SEGUNDO PARCIAL - DISEÑO DE SISTEMAS

PRUEBAS DE SOFTWARE 4
PRUEBAS DE SOFTWARE 4
Objetivos de las pruebas 4
Verificación y validación 4
Inspecciones vs. Pruebas 4
EQUIVOCACIÓN - DEFECTO - FALLA 4
Casos de prueba 4
Tipos de pruebas 5
GENERACIÓN DE CASOS DE PRUEBA 6
Complejidad ciclomática 6
Condiciones de borde 8
Tipos y tamaños de datos 9
Dominio de valores 10
Formato de valores 10
ARQUITECTURA DE INTERACCIÓN 11
MVC 11
API: la posta 11
Perdón, pero ¿qué es HTTP? 11
Peticiones HTTP 11
Métodos de petición HTTP 12
Códigos de estado HTTP 12
API: NIVEL 0 13
API: NIVEL 1 13
API: NIVEL 2 13
API: NIVEL 3 13
ORM Tecnología 14
Qué es qué 14
Diseño de JPA 14
persistence.xml 15
HIBERNATE: Estados de una entidad 15
PATRONES DE DISEÑO 17
PATRÓN FACTORY METHOD (creacional) 17
Cómo implementarlo 17
ABSTRACT FACTORY (creacional) 18
Cómo implementarlo 18
PATRÓN TEMPLATE METHOD (de comportamiento) 19
Cómo implementarlo 19
PATRÓN COMMAND (de comportamiento) 20
Cómo implementarlo 20
TECNOLOGÍAS 21
SEMINARIO DISEÑO Y MAQUETADO WEB 21
HTML + CSS 21

¿Qué es HTML? 21
Etiquetas, atributos y elementos 21
HTML - HyperText Markup Language. 21
Cómo crear un documento .html 22
ESTRUCTURA BÁSICA DE UN .html 22
The HTML Element 22
Tags HTML 22
CSS Cascading Style Sheets 23
¿Cómo relacionar el HTML con el CSS? 24
Atributos 24
Selectores 24
¿Qué es el atributo display? 25
display:block; VS display:inline; 25
JavaScript 25
¿Para qué nos sirve JavaScript? 25
¿Cómo inyectar JS en nuestro documento HTML? 26
Sintaxis 26
DOM Document Object Model 27
DOM API 27
Event listeners 28
PATRONES ARQUITECTÓNICOS 28
CATEGORIZACIÓN DE PATRONES 28
Patrones de Componentes y Subsistemas 28
Patrones de Distribución 28
Patrones de Concurrencia 29
Patrones de Memoria 29
Patrones de Recursos 29
Patrones de Seguridad y Confiabilidad 29
ALGUNOS PATRONES EN PARTICULAR 30
Arquitecturas por flujo de datos (dataflow) 30
Estructura 30
Variantes 30
ARQUITECTURAS CALL & RETURN 30
Estilos 30
CLIENTE-SERVIDOR 31
Estructura 31
Componentes 31
Características 31
Ventajas y desventajas 31
ARQUITECTURA EN CAPAS 31
Estructura 31
Ventajas y desventajas 32
REPOSITORIO (CENTRADA EN DATOS) 33
Estructura 33
Ventajas y desventajas 33
BROKER 34
Características 34

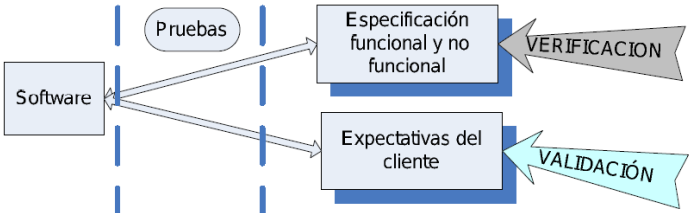
Estándares 34
EXTRACT TRANSFORM AND LOAD (ETL) 34
PEER-TO-PEER 34
Estructura 34
Ventajas y desventajas 34
PATRONES DE INTERACCIÓN (INTERFAZ) 35
¿Patrón Arquitectónico o de Interacción? 35
MVC VS MVVM 35
PATRONES DE INTEGRACIÓN 35
Integración 35
Patrones de integración 36
TECNOLOGÍAS FRONTEND 36

PRUEBAS DE SOFTWARE

Objetivos de las pruebas

- La prueba es un proceso de ejecución de un programa con la intención de descubrir un error (no busca "no encontrar errores" sino "sí encontrar errores").
- Un buen caso de prueba es aquel que tiene una alta probabilidad de descubrir un error no descubierto hasta entonces.
- Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Verificación y validación



Inspecciones vs. Pruebas

Inspección: técnicas estáticas: no se necesita ejecutar sw.
Pruebas: técnicas dinámicas: ejecución del software.

Inspecciones

- Analizan y comprueban las representaciones del sistema (ERS, modelos, código fuente).
- En todas las etapas del proceso.
- No se necesita ejecutar software.
- Localizan defectos de datos, controles, interfaz. Detectan anomalías como código no utilizado, variables sin inicializar, etc.

Pruebas

- Implican ejecutar una implementación del sw con datos de prueba.
- Se examinan salidas del software y entorno operacional.

EQUIVOCACIÓN - DEFECTO - FALLA

Equivocación: una acción humana que produce un resultado incorrecto.
Defecto: paso, proceso o definición de dato incorrecto. Ausencia de ciertas características.
Falla: resultado de ejecución incorrecta. Es el resultado obtenido y distinto al resultado esperado.
Una equivocación llega a uno o más defectos, que están presentes en el código. Un defecto lleva a cero, una o más fallas. La falla es la manifestación del defecto. Una falla tiene que ver con uno o más defectos.

Casos de prueba

¿Quién define los casos de prueba? ¿Quién los ejecuta? Importancia de los datos de prueba, preparación de lotes, etc.
Conjunto de valores de entrada, condiciones previas de ejecución, resultados esperados y las condiciones posteriores de ejecución, desarrollado para un objetivo particular o condición de prueba, tales como el ejercicio de un programa en particular o para verificar el cumplimiento de un requerimiento específico.
Conjunto de valores de entrada, condiciones previas de ejecución, resultados esperados y las condiciones posteriores de ejecución, desarrollado para un objetivo particular o condición de prueba, tales como el ejercicio de un programa en particular o para verificar el cumplimiento de un requerimiento específico.

¿Quién define los casos de prueba? ¿Quién los ejecuta? ¿Cuándo se ejecuta?

Tipos de pruebas

Pruebas de unidad

- Pruebas de unidad: componentes individuales.
- Objetivo: encontrar defectos en componentes (funciones, clases, otros componentes).
- Diseñar las pruebas que incluyan
  - Pruebas aisladas de todas las operaciones asociadas con el objeto.
  - Asignación y consulta de todos los atributos asociados con el objeto.
  - Ejecutar el objeto en todos sus posibles estados.

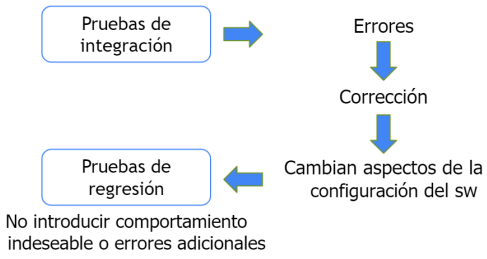
Smoke Test

- Test inicial. Se realiza al comienzo del proceso de pruebas.
- El tiempo de ejecución del smoke test completo debe ser corto. Los casos por lo general son positivos y lo que se espera encontrar es problemas en el proceso de despliegue (deploy) o que algún caso de uso clave en las pruebas, tiene algún problema que no podrá realizarse una prueba completa.
- Busca asegurar que la funcionalidad básica del software funciona correctamente.
- Sirve para quedarnos tranquilos que “por arriba” los casos de uso principales van a poder ser testeados.

Pruebas del sistema (integración)

- Integrar dos o más componentes y probar el sistema integrado.
- Comprobar
  - Componentes funcionan juntos.
  - Llamados correctamente.
  - Transfieren datos correctos.
- Incluir pruebas de rendimiento (estrés) e interfaz entre componentes.

Pruebas de Regresión



Pruebas de Aceptación (Uat)

- Denominado UAT (User Acceptance Testing)
- Este tipo de pruebas debe ser realizado por el cliente para quien fue desarrollado el cambio o producto.
- Son básicamente pruebas funcionales, sobre el sistema completo, y buscan una cobertura de la especificación de los requisitos y del manual del usuario.
- Estas pruebas nunca deben realizarse durante el desarrollo o testing, pues sería impresentable al cliente; sino que se realizan sobre el producto terminado.

GENERACIÓN DE CASOS DE PRUEBA

Complejidad ciclomática

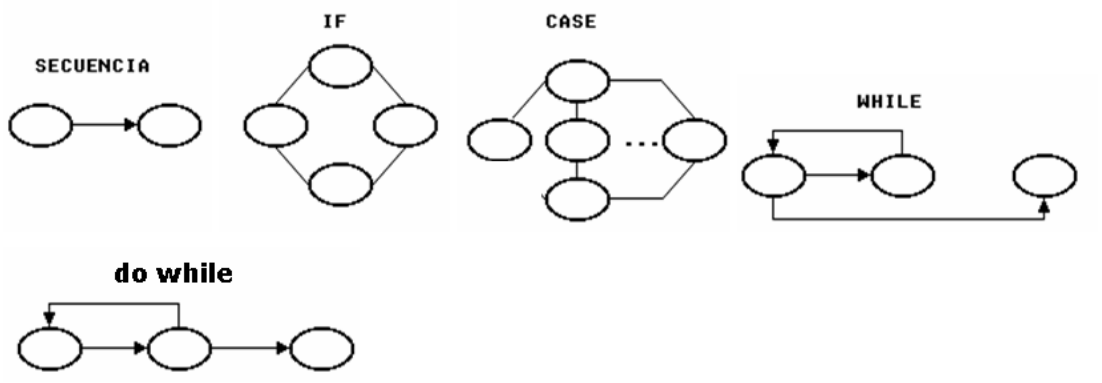
Basado en algoritmo / secuencialidad

Tiene su origen en el paradigma imperativo.

Aplicable en programación orientada a objetos para el análisis al interior de los métodos.

La Complejidad Ciclomática (Cyclomatic Complexity) es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Es una de las métricas de software más ampliamente aceptada, ya que ha sido concebida para ser independiente del lenguaje.
Esta métrica, propuesta por Thomas McCabe en 1976, se basa en el diagrama de flujo determinado por las estructuras de control de un determinado código. De dicho análisis se puede obtener una medida cuantitativa de la dificultad de de crear pruebas automáticas del código y también es una medición orientativa de la fiabilidad del mismo.

- Grafos de flujos:

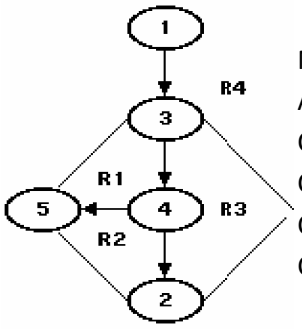


- Nodos (N): cada uno de los círculos
- Aristas (A): cada una de las líneas
- Complejidad ciclomática (CC) = A – N + 2

Casos de prueba

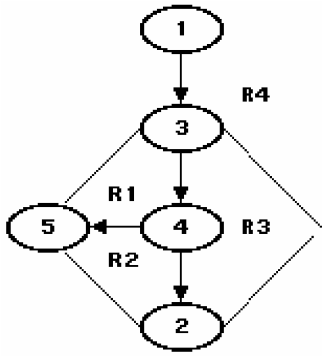
- Se debe definir por lo menos un caso de prueba para cada uno de los caminos existentes en el grafo.
- La cantidad de caminos del grafo es la complejidad ciclomática

Ejemplo básico:



N = 5
A = 7
Complejidad ciclomática
CC = A – N + 2
CC = 7 – 5 + 2
CC = 4

Ejemplo básico:



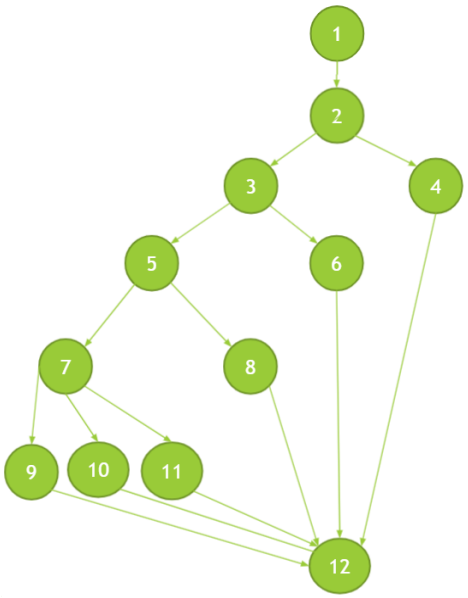
La Complejidad Ciclomática indica que deben generarse casos de pruebas para 4 escenarios:
Nodos 1, 3, 5 y 2
Nodos 1, 3, 4 y 2
Nodos 1, 3 y 2
Nodos 1, 3, 4, 5 y 2

Ejemplo Asignador:

Estamos desarrollando un software en un Videojuego para asignar acciones a un determinado Jugador. De cada jugador se conoce su nombre y edad. Las acciones se asignan según diversos criterios. Vamos a tener una clase

Jugador con los atributos nombre, edad y acción asignada. Vamos a tener una clase Asignador que será la responsable de toda la lógica de asignación de la acción al jugador.
Si la primera letra del nombre es A, se asigna acción A. Sino, se verifica si la segunda letra del nombre está en la primera mitad del abecedario (o es ñ). Si así fuera se asigna acción B. Sino, se verifica lo siguiente. Si el jugador es menor de edad, se asigna acción C. Sino, si es un adulto joven se asigna D; si es adulto mayor, E; sino F.

