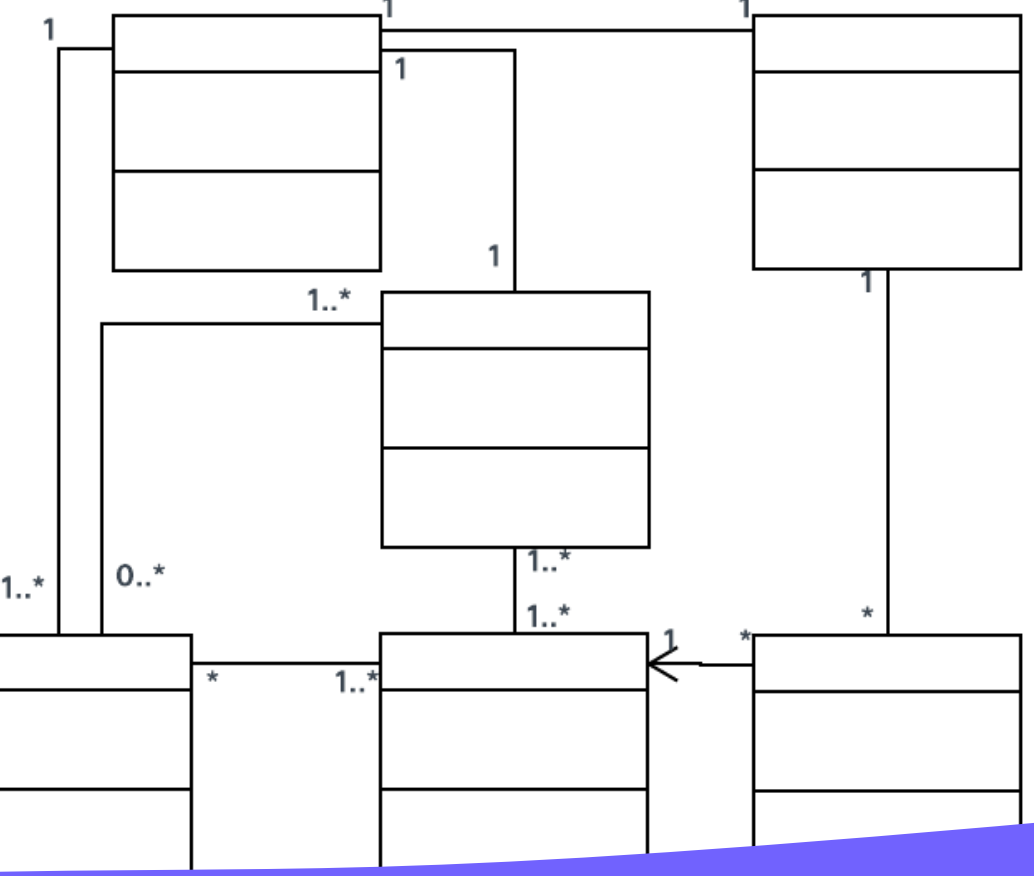
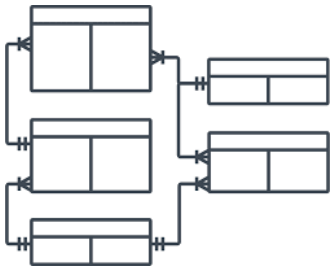




