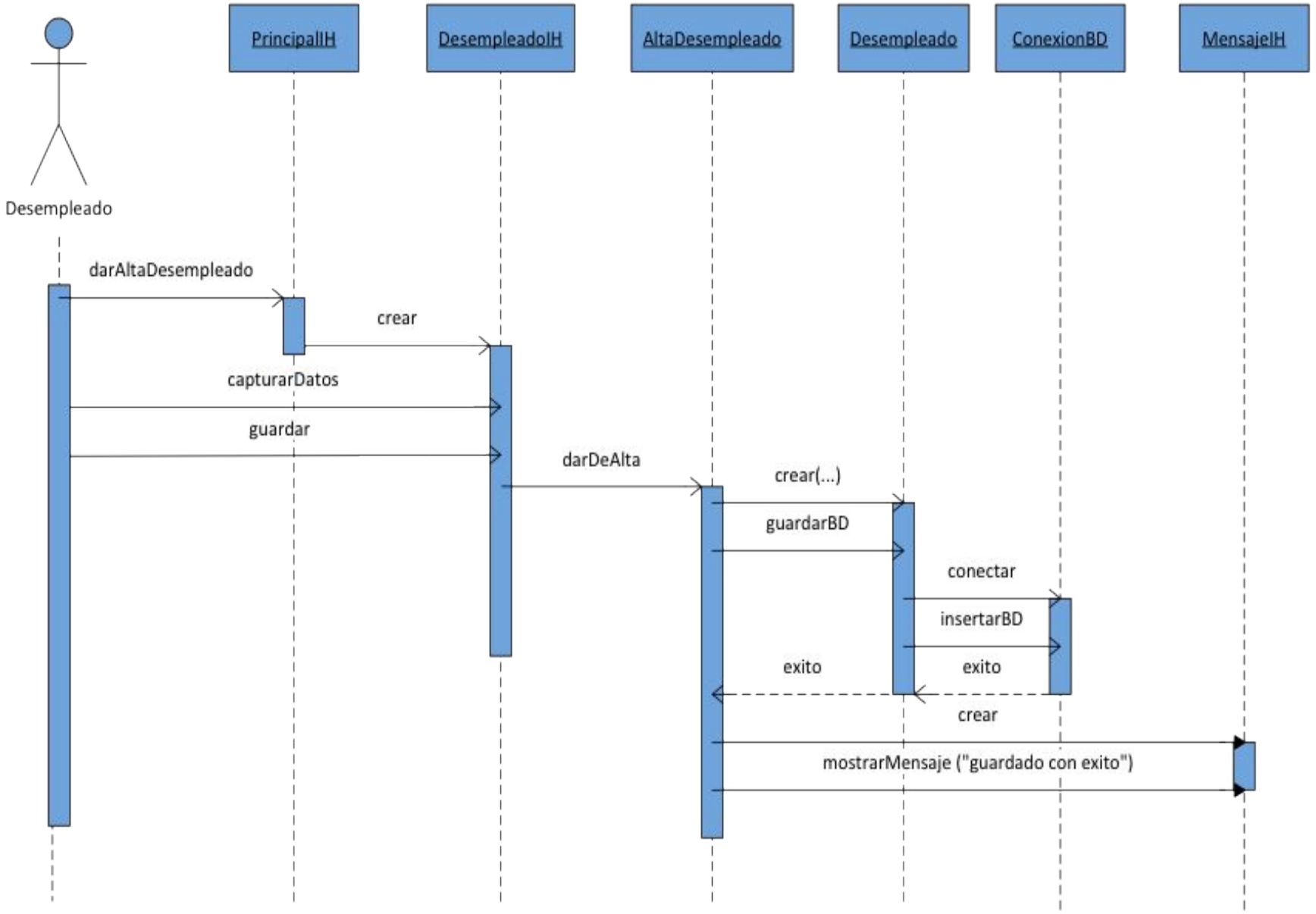
Hacer los diagramas de secuencia

- Los **diagramas de secuencia** y los **de estados** representan la ***vista dinámica*** del sistema.

- De acuerdo a esta vista se codificarán los algoritmos de intercambio de mensajes entre los objetos de las clases en el lenguaje de programación elegido para la construcción.

Diagramas de secuencia

- Los diagramas de secuencia muestran la interacción entre los objetos de las clases como una secuencia de envío de mensajes entre ellos, ordenados en el tiempo.
- Para su construcción se parte del detalle de los casos de uso. Para cada flujo normal, alternativo y excepcional de eventos en los casos de uso, se construye un diagrama de secuencia.


Desempleado
PrincipalIH
DesempleadoIH
AltaDesempleado
Desempleado
ConexionBD
MensajeIH
darAltaDesempleado
crear
capturarDatos
guardar
darDeAlta
crear(...)
guardarBD
conectar
insertarBD
exito
exito
crear
mostrarMensaje ("guardado con exito")

Hacer diagrama de secuencia (1)

- Por cada flujo de un caso de uso se crea un diagrama de secuencia de la siguiente manera:
  - Se representa al actor que corresponde al caso de uso poniéndolo arriba en el extremo superior izquierdo del diagrama.
  - El actor inicia las acciones del caso de uso enviando un mensaje a un objeto de una clase de la Vista que corresponde a la primera pantalla de este flujo. Se dibuja este objeto al lado derecho del actor. Los objetos, incluyendo al actor, tienen una linea punteada vertical que representa su "vida" en el tiempo.