El Diagrama de Actividades de este algoritmo:



12 Nodos
16 Aristas

CC = A – N + 2
CC = 16 – 12 + 2
CC = 6

6 Casos de Prueba

Caso de Prueba 1:

- Primera Letra no es A
- Segunda Letra no está al inicio del abecedario
- No es menor de edad
- Es adulto joven
- Dato de prueba: Johanna, 20
- Resultado esperado: D

Caso de Prueba 2:

- Primera Letra no es A
- Segunda Letra no está al inicio del abecedario
- No es menor de edad
- Es adulto mayor
- Dato de prueba: Ezequiel, 67
- Resultado esperado: E

Caso de Prueba 3:

- Primera Letra no es A
- Segunda Letra no está al inicio del abecedario
- No es menor de edad
- No es adulto mayor ni adulto joven
- Dato de prueba: Ezequiel, 44
- Resultado esperado: F

Caso de Prueba 4:

- Primera Letra no es A
- Segunda Letra no está al inicio del abecedario
- Es menor de edad
- Dato de prueba: Johanna, 17
- Resultado esperado: C

Caso de Prueba 5:

- Primera Letra no es A
- Segunda Letra está al inicio del abecedario
- Dato de prueba: Martin, 22
- Resultado esperado: B

Caso de Prueba 6:

- Primera Letra es A
- Dato de prueba: Alberto, 10
- Resultado esperado: A

Nro	Caso de prueba	Dato de prueba	Resultado esperado
1	Complejidad Ciclomática 1	Johanna, 20	D
2	Complejidad Ciclomática 2	Ezequiel, 67	E
3	Complejidad Ciclomática 3	Ezequiel, 44	F
4	Complejidad Ciclomática 4	Johanna, 17	C
5	Complejidad Ciclomática 5	Martin, 22	B
6	Complejidad Ciclomática 6	Alberto, 10	A

Condiciones de borde

- Se utiliza en el caso de campos que aceptan valores numéricos.
- Se deben identificar los intervalos válidos y los inválidos.
- Vamos a agregar a nuestro ejemplo un proceso de pre ejecución en la Asignación que define que si un jugador tiene menos de 6 años, se rechaza su participación.
- Cuando hablamos de menores de 6 años, en el código de programación utilizaremos “<6”. ¿Será menor? ¿O habrá que incluir igual?
- ¿Cuándo me dice que puedo vender bebida alcohólica a un mayor de 18? ¿Incluye el 18 o no? ¿Es mayor o mayor e igual?
- Estas dudas que muchas veces llegan al programador (y el programador tal vez resuelve como cree conveniente) llevan a marcar la importancia de ejecutar pruebas con este criterio.
- Vamos a agregar a nuestro ejemplo un proceso de pre ejecución en la Asignación que define que si un jugador tiene menos de 6 años, se rechaza su participación.
- Casos de prueba:
  - Menores de 6 años
  - Igual a 6 años
  - Mayores de 6 años

Nro	Caso de Prueba	Dato de Prueba	Resultado Esperado
1	Edad menor a 6 años	Juliana, 4	Mensaje de Error
2	Edad igual a 6 años	Irene, 6	A

Tipos y tamaños de datos

Basado en input de datos

Tipos de Datos

- Todo dato que deseamos probar tiene definido un tipo de dato específico, ya sea un número, un texto, etc.
- ¿Qué ocurre si en la interfaz de usuario se envían datos que no cumplen con ese tipo de dato?
- Debe probarse qué pasa cuando a un datos se le carga un valor con los siguientes tipos de datos (esta lista es solo como ejemplo y podrían detallarse más):
  - Número entero
  - Número decimal
  - Número negativo
  - Texto
  - Alfanumérico
  - Signos o símbolos

En nuestro ejemplo el Nombre es un texto por lo cual habría menos restricciones, ya que un número podrá ser procesado como texto… aunque tal vez debemos preguntarnos si queremos validarlo también. Si no lo validamos, pasará a nuestro algoritmo y nos dará una Acción.
Pero vamos a poner el foco en Edad, que requiere un número y si no fuese un entero fallará ya por el tipo de datos definido para los parámetros de nuestros métodos y para los atributos de los objetos.

Nro	Caso de Prueba	Dato de Prueba	Resultado Esperado
1	Edad tipo número decimal	Ariel, 14.4	Mensaje de Error
2	Edad tipo número negativo	Ariel, -9	Mensaje de Error
3	Edad tipo texto	Ariel, AFPAID	Mensaje de Error
4	Edad tipo alfanumérico	Ariel, AF2AIF	Mensaje de Error
5	Edad con símbolos	Martin, AF*ADF	Mensaje de Error

Para el caso de los campos de texto debe verificarse que el tamaño ingresado es el correcto.
Por ejemplo si tenemos un campo Nombre que acepta 20 caracteres, deberíamos verificar qué pasa si el usuario ingresa 21 caracteres.
Entonces deberíamos generar casos de prueba para cada una de las posibilidades:

- Nombre de tamaño 20
- Nombre de tamaño 21

En nuestro ejemplo si bien no hay limitaciones en cuanto al tamaño por definición de esa propiedad, el algoritmo estipula controlar en base a la primera y segunda letra del nombre, por lo cual sólo está preparado para aceptar nombres con por lo menos 2 letras.
¿Qué pasa si ponemos un nombre de 1 letra?
Allí tenemos un nuevo caso de prueba.

Nro	Caso de Prueba	Dato de Prueba	Resultado Esperado
1	Nombre de longitud 1	A, 8	Mensaje de Error

¿Y en código sin IU?

¿Se puede probar considerando estos criterios cuando hay código sin interfaz de usuario?
Lo importante es que nuestro test con esta técnica se haga sobre las clases o módulos que incluyen la validación
Si un método incluye validaciones del tipo de datos se puede probar (aunque atención que puede fallar antes si damos a una variable un valor con tipo de dato que esta variable no acepta, por eso en el ejemplo, definiremos el

INPUT de Edad y Nombre como string (de hecho, en muchos lenguajes el componente donde se ingresa el dato acepta sólo string)

Dominio de valores

Algunos campos tienen una cantidad acotada de valores posibles. Debemos asegurarnos que el usuario ingrese solo algunos de los valores allí válidos.
Por ejemplo, si las opciones son Tarjeta de Débito o Efectivo, no deberíamos permitir que ingrese Tarjeta de Crédito.
Y deberíamos generar los casos de prueba para verificar que eso no ocurra.
En general utilizamos lenguajes de programación que, en sus componentes, ya restringen esa posibilidad. Pero si así no fuera, debemos considerarlo.

Formato de valores

Algunos datos tienen un formato predefinido. Por ejemplo, podría ser el email.
Un correo electrónico debe tener un carácter @ (arroba) y un carácter . (punto) luego de la arroba
Casos de prueba posibles:

- Correo sin arroba
- Correo con arroba, pero sin punto luego de la arroba
- Correo con arroba y con punto, pero el punto está al final

Nro	Caso de Prueba	Dato de Prueba	Resultado Esperado
1	Correo sin arroba	pepe	Mensaje de Error
2	Correo con arroba, pero sin punto luego de la arroba	pepe@hotmailcom	Mensaje de Error
3	Correo con arroba y con punto, pero el punto está al final	pepe@hotmail.	Mensaje de Error

Datos obligatorios

Debe probarse que los datos obligatorios se encuentren realmente completados
Para cada uno de esos datos obligatorios debe generarse un caso de prueba con el dato vacío y otro caso de prueba con el dato con sólo espacios

Relación de datos

Una vez que se probaron cada uno de los datos debe verificarse la relación entre ellos.
Por ejemplo, si puede elegir Tipo de Pago y elije Tarjeta, entonces debe ingresar el número de la tarjeta
Del caso anterior se deriva un caso de prueba:

- Usuario selecciona el Tipo de pago = Tarjeta, pero no ingresa tarjeta

Considerar la persistencia

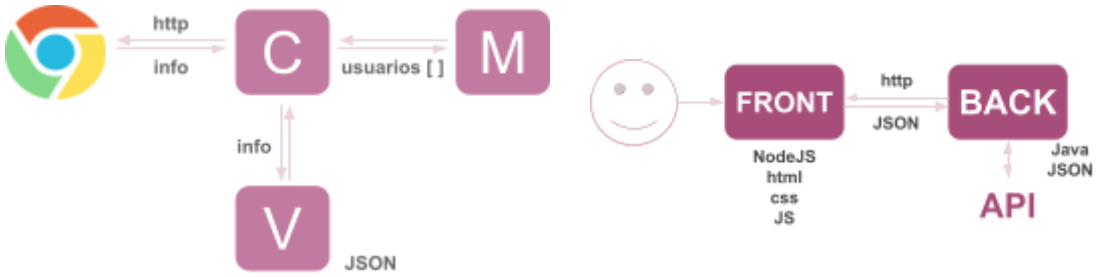
Si los datos son almacenados con alguna estrategia de persistencia (archivos, bases de datos) que manejan tipos de datos, tamaños, restricciones, etc. debemos considerarlo.
En nuestro ejemplo no nos preocupamos por el tamaño del campo nombre en cuanto a su longitud máxima.
¿Pero qué ocurre si ese nombre luego será guardado en una base de datos MySql que tiene definido un tipo de dato varchar(40)?
Debemos incorporar un caso de prueba para un Nombre con 41 caracteres

Selenium test de regresión: test para cosas que ya existen. La interfaz de usuario no cambia. ????dónde va esto???

MVC

Modelo: negocio.
Vista: html, css.
Controller: rutas web.

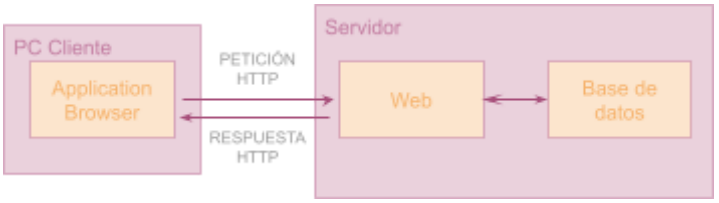
La DB está en un módulo aparte llamado Data Access Object (DAOS).



API: la posta

API (Application Programming Interface) es un conjunto de reglas y especificaciones que las aplicaciones pueden seguir para comunicarse entre ellas.
Permite hacer uso de funciones ya existentes en otro software.
Existen distintos tipos de APIs:

- APIs de Servicios Web
- APIs de Librerías
- APIs de SO



Perdón, pero ¿qué es HTTP?