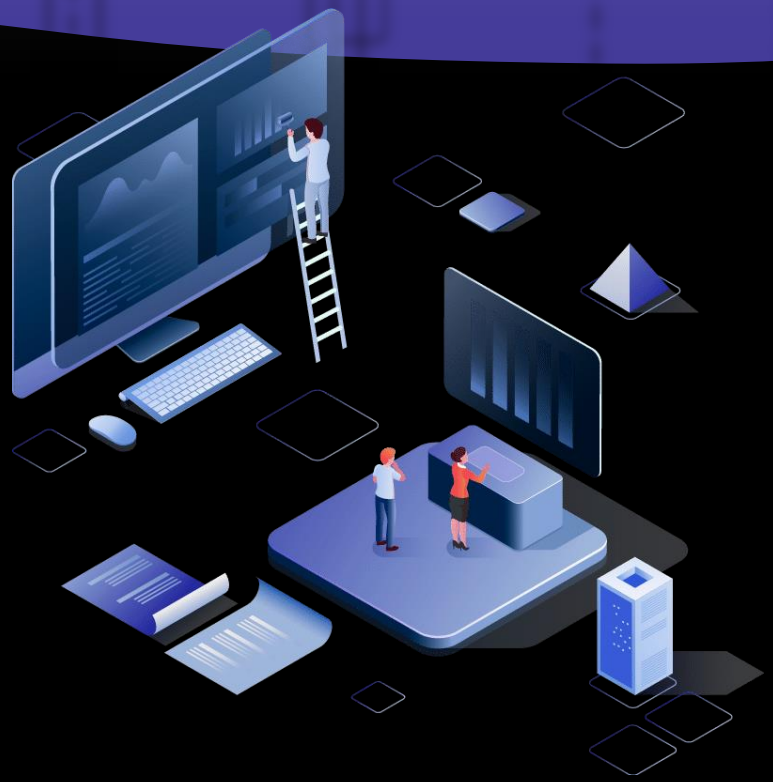
Diseño de Sistemas

UTN.BA
DPTO. INGENIERÍA EN SISTEMAS DE INFORMACIÓN
CÁTEDRA DISEÑO DE SISTEMAS

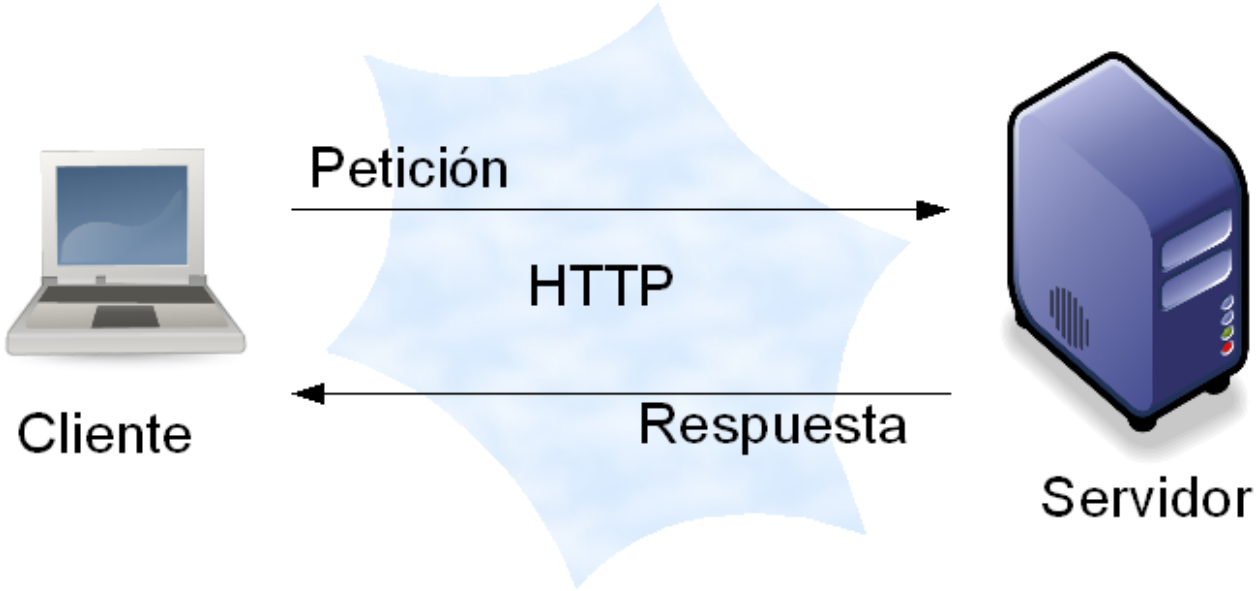
Agenda

- Cliente – Servidor
- MVC
- Principios SOLID

Cliente – Servidor (Estilo Arquitectónico)