Hacer diagrama de secuencia (2)

- Enseguida se analiza cual de las clases del Controlador tiene el método que respondería a la petición hecha por el actor y se dibuja un objeto de esta clase a la derecha del objeto anterior.
- Se identifican la(s) clase(s) del Servicio involucradas en el flujo del caso de uso y se dibujan sus objetos de manera parecida.
- Se continua con el análisis para identificar las clases del Modelo necesarias para completar el flujo dibujando sus objetos.
- Terminando con la representación de la Persistencia.

Hacer diagrama de secuencia (3)

- El flujo de eventos de un caso de uso se representa como envío de mensaje de un objeto al otro.
- Cada mensaje entre objetos aparece como una flecha dirigida del objeto fuente al objeto receptor, etiquetado con el nombre del método correspondiente.
- La duración de la ejecución de un método se representa como una barra más gruesa en la línea de vida del objeto.
- Para visualizar el orden temporal del envío de los mensajes, la flecha de un mensaje posterior se dibuja más abajo que la del mensaje anterior.

Refinamiento de diagramas de clases

- Una vez terminados los diagramas de secuencia para todos los casos de uso, se revisa la consistencia entre los diagramas de clase y de secuencia.
- Si alguna clase no se usó en ningún diagrama de secuencia se elimina del diagrama de clases.
- Si faltó alguna que no está en el diagrama de clase, se incorpora.
- Se modifican los diagramas de clases según lo encontrado durante la construcción de los diagramas de secuencia.

Actividad 4. Definir la base de datos (1/2)

- A partir de las clases de Modelo de todos los microservicios, que requieren de la persistencia de sus datos , se diseña la base de datos.
- Para modelar la base de datos se usan los conocimientos y técnicas del curso de Bases de Datos.
- Uno de los diagramas más utilizados para este fin, es el de Entidad-Relación, en el cual se hace la correspondencia entre las clases del diagrama de clases del Modelo, que se considere sean persistentes, con una Entidad del diagrama Entidad-Relación.
- Los atributos de las clases se convierten en los atributos de las Entidades. Las relaciones entre las Entidades se construyen a partir de las relaciones entre clases.

Actividad 4. Definir la base de datos(2/2)

- En una reunión de toma de decisiones (ver la práctica PS3 Unidad 3) se hace el diseño de la Base de Datos.
- Cada equipo propone el diagrama E-R de los datos que necesita para realizar su microservicio.
- Los representantes de los equipos revisan y discuten sus propuestas para llegar a una definición del diagrama E-R unificado que satisface las necesidades de todos los equipos.

Actividad 5. Integrar los documentos del *Diseño General de Arquitectura de Microservicios y de la BD de la Iteración* y del *Diseño detallado del Microservicio* (por equipo)

- Los Resposables del equipo de integración designan a alguien responsable de integrar todos los elementos al *Diseño General de Arquitectura de Microservicios y de la BD de la Iteración* y resguardarlo en el repositorio compartido.
- E*l Responsable de Calidad* de cada equipo se responsabiliza por integrar *Diseño detallado del Microservicio* (por equipo) y resguardarlo en el repositorio compartido.

PD2 Diseño de software

Resultados

Condiciones

- Diseño del software completo

Productos de trabajo

- *Diseño General de Arquitectura de Microservicios y de la BD de la Iteración* y del *Diseño detallado del Microservicio* (por equipo) documentados y resguardados.

Resultados de esta unidad

- **Condiciones**
  - Diseño del software completo
- **Productos de trabajo**
  - *Diseño General de Arquitectura de Microservicios y de la BD de la Iteración*
  - *Diseño detallado del Microservicio* (por equipo)

Bibliografía

- Ambler S.W. (2005). *The Elements of UML 2.0 Style.* Cambridge University Press.
- Booch G., R. J. (1998). *The Unified Modeling Languajes. Users Guide.* Addison Wesley.
- Buschamann F., R. M. (1996). *A System of Patterns. Patterns-oriented software architecture.* John Wiley.
- Erich Gamma, Richard Helm, Ralph Johnson , John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Design.* Addison Wesley, Enero 1995
- ISO/IEC25010. (2011). *Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models.* ISO.
- ISO/IEC29110. (2011). *29110-5-1-2Software Engineering-lifecycle Profiles for Very Small Entities Management and Engineering Guidde. s.1.* Software Engineering.
- Pressman R:S. (n.d.). *Ingeniería de software. Un enfoque práctico.* Mc Graw Hill.
- Rosenberg D., S. K. (2001). *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example.* Addison Wesley.
- Schmuller J. (2000). *Aprendiendo UML en 24 horas.* Prentice Hall.
- SWEBOK. (2014). *Guide to the Software Engineering Body of Knowledge v3.0.* IEEE Computer Society.
- Wikipedia. https://es.wikipedia.org/wiki/Arquitectura_de_microservicios