HTTP: HyperText Transfer Protocol

Es el protocolo que usa un navegador para comunicarse con un servidor web. Este tiene las siguientes características:

- Esquema sencillo de petición-respuesta, yo pido algo al servidor y el servidor me lo devuelve.
- No guarda estados. Una petición HTTP no tiene memoria de lo que se hizo anteriormente (son peticiones independientes).
- Existen básicamente dos mensajes: uno de petición y otro de respuesta.
- Ambos mensajes tienen una línea inicial (que cambia según sea una petición o una respuesta), cabeceras y un cuerpo.
- En la línea inicial puede haber métodos de petición o un código de respuesta.
- En las cabeceras hay metadatos que son modificables/flexibles.
- El cuerpo es opcional y depende si en la respuesta se devuelve algo, o si en la petición se envían parámetros.

Peticiones HTTP

Para generar una petición HTTP se necesitan:

- Un **método de petición HTTP**
- Una URL (dirección web) conocido como recurso, al cual le envío la petición
- Cabeceras (generalmente son autogeneradas aunque se pueden modificar e ser necesario)
- Un cuerpo de ser necesario

Métodos de petición HTTP

- GET: sirve para recuperar/solicitar un recurso especificado. Esta petición sólo debe recuperar datos y no tener ningún otro efecto
- HEAD: idéntico a GET pero no traer el recurso en sí, el cuerpo viene vacío, solo trae las cabeceras (los metadatos del recurso)

- POST: envía datos para que sean procesador por el recurso especificado. Datos se incluyen en el cuerpo. Necesita de una cabecera llamada *Content-Type* que especifica que estoy mandando en el body. Puedo estar mandando JSON, valores cargados de un formulario, texto plano, archivos…
- PUT: orientado a la modificación de recursos. Por ende como parámetros envío los datos actualizados que quiero que se apliquen al recurso especificado. Actualiza/sobrescribe todo el recurso
- PATCH: similar a PUT pero actualiza parcialmente el recurso
- DELETE: borra el recurso especificado

Existen otros métodos pero estos son los más interesantes/comunes/usados.

Estos métodos representan, básicamente, a un ABM/CRUD

Operation	SQL	HTTP	DDS
Create	INSERT	PUT/POST	write
Read (Retrieve)	SELECT	GET	read/take
Update (Modify)	UPDATE	PUT/PATCH	write
Delete (Destroy)	DELETE	DELETE	dispose

Códigos de estado HTTP

Los códigos de estado se dividen por secciones en base al número:

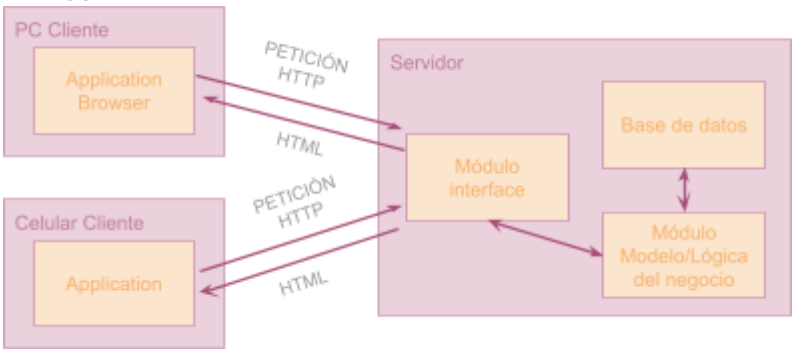
- **1xx**: respuesta informativa, la petición fue recibida y entendida
- **2xx**: petición correcta, la petición fue recibida, entendida y aceptada
- **3xx**: redirección
- **4xx**: errores del cliente
- **5xx**: errores del servidor

Dentro de estas secciones existen códigos que representan situaciones más específicas. Los más interesantes/usados:

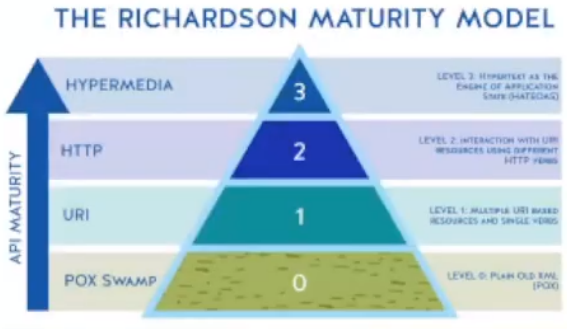
- 200: OK
- 201: recurso creado
- 202: solicitud aceptada y en procesamiento
- 204: solicitud procesada correctamente pero servidor no retorna contenido
- 307: recurso movido temporalmente
- 308: recurso movido permanentemente
- 400: petición no se pudo procesar debido a error del cliente (ej; parámetros mal definidos)
- 401: no autorización. El cliente debe autenticarse primero antes de realizar esa petición
- 403: petición prohibida. El cliente se autentico correctamente pero no tiene los permisos para realizar esta petición
- 404: recurso no encontrado
- 405: método HTTP no soportado para el recurso solicitado
- 500: error interno del servidor (explotó todo)
- 503: servicio no disponible debido a que se encuentra caído o en mantenimiento (temporal)

SI LO VALIDO EN EL FRONT LO PUEDO SALTEAR

2do approach de solución



El protocolo HTTP existe pero la idea de usar distintos métodos, recursos, códigos de respuesta es un estándar. Al ser un estándar hay APIs que lo respetan y otras APIs que no respetan nada.
Respetar este estándar es lo que genera la diferencia entre una API Rest y el resto de las APIs.
Ahora puede que mi API respete el estándar parcialmente, por ende se creó la idea de nivel de madurez de una API, el cual define que una API es más madura cuanto más respete al estándar.
Existen 4 niveles definidos según el modelo de Richardson Maturity:



API: NIVEL 0
- Usa HTTP como protocolo de comunicación
- Peticiones no almacenan estados

API: NIVEL 1
- Usa HTTP como protocolo de comunicación
- Peticiones no almacenan estados
- Introduce la idea de existencia de distintos recursos y cada recurso tiene su propia URL de acceso

Ejemplo:
- Si quiero acceder a todos los posts mi URL tendría que ser: https://www.misitio.com/api/posts
- Si quiero acceder a todos los usuarios registrados mi URL tendría que ser: https://www.misitio.com/api/usuarios
- Ahora, si quiero acceder a un post especifico debo saber su ID (por ej 324) y la URL tendría que ser: https://www.misitio.com/api/posts/324

API: NIVEL 2
- Usa HTTP como protocolo de comunicación
- Peticiones no almacenan estados
- Introduce la idea de existencia de distintos recursos y cada recurso tiene su propia URL de acceso
- Puedo ejecutar operaciones CRUD sobre los distintos recursos respetando los métodos HTTP

API: NIVEL 3
- Usa HTTP como protocolo de comunicación
- Peticiones no almacenan estados
- Introduce la idea de existencia de distintos recursos y cada recurso tiene su propia URL de acceso
- Puedo ejecutar operaciones CRUD sobre los distintos recursos respetando los métodos HTTP
- Puedo descubrir nuevos servicios al consultar recursos como si estuviera navegando por una página web

Ejemplo:
- Género una petición de búsqueda de los últimos 25 posts de un usuario
- El servidor me devuelve los últimos 25 posts del usuario en formato JSON
- Además me devuelve una URL ya armada para generar la petición de buscar los 25 posts siguientes de ese usuario

Generalmente cuando uno solicita una colección de recursos a una API, la cantidad está limitada a modo de no saturar el sistema. Por ende la API te ofrece esa colección de manera paginada dandote una cantidad finita de objetos de esa colección, pero teniendo acceso a la primera, anterior, siguiente o última página, similar a lo que hace Google con las búsquedas.

CRUD = ABMC

Create Read Update Delete
Alta Baja Modificación Consulta

https://github.com/jonybuzz/demo-spring-web

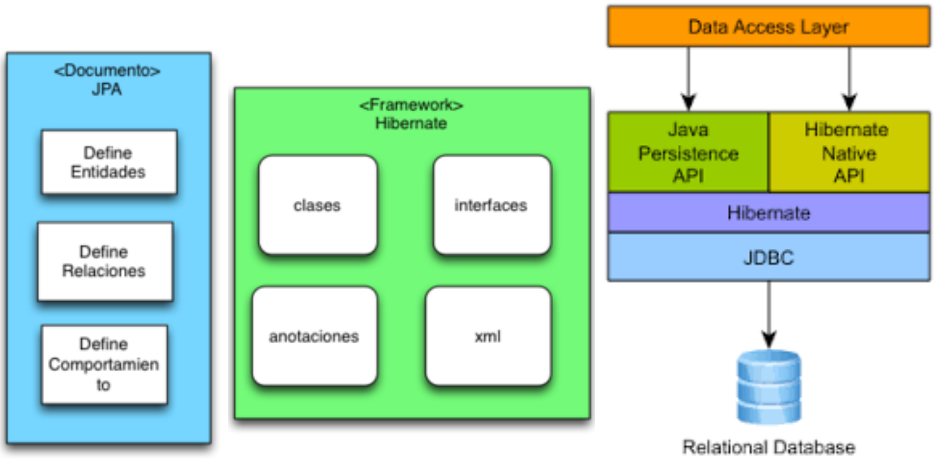
ORM Tecnología

Hibernate es solo un medio para persistir. Es una herramienta.
API es una interfaz, no importa cómo.

Qué es qué

JPA: Es la especificación de Java (API) para acceder a la persistencia de objetos
Hibernate: Es la implementación de JPA más usada. Hay otras como Eclipselink o Toplink…
JDBC: Es el conector entre una aplicación Java y un motor de BD particular (MySQL, Oracle, SqlServer, etc).

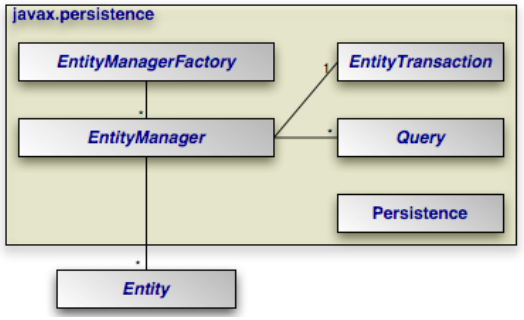
Hibernate genera las queries por nosotros y se comunica con el driver JDBC (una librería), que recibe queries como String y las traslada a la BD.



Diseño de JPA

- Entidades: Son los objetos de nuestro negocio que se persisten.
- EntityManager: Interfaz que gestiona la persistencia de entidades (objetos) en el contexto.
- EntityManagerFactory: Crea EntityManagers.
- Persistence: Esta clase contiene métodos para obtener un EntityManagerFactory.
- Query: Esta interfaz se usa para obtener objetos que cumplan ciertos criterios más complejos.