Cliente - Servidor



Cliente - Servidor

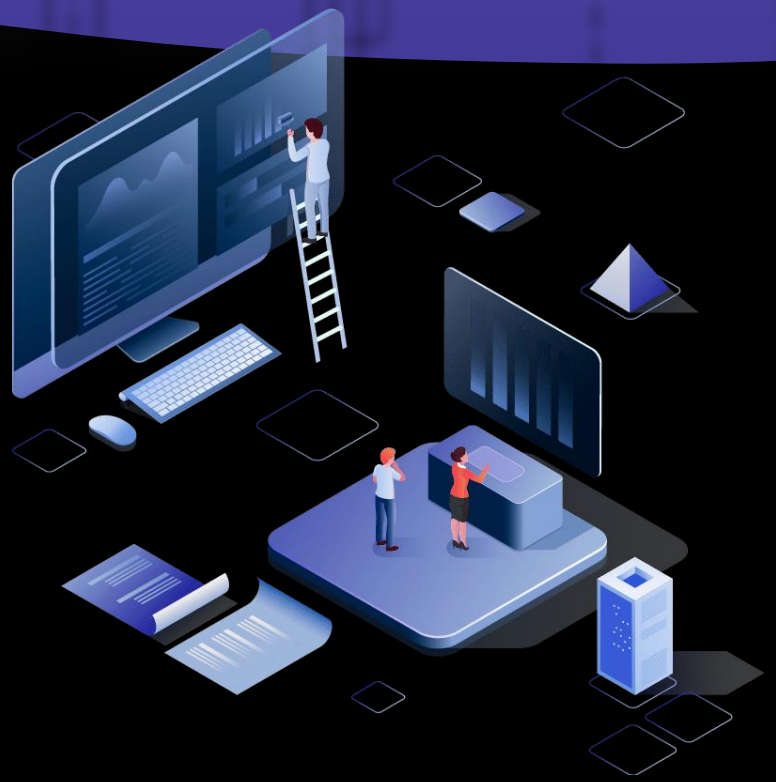
Ventajas

- Cambios de funcionalidad Centralizados (Mantenibilidad)
- Testeable
- Mantenible

Desventajas

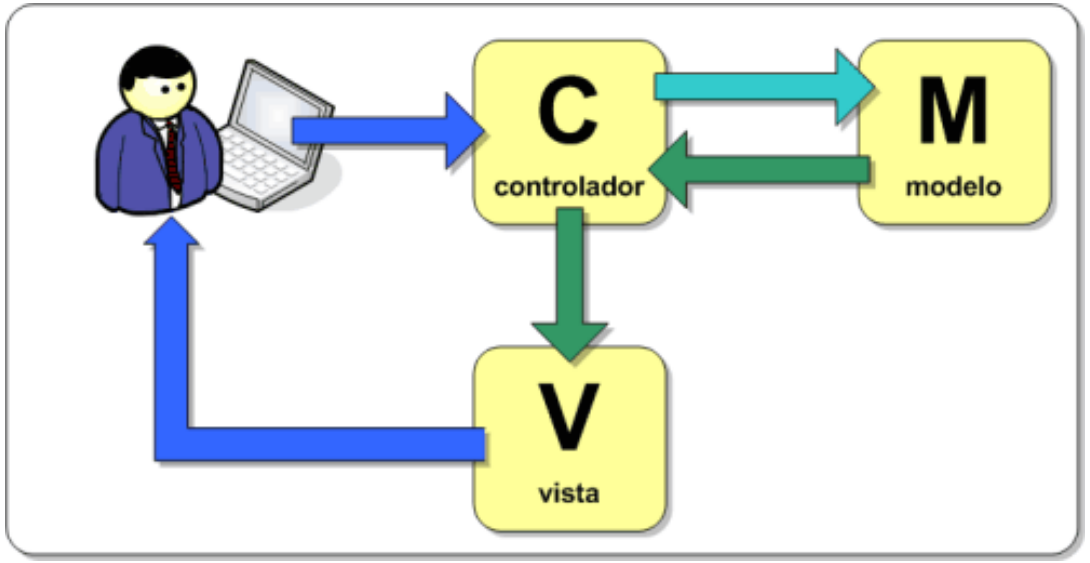
- El servidor puede ser un cuello de botella (Performance)
- Único punto de falla (Disponbilidad)

MVC (Patrón Arquitectónico)



MVC

Modelo-vista-controlador (MVC) es un **patrón** de arquitectura de software **de interacción** que separa la arquitectura en tres componentes: el modelo, la vista y el controlador.