Transacción: conjunto de operaciones … se va a ejecutar uno o ninguno. 8:56 completar



persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
```

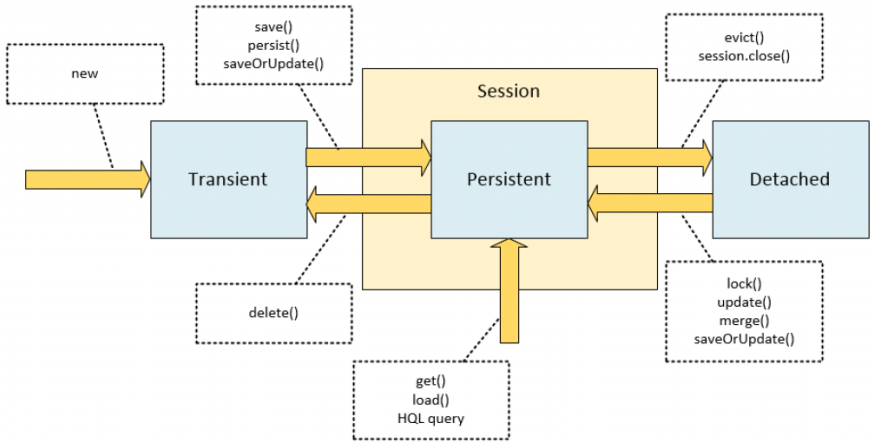
```
version="2.0">
 <persistence-unit name="UnidadPersonas">
  <class>com.example.dominio.Persona</class>

  <properties>
   <property name= "hibernate.show_sql" value="true" />
   <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
   <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
   <property name="javax.persistence.jdbc.user" value="root" />
   <property name="javax.persistence.jdbc.password" value="mypass" />
   <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost/jpa" />
  </properties>

 </persistence-unit>
</persistence>
```

HIBERNATE: Estados de una entidad

Las **entidades son clases POJO** (sólo atributos con getters y setters). Al crear una instancia, Hibernate no la conoce y se dice que está en estado Transient. Cuando la **persistimos** o la **obtenemos** con el EntityManager, pasa a estar **administrada por Hibernate**.

¿Qué significa administrada?

Que **mantiene una relación entre el objeto en memoria y el registro** en la BD. Cada vez que lo modifiquemos se va a disparar una query.



Ejemplo: Proyecto

https://github.com/jonybuzz/demo-hibernate

- Agregar en pom.xml las dependencias de hibernate y el Driver
- Configurar conexión en persistence.xml
- Agregar anotaciones de JPA en las entidades

pom.xml

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.25</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>5.4.18.Final</version>
```

```
</dependency>
```

Ejemplo: Entidad

@Entity: Obligatorio
@Table: Opcional
@Id @GeneratedValue: Obligatorio. Indica cuál es la PK y que debe autogenerarse cuando insertamos
@ManyToOne: Indica que la relación es muchos a uno respecto de Curso. Puede ir acompañado por @JoinColumn para indicar la FK
@ManyToMany: Indica que la relación es muchos. Puede ir con @JoinTable para especificar la tabla intermedia.
@OneToMany: Indica que la relación es uno a muchos respecto de Curso. Debe haber un ManyToOne del otro lado.
Profesor y Alumno también deben ser entidades.

```
@Entity
@Table(name = "curso")
public class Curso {

   @Id
   @GeneratedValue
   private Long codigo;

   private String nombre;

   @ManyToOne
   @JoinColumn(name = "id_profesor")
   private Profesor profesor;

   @OneToMany(mappedBy = "curso")
   private List<Alumno> alumnos;
   // getters, setters
}
```

Ejemplo: Ejecución

EntityManagerFactory: Debe crearse una sola vez durante la vida de la aplicación.
EntityManager: Puede crearse una sola vez o cada vez que realizo una acción (request, schedule)
EntityTransaction: Debe abrirse una transacción cada vez que se quieren ejecutar un conjunto de operaciones atómicas contra la BD. La transacción puede confirmarse (commit) o revertirse (rollback).

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("demo-hibernate-PU");
EntityManager em = factory.createEntityManager();
EntityTransaction tx = em.getTransaction();

tx.begin();
tx.commit();
tx.rollback();
```

Links

- ¿Qué es JPA?: http://oraclejuniors.blogspot.com/2014/11/que-es-jpa-java-persistence-api.html
- Interfaces principales de JPA: https://www.tutorialspoint.com/es/jpa/jpa_architecture.htm
- Hibernate: https://www.baeldung.com/hibernate-save-persist-update-merge-saveorupdate
- Más sobre relaciones OneToMany y ManyToOne:
- https://thoughts-on-java.org/best-practices-many-one-one-many-associations-mappings/
- Proyecto de ejemplo: https://github.com/jonybuzz/demo-hibernate
- Herencia: https://www.baeldung.com/hibernate-inheritance

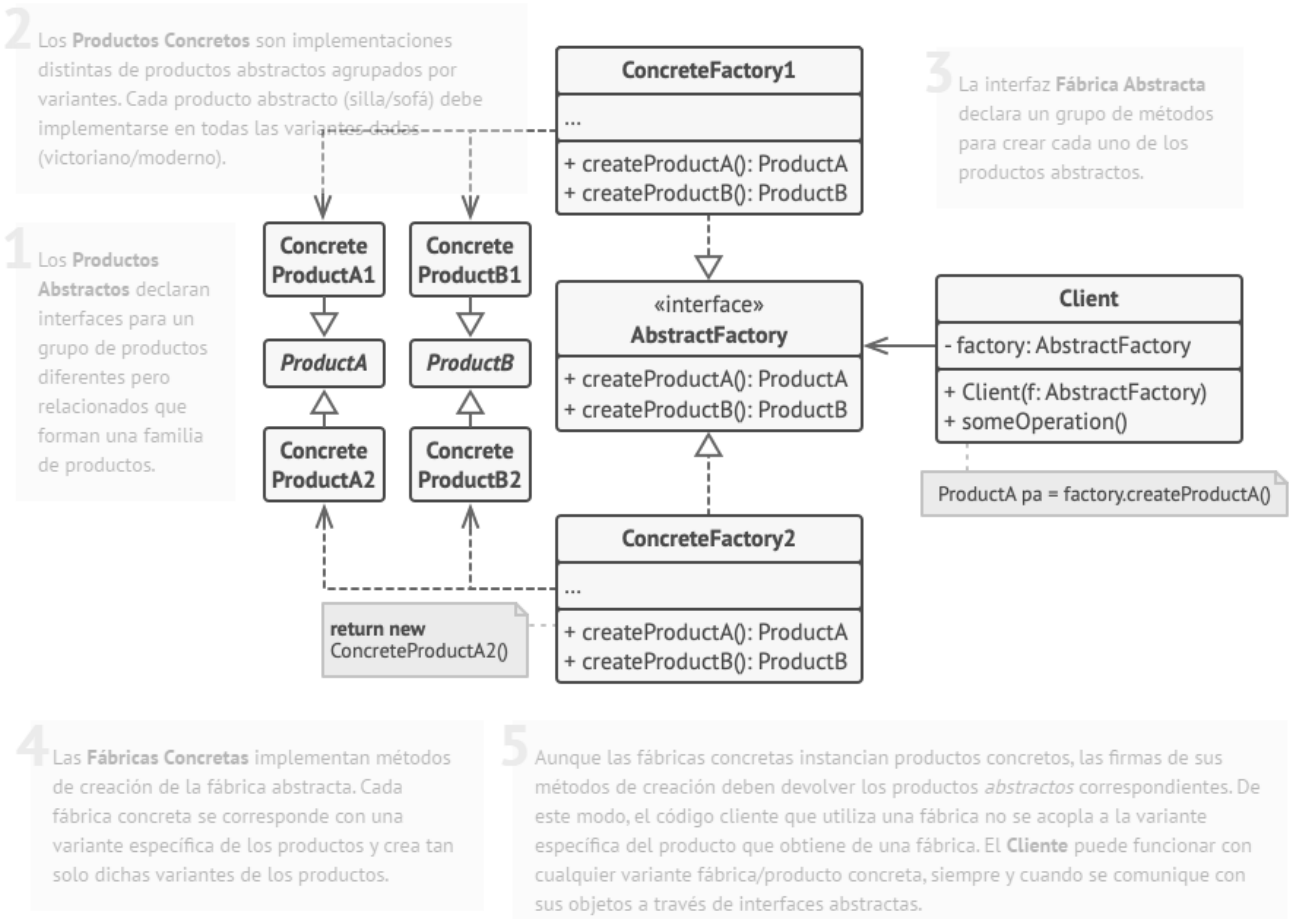
PATRÓN FACTORY METHOD (creacional)

Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

Cómo implementarlo

1. Haz que todos los productos sigan la misma interfaz. Esta interfaz deberá declarar métodos que tengan sentido en todos los productos.
2. Añade un patrón Factory Method vacío dentro de la clase creadora. El tipo de retorno del método deberá coincidir con la interfaz común de los productos.
3. Encuentra todas las referencias a constructores de producto en el código de la clase creadora. Una a una, sustitúyelas por invocaciones al Factory Method, mientras extraes el código de creación de productos para colocarlo dentro del Factory Method. Puede ser que tengas que añadir un parámetro temporal al Factory Method para controlar el tipo de producto devuelto.
4. Ahora, crea un grupo de subclases creadoras para cada tipo de producto enumerado en el Factory Method. Sobrescribe el Factory Method en las subclases y extrae las partes adecuadas del código constructor del método base.
5. Si hay demasiados tipos de producto y no tiene sentido crear subclases para todos ellos, puedes reutilizar el parámetro de control de la clase base en las subclases.
   Si, tras todas las extracciones, el Factory Method base queda vacío, puedes hacerlo abstracto. Si queda algo dentro, puedes convertirlo en un comportamiento por defecto del método.

Utiliza el Método Fábrica cuando no conozcas de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar tu código.
Utiliza el Factory Method cuando quieras ahorrar recursos del sistema mediante la reutilización de objetos existentes en lugar de reconstruirlos cada vez.



https://refactoring.guru/es/design-patterns/factory-method

ABSTRACT FACTORY (creacional)

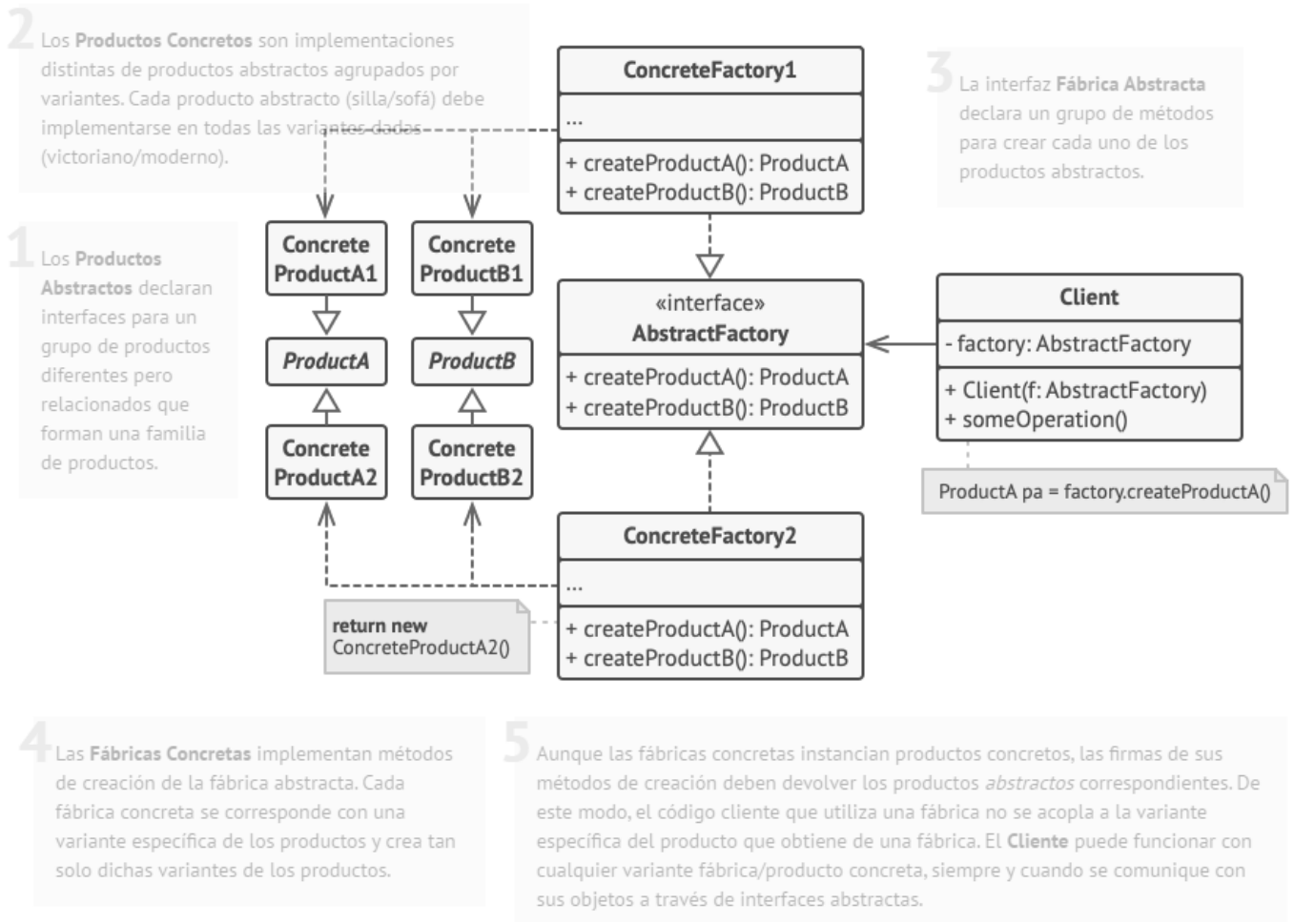
Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.

Generalmente se implementa utilizando Factory Method y por tanto provee al menos toda la flexibilidad de éste. La diferencia principal entre ambos es que el primero trata con familias de productos, mientras que el otro se preocupa por un único producto.

Cómo implementarlo

1. Mapea una matriz de distintos tipos de productos frente a variantes de dichos productos.
2. Declara interfaces abstractas de producto para todos los tipos de productos. Después haz que todas las clases concretas de productos implementen esas interfaces.
3. Declara la interfaz de la fábrica abstracta con un grupo de métodos de creación para todos los productos abstractos.
4. Implementa un grupo de clases concretas de fábrica, una por cada variante de producto.
5. Crea un código de inicialización de la fábrica en algún punto de la aplicación. Deberá instanciar una de las clases concretas de la fábrica, dependiendo de la configuración de la aplicación o del entorno actual. Pasa este objeto de fábrica a todas las clases que construyen productos.
6. Explora el código y encuentra todas las llamadas directas a constructores de producto. Sustitúyelas por llamadas al método de creación adecuado dentro del objeto de fábrica.

Utiliza el patrón Abstract Factory cuando tu código deba funcionar con varias familias de productos relacionados, pero no desees que dependa de las clases concretas de esos productos, ya que puede ser que no los conozcas de antemano o sencillamente quieras permitir una futura extensibilidad.



https://refactoring.guru/es/design-patterns/abstract-factory

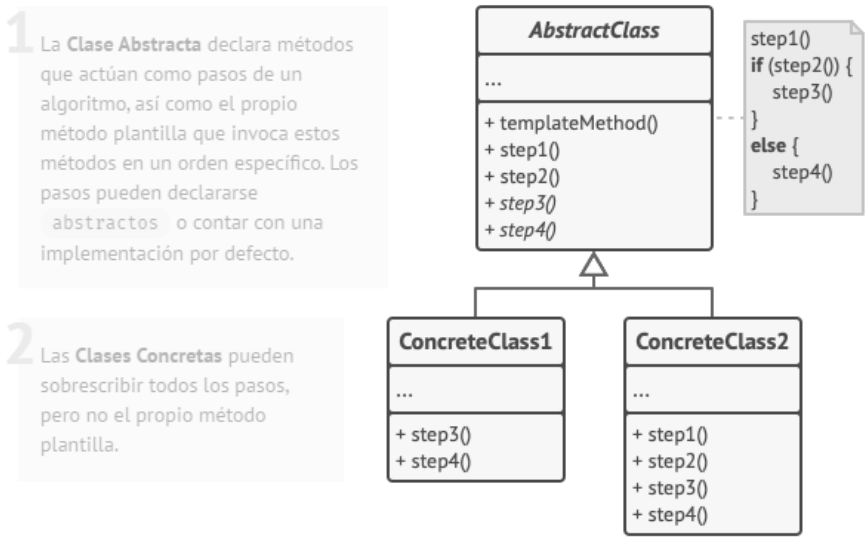
PATRÓN TEMPLATE METHOD (de comportamiento)

Permite definir el esqueleto de un algoritmo en una operación, aunque difieren algunos pasos en las subclases.
Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.

Utiliza el patrón Template Method cuando quieras permitir a tus clientes que extiendan únicamente pasos particulares de un algoritmo, pero no todo el algoritmo o su estructura.
Utiliza el patrón cuando tengas muchas clases que contengan algoritmos casi idénticos, pero con algunas diferencias mínimas. Como resultado, puede que tengas que modificar todas las clases cuando el algoritmo cambie.

Cómo implementarlo

1. Analiza el algoritmo objetivo para ver si puedes dividirlo en pasos. Considera qué pasos son comunes a todas las subclases y cuáles siempre serán únicos.
2. Crea la clase base abstracta y declara el método plantilla y un grupo de métodos abstractos que representan los pasos del algoritmo. Perfila la estructura del algoritmo en el método plantilla ejecutando los pasos correspondientes. Considera declarar el método plantilla como final para evitar que las subclases lo sobrescriban.
3. No hay problema en que todos los pasos acaben siendo abstractos. Sin embargo, a algunos pasos les vendría bien tener una implementación por defecto. Las subclases no tienen que implementar estos métodos.
4. Piensa en añadir ganchos entre los pasos cruciales del algoritmo.
5. Para cada variación del algoritmo, crea una nueva subclase concreta. Ésta *debe* implementar todos los pasos abstractos, pero también *puede* sobrescribir algunos de los opcionales.



https://refactoring.guru/es/design-patterns/template-method

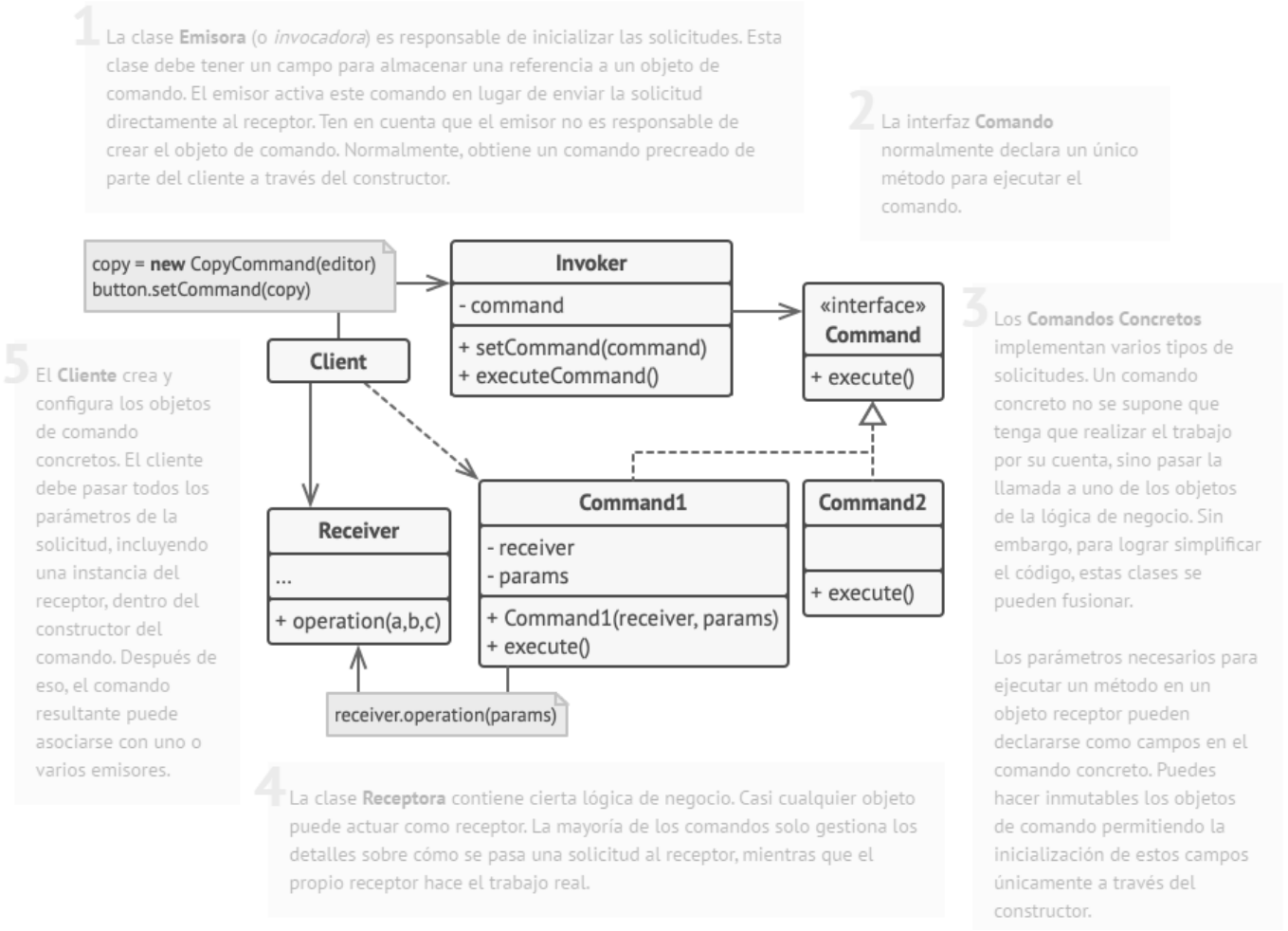
PATRÓN COMMAND (de comportamiento)

Ofrece una interfaz común que permite invocar las acciones de forma uniforme y extender el sistema con nuevas acciones de forma más sencilla.
Permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma
Command es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.

Utiliza el patrón Command cuando quieras parametrizar objetos con operaciones.
Utiliza el patrón Command cuando quieras poner operaciones en cola, programar su ejecución, o ejecutarlas de forma remota.
Utiliza el patrón Command cuando quieras implementar operaciones reversibles.

Cómo implementarlo

1. Declara la interfaz de comando con un único método de ejecución.
2. Empieza extrayendo solicitudes y poniéndolas dentro de clases concretas de comando que implementen la interfaz de comando. Cada clase debe contar con un grupo de campos para almacenar los argumentos de las solicitudes junto con referencias al objeto receptor. Todos estos valores deben inicializarse a través del constructor del comando.
3. Identifica clases que actúen como emisoras. Añade los campos para almacenar comandos dentro de estas clases. Las emisoras deberán comunicarse con sus comandos tan solo a través de la interfaz de comando. Normalmente las emisoras no crean objetos de comando por su cuenta, sino que los obtienen del código cliente.
4. Cambia las emisoras de forma que ejecuten el comando en lugar de enviar directamente una solicitud al receptor.
5. El cliente debe inicializar objetos en el siguiente orden:
   a. Crear receptores.
   b. Crear comandos y asociarlos con receptores si es necesario.
   c. Crear emisores y asociarlos con comandos específicos.



https://refactoring.guru/es/design-patterns/command

TECNOLOGÍAS

SEMINARIO DISEÑO Y MAQUETADO WEB

HTML + CSS

¿Qué es HTML?

Es un lenguaje de marcado de texto (y más...) que interpreta el navegador y nos permite visualizar un documento estructurado y dividido en secciones. HTML define el contenido de las páginas web. Se trata de un conjunto de

etiquetas que sirven para definir el texto y otros elementos que compondrán una página web, como imágenes, listas, tablas, etc. No es un lenguaje de programación ya que no permite desarrollar procesos lógicos.

Etiquetas, atributos y elementos

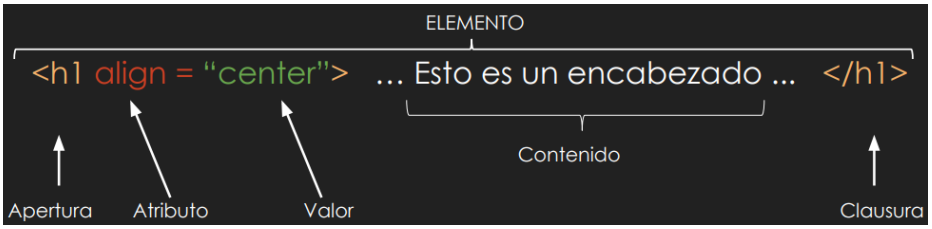
- Etiqueta: son la parte del código que permite generar un elemento visual en el buscador. Sirven como "Esqueleto" de nuestra página web.

Un documento HTML está formado por: ETIQUETAS y ATRIBUTOS que en conjunto conforman ELEMENTO.

- `<h1>` etiqueta de apertura
- `</h1>` etiqueta de clausura
- Atributo: característica que deseamos modificar de una etiqueta.
- Valor: definición de la característica que vamos a modificar.

HTML - HyperText Markup Language.

Sintaxis de un elemento



Cómo crear un documento .html

Un documento html es simplemente un documento de texto con extensión '.html'. Para comenzar a desarrollar nuestra página web solamente necesitamos crear nuestro archivo y abrirlo con un editor de texto.

Editores de texto:

- Atom https://atom.io/
- Sublime text https://www.sublimetext.com/
- Visual Studio Code https://code.visualstudio.com/

ESTRUCTURA BÁSICA DE UN .html

The HTML Element

El elemento es un contenedor de 'metadata' y se ubica entre el tag y el tag . La metadata de HTML son datos que utilizaremos dentro de nuestro documento. Estos datos no son visibles en nuestra página web.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset = "uff-8">
        <title> Título de mi página web </title>
    </head>

    <body>

            Hola mundo!

    </body>
</html>
```

El charset permite definir la codificación de caracteres a utilizar. Se utiliza para evitar la visualización incorrecta en algunos navegadores.