MVC - Modelo

Está vinculado con la representación de los datos con los cuales el sistema opera. Está relacionado a la lógica y las reglas del negocio.

- Encapsula el estado del sistema
- Gestiona todos los accesos a dichos datos (consultas, actualizaciones, etc.)
- Valida la especificación de la lógica detallada en el “negocio”
- Envía a la Vista, a través del controller, los datos solicitados para que sean visualizados

MVC - Controlador

Responde a eventos (usualmente acciones del usuario) e invoca peticiones al modelo cuando se hace alguna solicitud sobre los datos.

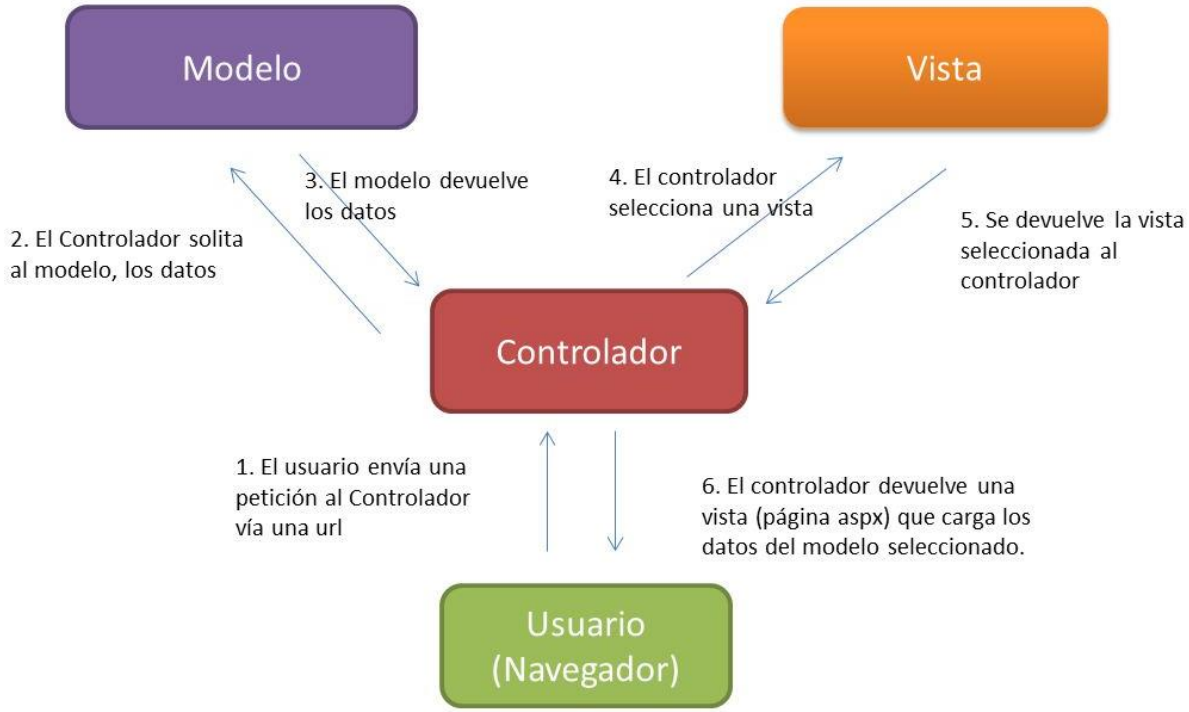
- Es el intermediario entre el modelo y la vista
- Define el comportamiento del sistema
- Traduce acciones del usuario a actualizaciones del modelo
- Es el gestor del ciclo de vida del sistema
- Responde a eventos
- Invoca peticiones al modelo

MVC - Vista

Presenta los datos y su forma de interactuar en un formato adecuado para el usuario.