Tags HTML

- *Títulos*

```
<h1> Título Principal </h1>

<h2> Subtítulo </h2>
<h3> Otro subtítulo </h3>
<h4> ... </h4>
<h5> ... </h5>
<h6> ... </h6>
```

Nota: Este tag es útil para el posicionamiento en buscadores. El H1 debería utilizarse una sola vez en la página.

- *Párrafos*

```
<p>
    Lorem ipsum dolor sit amet, consectetur
    adipisicing elit, sed do eiusmod tempor
    incididunt ut labore et dolore magna aliqua.
</p>
```

- *Listas Ordenadas*

```
<ol>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ol>
```

Tag para listar ítems de manera consecutiva y numerada. En el navegador se vería:

1. Item 1
2. Item 2
3. Item 3

- *Listas desordenadas*

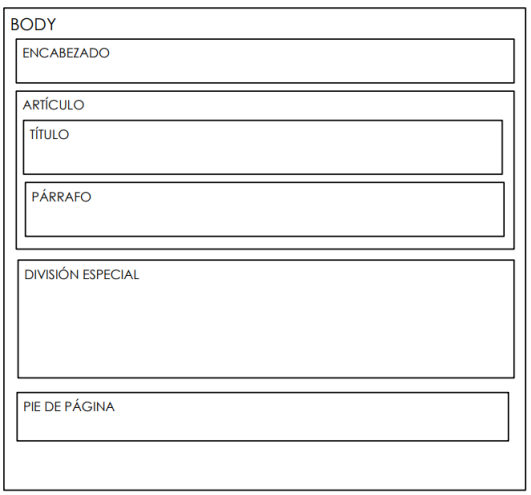
```
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>
```

Tag para listar ítems de manera consecutiva, sin numerar. En el navegador se vería:

- Item 1
- Item 2
- Item 3

- *Tags para división del html*

```
<body>...</body>
<header> .... </header>
<footer> ... </footer>
<article>...</article>
<div>...</div>
```



- *Otros tags útiles*

```
<table>...</table>
<nav>...</nav>
<form>...</form>
<br>
```

https://www.w3schools.com/TAGS/default.ASP

CSS Cascading Style Sheets

Las hojas de estilo sirven para "estilizar" nuestro contenido HTML. Con CSS podemos cambiar colores, fondos, tipografías, anchos, altos, posiciones, etc.

.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset = "utf-8">
        <title> Título de mi página web </title>
    </head>
    <body>
        <h1> Mi nombre es Rodrigo</h1>
        <p>
            Estoy haciendo mi primer página web con estilos
        </p>
    </body>
</html>
```

.css

```
body{
  background-color; black;
}

h1{
 color:blue;
 font-size:28px;
 text-decoration:underline;
 text-align: center;
}

p{
 color:white;
}
```

¿Cómo relacionar el HTML con el CSS?

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset = "utf-8">
        <link rel="stylesheet" href="estilos.css">
        <title> Título de mi página web </title>
    </head>
    <body>
        <h1> Mi nombre es Rodrigo</h1>
        <p> Estoy haciendo mi primer página  web con estilos </p>
    </body>
</html>
```

Atributos

- *Tamaño de fuente*
  font-size: 22px;
- *Alineación del texto*
  text-align: center/left/right.
- *Color de fuente*
  color: red;
- *Color de fondo*
  background-color: rgb(142,98,132);
- *Subrayado*
  text-decoration: underline;
- *Ancho y alto*
  width: 500px;
  height: 700px;
- *Márgenes exteriores*
  margin: 10px;
  margin-top: 40px;
- *Márgenes Interiores*
  padding:5px;
- *Bordes*
  border: 1px solid rgb(142, 98,132);
  border-radius: 5px;

Selectores

Clases: Una clase es la definición de un estilo que, en principio, no está asociado a ninguna etiqueta HTML pero que podemos asociar a etiquetas concretas.

```
.titulo{
    color: green;
    font-size: 60px;
}
```

Desde nuestra hoja de estilos, definimos las propiedades de estilo que nuestra clase tendrá. Se definen comenzando con un Punto.
Una vez definida la clase y sus propiedades de estilos, debemos informarle a QUÉ elemento le aplicamos dicha clase.

```
<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet" href="estilos.css">
        <meta charset = "utf-8">
        <title> Título de mi página web </title>
    </head>
    <body>
        <h1 class = "titulo"> Mi nombre es Rodrigo
</h1>
    </body>
</html>
```

.CSS .Selector{ atributo:valor; }
.html class = "nombreClase"
.class .Selector{ atributo: valor;}
#id #Selector{ atributo:valor;}
tag Selector { atributo:valor;}
.selector elemento{ atributo: valor;}
Nota: No puede haber Ids repetidos.

¿Qué es el atributo display?

Todos los elementos de nuestro HTML tienen el atributo "display" que define cómo se muestran y cómo se comportan respecto a otros elementos. Si desde nuestra hoja de estilos, le asignamos a un elemento el atributo "display:none;", dicho elemento no será visible en nuestra página.

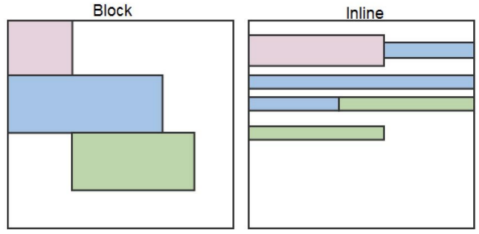
display:block; VS display:inline;

DISPLAY: BLOCK; Define un elemento con comportamiento de bloque. Recibe las propiedades del modelo de caja.

```
<div></div>
<header></header>
<h1></h1>
<ul></ul>
```

DISPLAY:INLINE; Define un elemento con comportamiento en línea. No recibe algunas propiedades del modelo de caja.

```
<a></a>
 <img>
<button> </button>
```



Nota: Desde nuestra hoja de estilos (css) podemos modificar el "display" de los diferentes elementos.

JavaScript

JavaScript es un lenguaje de programación interpretado (no se compila). No tiene nada que ver con Java (sólo parte de la sintaxis). La motivación inicial del lenguaje fue poder darle cierto dinamismo a los documentos HTML. Todos los navegadores modernos interpretan JS. Hoy en día no se limita sólo al uso del lado del cliente.
Entre otras cosas, con JS se puede manipular el documento. Acceder a elementos del documento HTML. Modificar atributos de elementos. Crear elementos nuevos o eliminar elementos existentes. Realizar requests al servidor.

¿Para qué nos sirve JavaScript?

- Juegos.
- Mapas.
- Galerías.
- Buscadores.
- Formularios.

- etc...

¿Cómo inyectar JS en nuestro documento HTML?

Mediante un archivo externo:

```
<head>
    <script src="miArchivo.js"></script>


</head>
```

Mediante código inline:

```
<head>
    <script>
        function miFuncion() {

        }
    </script>


</head>
```

Sintaxis

Tipos base:

- String ---> “Hoy es sábado”
- Number ---> 10.4
- Boolean ---> true/false
- Object ---> {}
- Array---> []
- Null - Undefined

If

```
if (condición) {
    //Hacer esto si se cumple la condición
}
```

Switch

```
switch(color) {
    case “Rojo”:
        console.log(“Usted eligió color Rojo”);
        break;
    case “Azul”:
        console.log(“Usted eligió color azul”);
        break;
    default:
        console.log(“Es otro color”);
```

For

```
for (inicio; condicion; incremento){
    //Hacer esto mientras la condicion sea verdadera
}
for(var i = 0; i < 10; i++;){
    console.log(“El contador vale “ + i);
}
```

While

```
while (condicion){
    // Realizar mientras sea verdadera la cond.
}
```

Arrays

```
var profesores = ["Alejandro", "Ezequiel", "Rodrigo","Lucas"];
                  0            1           2         3
```

Operaciones con arrays:

- Añadir un elemento al final del array:
  array.push("elemento")
- Añadir un elemento al comienzo del array:
  array.unshift("elemento")
- Eliminar el último elemento de un array:
  array.pop()
- Eliminar el primer elemento de un array:
  array.shift()

Objetos

- Un objeto es una colección de propiedades.
- Una propiedad es una asociación entre un nombre (una clave) y un valor.
- El valor de una propiedad puede ser una función, en cuyo caso se denomina "Método".

```
var auto = {
    marca : 'Chevrolet',
    modelo : 'Corsa',
    kilometraje : '120000',
    color : 'Blanco'
};
```

Podemos acceder a una propiedad de un objeto de varias formas:
auto.marca ---> "Chevrolet"
auto["marca"] ---> "Chevrolet

Funciones anónimas

Las funciones anónimas son funciones que al momento de declararlas no les asignamos ningún nombre.

```
var saludo = function () {
        // Código;
}
```

Podemos asignar estas funciones a una variable. Para ejecutar dicha función, debemos hacerlo llamándola por el nombre de la variable.

Funciones anónimas autoinvocadas

Estas funciones, una vez declaradas se llaman a sí mismas, ejecutando el código en ellas.

```
(function (){
        //código;
})();
```

Una de las razones por las que son útiles es porque permiten ejecutar código encapsulando variables al entorno de la función ahorrando posibles errores.

DOM Document Object Model

Es un modelo que representa en forma de árbol la estructura de un documento HTML. Mediante este modelo se puede "navegar" a través del documento para manipular los elementos.

DOM API

Para manipular los nodos se utiliza la DOM API
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model
A través de esta API se puede acceder a los nodos, iterarlos, modificar atributos, crear nuevos elementos, etc. Algunos métodos y propiedades útiles se detallan a continuación.

Métodos para seleccionar

- document.getElementById(<id>);
- document.querySelectorAll(<selector>);

Métodos para trabajar con Elementos

- document.createElement(<tag>);
- <element>.setAttribute(<att>, <value>);
- <element>.getAttribute(<att>, <value>);
- <element>.removeAttribute(<att>);
- <element>.classList
- <element>.innerHTML = <value>
- <element>.appendChild(otro element)

Event listeners

Podemos utilizar javascript para que realice ciertas funciones respondiendo a eventos determinados. Algunos de los eventos más utilizados son:

- onclick
- onchange
- onload
- onkeyup
- onkeydown
- onkeypress

¿Cómo usamos estos eventos? Una de las formas es definir un atributo en nuestro elemento html que indica a qué evento queremos que javascript responda y que función queremos que ejecute.

```
<button type="button" name="button" onclick="cargarDatosEnTabla()">
  Cargar datos
</button>
```

PATRONES ARQUITECTÓNICOS

Atributo de calidad: cualidad genérica. Cualidad que se espera de algo. La seguridad es un atributo de calidad. Ay esto está en el resumen del 1er parcial.
El requerimiento no va tanto por la usabilidad. comprobable, medible, implementable. Se resuelve a través de la estructura del sw.
¿Cómo hago desde el diseño para darle respuesta al atributo y requerimiento?
Defino un requerimiento para darle estructura al atributo.
Cumplimiento de los requerimientos es lo que da calidad.

El diagrama de componentes y despliegue acompaña todo esto.
¿Por qué elegir esta opción? ¿Atributos de calidad para compararlos, en qué arquitectura falla?

CATEGORIZACIÓN DE PATRONES

Patrones de Componentes y Subsistemas

Los siguientes Patrones podemos considerarlos los de más alto nivel y proponen soluciones arquitectónicas a nivel Sistema. Se debe comprender que las soluciones presentadas utilizan como estrategia componentes vinculados entre sí. El Componente es una estructura de software que cumple una función específica

- Arquitectura en Capas
- Arquitectura en 5 Capas
- Microkernel
- Arquitectura de Canal
- Patrón de Contenedor Recursivo
- Patrón de Control Jerárquico
- Máquina Virtual
- Arquitectura basada en Componentes
- Patrón de Sistemas de Tiempo Real Orientado a Objetos

Patrones de Distribución

- Memoria Compartida
- Llamada a Procedimiento Remoto
- Patrón Observador
- Bus de Datos
- Proxy
- Broker

Patrones de Concurrencia

Los siguientes Patrones se centran en la concurrencia. Incluye el control y asignación de los elementos arquitectónicos. Esto incluye la gestión de recursos que deben estar protegidos del acceso simultáneo.

- Patrón de Cola de Mensajes
- Patrón de Interrupción
- Patrón de Llamada Protegida
- Patrón RendezVous
- Patrón de Ejecución Cíclica
- Patrón Round Robin
- Patrón de Prioridad Estática
- Patrón de Prioridad Dinámica

Patrones de Memoria

Los siguientes Patrones se centran en el manejo eficiente de memoria y la gestión robusta de elementos de software. Estos Patrones son de vital importancia en Sistemas de Tiempo Real y Sistemas Embebidos donde el recurso memoria es finito

- Patrón de Asignación Estática
- Patrón de Asignación de Conjuntos
- Patrón de Buffer de Tamaño Fijo
- Patrón de Puntero Inteligente
- Patrón de “Recolección de Basura”
- Patrón de “Compactador de Basura”

Patrones de Recursos

Los siguientes Patrones se centran en el manejo eficiente de recursos Hardware. Estos Patrones tienen vínculo directo con los Patrones de Concurrencia y también con los Patrones de Memoria, si bien ahora nos importa el recurso específico y no el manejo que hace ese recurso de memoria

- Patrón de Sección Crítica
- Patrón de Prioridad por Herencia
- Patrón de Prioridad más Alta (por Herencia)
- Patrón de Prioridad más Alta Limitada (por Herencia)
- Patrón de Bloqueo Simultáneo
- Patrón de Bloqueo Ordenado

Patrones de Seguridad y Confiabilidad

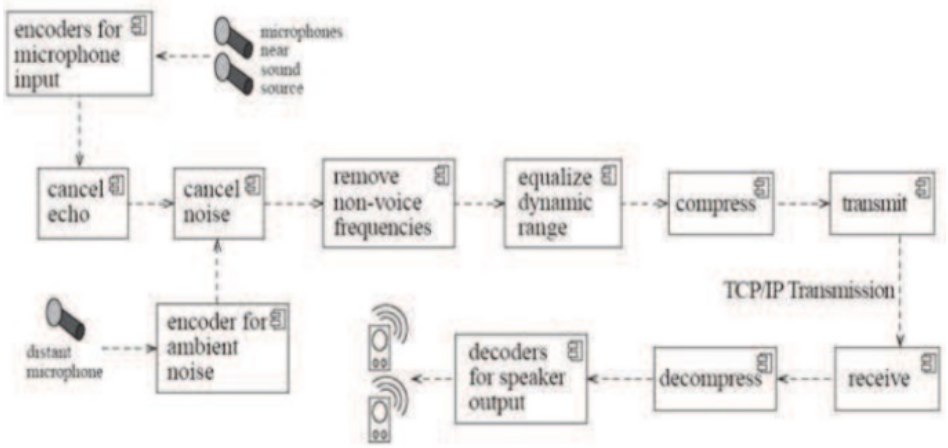
Los siguientes Patrones se centran en el manejo de la seguridad y la confiabilidad del Sistema. La seguridad puede definirse como la ausencia de accidentes o pérdidas, mientras que la fiabilidad se refiere a la probabilidad que un sistema continuará funcionando durante un período de tiempo especificado. Un sistema seguro puede fallar frecuentemente siempre que no cause accidentes, mientras que la fiabilidad de un sistema no se refiere en absoluto a las consecuencias que produce en caso de fallo. Sin embargo, la seguridad y la fiabilidad tienen un aspecto importante en común: su manejo requiere redundancia de algún tipo en el diseño de sistemas. Esta redundancia es necesaria tanto para identificar los riesgos, condiciones peligrosas, como para tomar medidas correctivas.

- Patrón de Canal Único Protegido
- Patrón de Redundancia Homogénea
- Patrón Monitor-Actuador
- Patrón de Vigilancia
- Patrón de Ejecución de Seguridad

ALGUNOS PATRONES EN PARTICULAR

Arquitecturas por flujo de datos (dataflow)

Estructura



- Proporciona una estructura para sistemas que procesan una corriente de datos. Cada paso de la transformación se encapsula en un componente. Los datos se pasan a través de diversos componentes.
- Cada filtro actúa como un procesamiento de señales (flujo de entrada se transforman en un flujo de salida en función del conjunto de las normas). Los filtros se conectan entre sí, son entidades independientes (no comparten estado con otros filtros).
- Su ventaja es que la arquitectura es muy flexible:
  - Los componentes podrían ser reemplazados.
  - Se podrían insertar nuevos componentes.
  - Algunos componentes pueden ser reordenados.

Ejemplo

Compiladores tradicionales con sistema de lotes

Lex: Lexical Analyzer

Syn: Syntax Checker

Sem: Semantic Analyzer

Opt: Optimizer

Code: Code Generator

Variantes

Secuencial en lote:

- Los datos entran en el sistema y fluyen a través de los componentes independientes.
- Una tarea compleja puede ser dividida en subtareas. Cada subtarea es realizada por un componente.
- En esta variante los datos pasan en su totalidad de un componente a otro.

Pipe & Filters (Tuberías):

- Esta estructura es más flexible. Acepta ciclos y los componentes pueden ser reconectados.
- No es necesario el pasaje de la totalidad de los datos.

ARQUITECTURAS CALL & RETURN

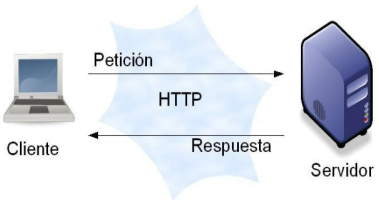
El sistema se ve como una serie de llamadas a procedimientos y funciones

Estilos

- Arquitectura por capas
- Cliente-Servidor (Arquitectura de 2 capas)
- Arquitectura orientada a servicios

CLIENTE-SERVIDOR

Estructura



Componentes

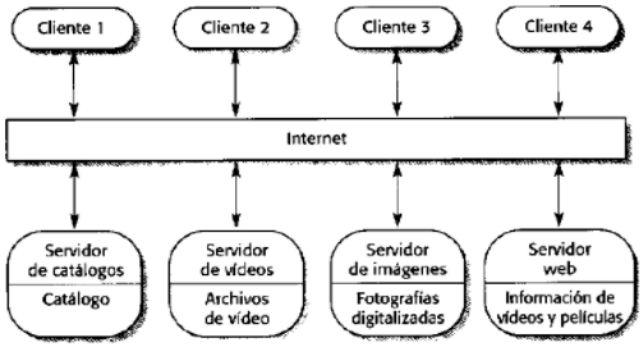
- Servidores: ofrecen servicios a otros subsistemas
- Clientes: puede haber varias instancias ejecutándose concurrentemente
- Red: para acceder servicios

Características

- Clientes conocen servidores y servicios disponibles
- Servidores no necesitan conocer identidad de clientes
- Clientes acceden a servicios a través de llamadas a procedimientos remotos

Ejemplo

Sistema de biblioteca de películas y fotografías



Se usa cuando…

- Desde varias ubicaciones se tiene que ingresar los datos en una base de datos compartida.
- Cuando la carga de un sistema es variable ya que los servidores se pueden replicar

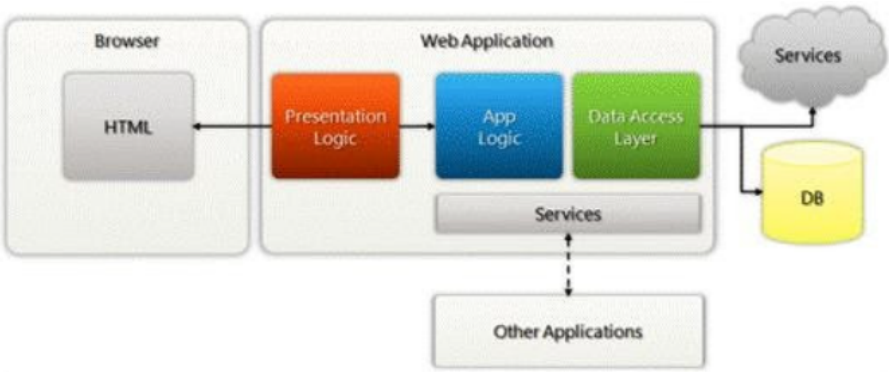
Ventajas y desventajas

- *Ventajas*
  - Cambios de funcionalidad centralizados
  - Testable
  - Mantenible
-
- *Desventajas*
  - El servidor puede ser un Cuello de botella
  - Único punto de falla
- *Variantes*:
  - Cliente liviano (Thin Client) o Cliente pesado (Fat Client)
  - Stateless or Stateful

ARQUITECTURA EN CAPAS

Estructura

- Capa de presentación
- Capa de lógica o negocio
- Capa de persistencia



Se usa cuando….

- Deben construirse nuevas facilidades encima de los sistemas existentes
- El desarrollo se dispersa a través de varios equipos de trabajo y cada uno es responsable por una capa de funcionalidad
- Existe un requerimiento para seguridad multinivel

Ventajas y desventajas

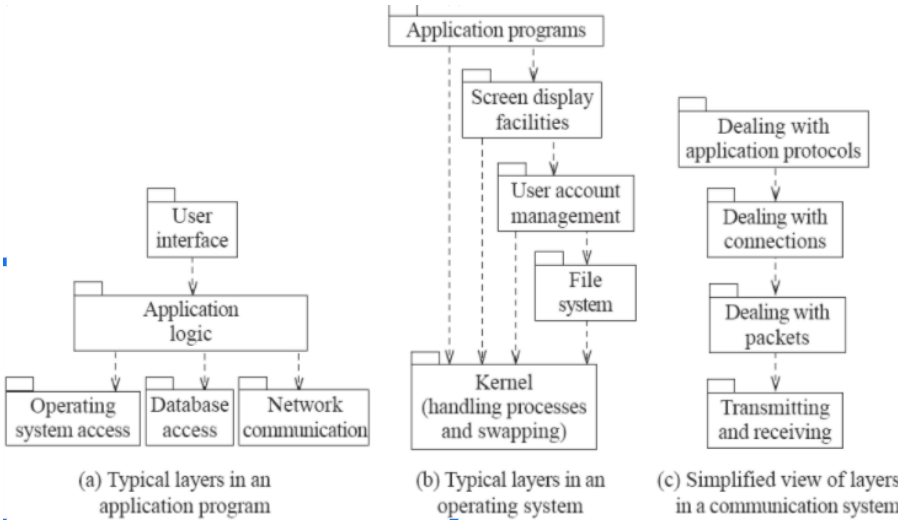
Ventajas

- Soporta el desarrollo incremental del sistema
- Soporta bien los cambios y es portable
- Capas más internas pueden ser re-implementadas ante nueva BD o SO

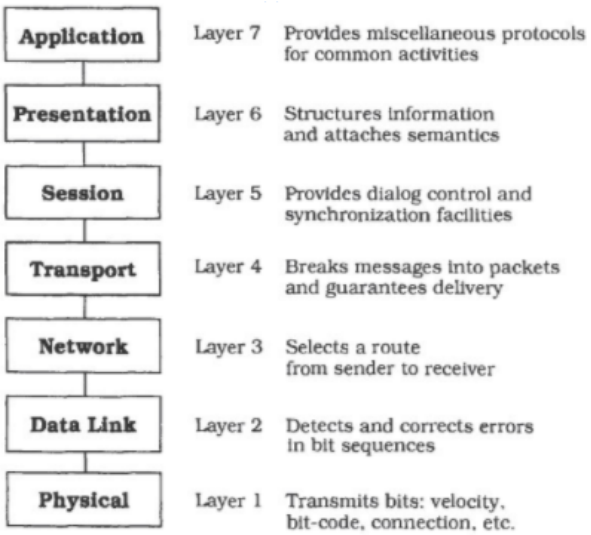
Desventajas

- Servicios de nivel superior deben atravesar todas las capas
- Rendimiento: un servicio tiene que ser interpretado varias veces en diferentes capas

Ejemplo



- OSI and TCP/IP PROTOCOL

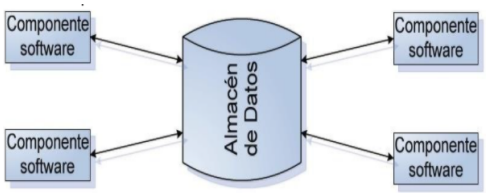


- Mac osx (Sistema operativo)
  - Aqua: user interface
  - Carbon: procedural api
  - Cocoa: object oriented api
  - Quartz: pdf based graphicssystem
  - Darwin: unix based kernel