- Posee lógica para poder representar los datos de la forma más “amigable” para el usuario
- Envía acciones del usuario al controlador
- Solicita actualizaciones al modelo a través del controller
- La comunicación entre la vista y el controlador se realiza mediante objetos de transferencia (*DTO - Data Transfer Objects*)

MVC - Web


Modelo
Vista
3. El modelo devuelve los datos
4. El controlador selecciona una vista
2. El Controlador solita al modelo, los datos
5. Se devuelve la vista seleccionada al controlador
Controlador
1. El usuario envía una petición al Controlador vía una url
6. El controlador devuelve una vista (página aspx) que carga los datos del modelo seleccionado.
Usuario (Navegador)

MVC – Validación de Datos

¿Dónde se validan los datos?

- ***EN CADA CAPA***. Esto se lo denomina ***validación en profundidad***.
- En la vista se puede validar campos requeridos y consistencia (números y/o letras donde corresponda, entre otras cosas).
- En el controlador se debería volver a validar la consistencia de los datos y los campos requeridos. También se puede realizar una validación adicional contra un servicio externo.
- En el modelo se deberían realizar las validaciones explícitas del negocio.

MVC – Ventajas y Desventajas

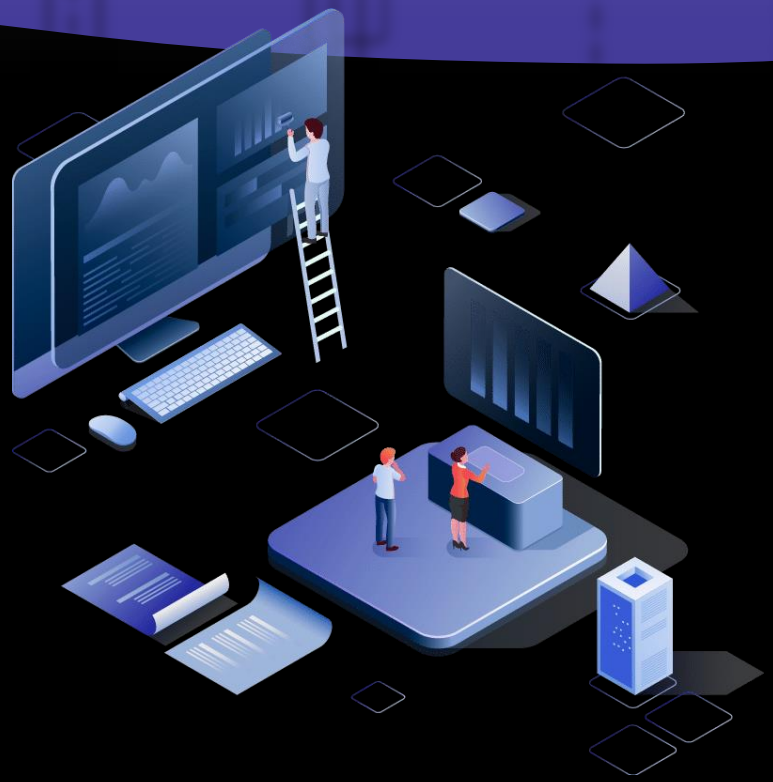
Ventajas

- Buena separación de intereses (concerns)
- Reusabilidad de Vistas y Controladores
- Flexibilidad

Desventajas

- Mayor complejidad de los Sistemas
- No siempre útil en aplicaciones con "*Poca Interactividad*" o con "*Vistas Simples*"
- Difícil de Testear "como un todo"