Mac OS X system architecture

REPOSITORIO (CENTRADA EN DATOS)

Estructura



Se usa cuando…

- Se tiene un sistema donde se generan grandes volúmenes de información que deben almacenarse durante mucho tiempo

Ventajas y desventajas

Ventajas

- Forma eficiente de compartir gran cantidad de datos. No hay necesidad de transmitir datos explícitamente de un subsistema a otro
- Los subsistemas que producen datos no necesitan conocer cómo se utilizan sus datos por otros sistemas
- Fácil de administrar
- Posibilidad de generar copias de seguridad del modelo de datos fácilmente

Desventajas

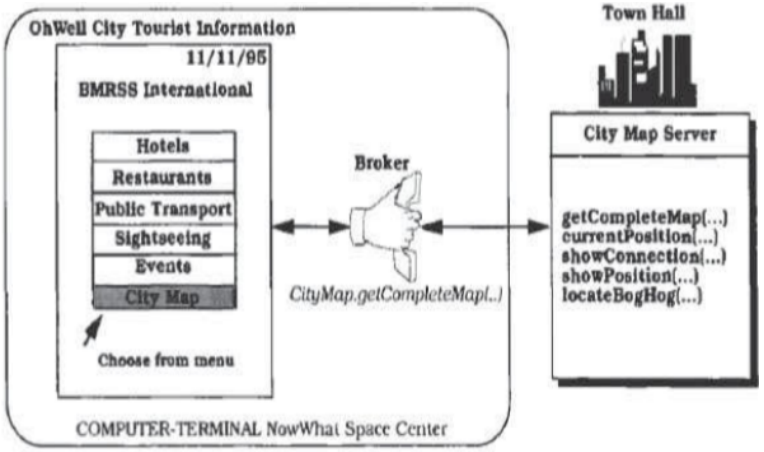
- Único punto de falla (los problemas afectan a todo el sistema)
  - Difícil integrar nuevos subsistemas si su modelo de datos no se ajusta al esquema
  - Cualquier modificación puede impactar sobre todos los componentes del software.

BROKER

Características

Se utiliza para estructurar un sistema distribuido de software con componentes desacoplados que interactúan de forma remota por medio de invocaciones de servicio. Utiliza un componente como intermediario responsable para la comunicación y coordinación, tales como solicitudes de reenvío, la transmisión de resultados y excepciones.

Ejemplo



Estándares

- CORBA (Common Object Request Broker Architecture) definido por el Object Management Group (OMG): que permite que diversos componentes de software escritos en múltiples lenguajes de programación y que corren en diferente computadoras, puedan trabajar juntos; es decir, facilita el desarrollo de aplicaciones distribuidas en entornos heterogéneos;
- DCOM (Distributed Component Object Model) es una tecnología propietaria de Microsoft para desarrollar componentes de software distribuidos sobre varios ordenadores y que se comunican entre sí.

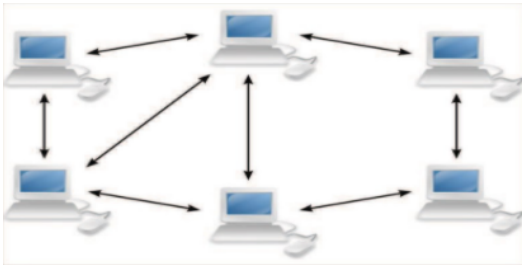
EXTRACT TRANSFORM AND LOAD (ETL)

Es el proceso que permite a las organizaciones mover datos desde múltiples fuentes, reformatearlos y limpiarlos, y cargarlos en otra base de datos, data mart, o data warehouse para analizar, o en otro sistema operacional para apoyar un proceso de negocio.

PEER-TO-PEER

Estructura

- El principio de organización central es que los pares son libres para comunicarse con cualquier otro par, sin necesidad de utilizar un servidor central
- Componentes: pares y red



Ventajas y desventajas

Ventajas

- Escalabilidad: las redes P2P tienen un alcance mundial con cientos de millones de usuarios potenciales;

- Robustez: incrementa la robustez en caso de haber fallos en la réplica excesiva de los datos hacia múltiples destinos;
- Descentralización: no existen nodos con funciones especiales, y por tanto ningún nodo es imprescindible para el funcionamiento de la red
- Distribución de costos entre los usuarios: se comparten o donan recursos a cambio de recursos;
- Anonimato: es deseable que en estas redes quede anónimo el autor de un contenido, el editor, el lector, el servidor que lo alberga y la petición para encontrarlo, siempre que así lo necesiten los usuarios;
- Seguridad: Los objetivos de un P2P seguro serían identificar y evitar los nodos maliciosos, evitar el contenido infectado, evitar el espionaje de las comunicaciones entre nodos, creación de grupos seguros de nodos dentro de la red, protección de los recursos de la red.

Desventajas

- Partición posible de la red si no hay lista de pares disponibles;
- Dificultad para garantizar una respuesta particular del sistema en cualquier punto en el tiempo.

Ejemplos

- Napster (fue un servicio de distribución de archivos de música (en formato MP3)
- Kademlia (Especifica la estructura de la red, regula la comunicación entre nodos y el intercambio de información, se crea una nueva red virtual sobre una red LAN/WAN existente)
- Ares Galaxy (es un programa P2P de compartición de archivos)
- BitCoin (es una moneda electrónica descentralizada)

PATRONES DE INTERACCIÓN (INTERFAZ)

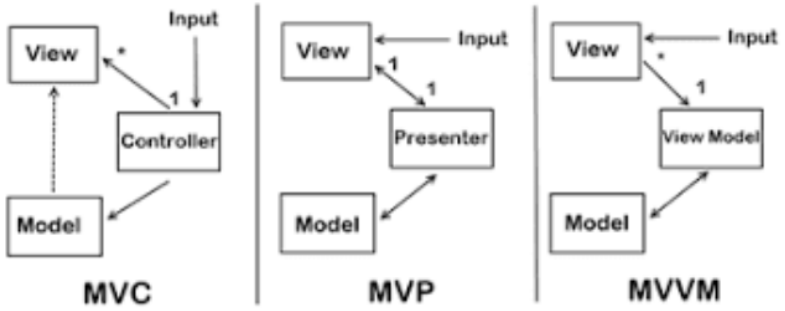
¿Patrón Arquitectónico o de Interacción?

Es habitual el debate acerca de si MVC es un patrón arquitectónico o un patrón de interacción asociado a la interfaz.
Si bien lo presentamos entre los patrones arquitectónicos, creemos que mayormente es un patrón que define cómo es la interacción entre usuario y sistema, por lo cual es un patrón de interacción.

MVC VS MVVM

- *MVC: Model-View-Controller:* significa Modelo Vista Controlador, porque en este patrón de diseño se separan los datos de una aplicación, la interfaz de usuario, y la lógica de negocio en tres componentes distintos. Cuando la lógica de negocio realiza un cambio, es necesario que ella sea la que actualiza la vista.
- *MVVM: Model-View-ViewModel:* significa Modelo Vista VistaModelo, porque en este patrón de diseño se separan los datos de la aplicación, la interfaz de usuario pero en vez de controlar manualmente los cambios en la vista o en los datos, estos se actualizan directamente cuando sucede un cambio en ellos, por ejemplo si la vista actualiza un dato que está presentando se actualiza el modelo automáticamente y viceversa.

Las principales diferencias entre MVC y MVVM son que en MVVM el “Controller” cambia a “ViewModel” y hay un “binder” que sincroniza la información en vez de hacerlo un controlador “Controller” como sucede en MVC.
El MVVM puede ser entendido como una correlación con el patrón de diseño Observer.



PATRONES DE INTEGRACIÓN

Integración

La integración de aplicaciones para empresas, también conocido por las siglas EAI (del inglés enterprise application integration) o EII (enterprise information integration, integración de la información de la empresa), se

define como el uso de software y principios de arquitectura de sistemas para integrar un conjunto de aplicaciones, dentro de cualquier empresa.
Entre los patrones arquitectónicos y los patrones de integración la diferencia es el nivel de abstracción

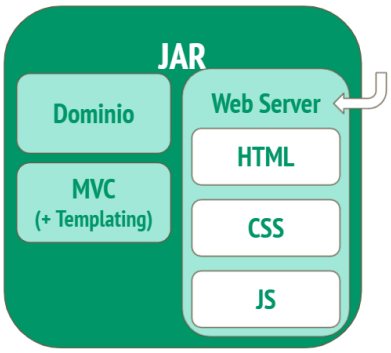
Patrones de integración

- Por base de datos:
  - Compartir información entre sistemas utilizando un mecanismo común de almacenamiento de datos
  - Similar a lo explicado cómo centrado en datos (repositorio)
- Por cola de mensajes:
  - Intercambio de mensajes a través de servicios que los distribuyen
  - Ejemplo: Amazon Simple Queue Service, IBM MQ
- RPC:
  - Los sistemas participantes en la integración exponen una interfaz pública sobre la que el resto de sistemas pueden invocar remotamente operaciones e intercambiar datos. Encapsulamiento de funcionalidades.
- Por bus:
  - Middleware / Broker

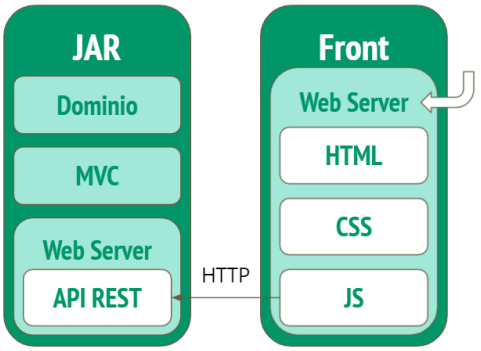
TECNOLOGÍAS FRONTEND

TIPOS DE DESPLIEGUE DE UNA WEB APP

Frontend embebido / empaquetado



Frontend independiente



HTML + CSS

Inspección del navegador, algunos elementos y sus estilos

```
<html>
<head>
  <link rel="stylesheet" type="text/css" href="mystyle.css">
  <style>
    body {background-color: linen;}
  </style>
</head>
<body style="background-color: lavender">
  <h1>Multiple Styles Will Cascade into One</h1>
  <p>Here, the background color of the page is set with inline CSS,
     and also with an internal CSS, and also with an external CSS.
  </p>
</body>
</html>
```

* * soy una simple etiqueta, no hago nada de nada

<p> soy un párrafo

<b> soy bold

<i> soy el estilo cursiva

<table> soy una tabla

<tr> soy una fila

<td> soy una columna

Bootstrap

Sistema de columnas

.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1
.col-md-8								.col-md-4			
.col-md-4				.col-md-4				.col-md-4			
.col-md-6						.col-md-6					

JavaScript

onClick | Manipulación del DOM | Usando APIs REST

```
var x = 10;
var y = 20;
var z=x+y;
document.write(z);

var data=200;//global variable
function a(){
  document.writeln(data);
}
a(); //calling JavaScript function

fetch("http://localhost:8080/mascotas")
  .then(response => response.json())
  .then(data => console.log(data));
```

Vue.js

Framework Reactivo + Templating

```
var x = 10;
var y = 20;
var z=x+y;
document.write(z);

var data=200;//global variable
function a(){
  document.writeln(data);
}
a(); //calling JavaScript function

fetch("http://localhost:8080/mascotas")
  .then(response => response.json())
  .then(data => console.log(data));
```

Links

- Proyecto visto en clase: https://github.com/jonybuzz/demo-frontend-vue
- W3 Schools (HTML, CSS, JS): https://www.w3schools.com
- Página de ejemplo elementos HTML: https://sceu.frba.utn.edu.ar/e-learning/detalle/experto-universitario/1745/experto-universitario-en-programacion-con-especializacion-en-un-lenguaje
- Bootstrap: https://getbootstrap.com/docs/4.3/layout

- CORS in Spark: https://gist.github.com/saeidzebardast/e375b7d17be3e0f4dddf
- Plantillas gratis: https://www.creative-tim.com/bootstrap-themes/free
- Vue.js 2: https://vuejs.org
- Intro a Vue.js (inglés): https://www.vuemastery.com/courses/intro-to-vue-js
- Página que te dice que tag de html funciona en los distintos browsers con sus diferentes verisones https://caniuse.com/
- https://www.creative-tim.com/