Principios SOLID

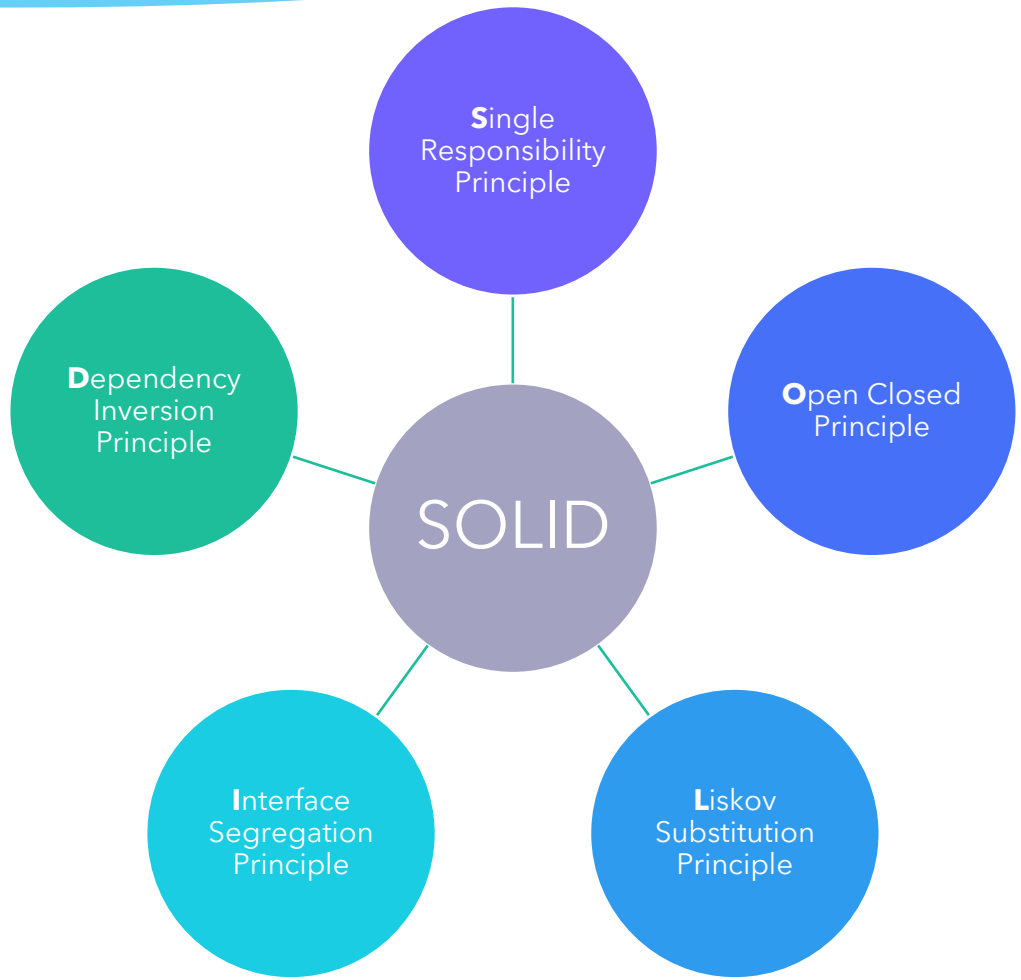


Principios SOLID

Son principios básicos de Programación Orientada a Objetos y Diseño de Sistemas que nos ayudan a obtener mejores diseños implementando una serie de reglas o principios.

Nos ayudan a evitar la generación de "Código Sucio"

Principios SOLID



SOLID - Single Responsibility Principle

- *Cada clase debe tener responsabilidad sobre una sola parte de la funcionalidad del software.*

- *Esta responsabilidad debe estar encapsulada por la clase, y todos sus servicios deben estar estrechamente alineados con esa responsabilidad.*

Evitar la clase "Dios" propiciando la alta Cohesión

SOLID - Open Closed Principle

- *Las entidades deben estar abiertas para la expansión, pero cerradas para su modificación.*
- *Se basa en la implementación de herencias y el uso de interfaces para resolver el problema.*

Se sugiere evitar la utilización excesiva de los "switchs" y propiciar el polimorfismo entre objetos

SOLID - Liskov Substitution Principle

"Cada clase que hereda de otra puede usarse como su superclase sin necesidad de conocer las diferencias entre las clases derivadas."

Se debería utilizar correctamente la herencia

SOLID – Interface Segregation Principle

- *Los clientes de un componente sólo deberían conocer de éste aquellos métodos que realmente usan y no aquellos que no necesitan usar.*

- *Muchas interfaces cliente específicas son mejores que una interfaz de propósito general.*

Se debería propiciar un diseño orientado a interfaces, para mantener el acoplamiento entre clases al mínimo posible, y también evitar generar interfaces extensas (con muchos métodos)

SOLID – Dependency Inversion Principle

"Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Es una forma de desacoplar módulos."

Se sugiere utilizar inyectores de dependencias

Lectura sobre inyección de dependencias

Gracias