

# Temas de Hoy



Patrones de Diseño



Diseño de Software



Design Patterns

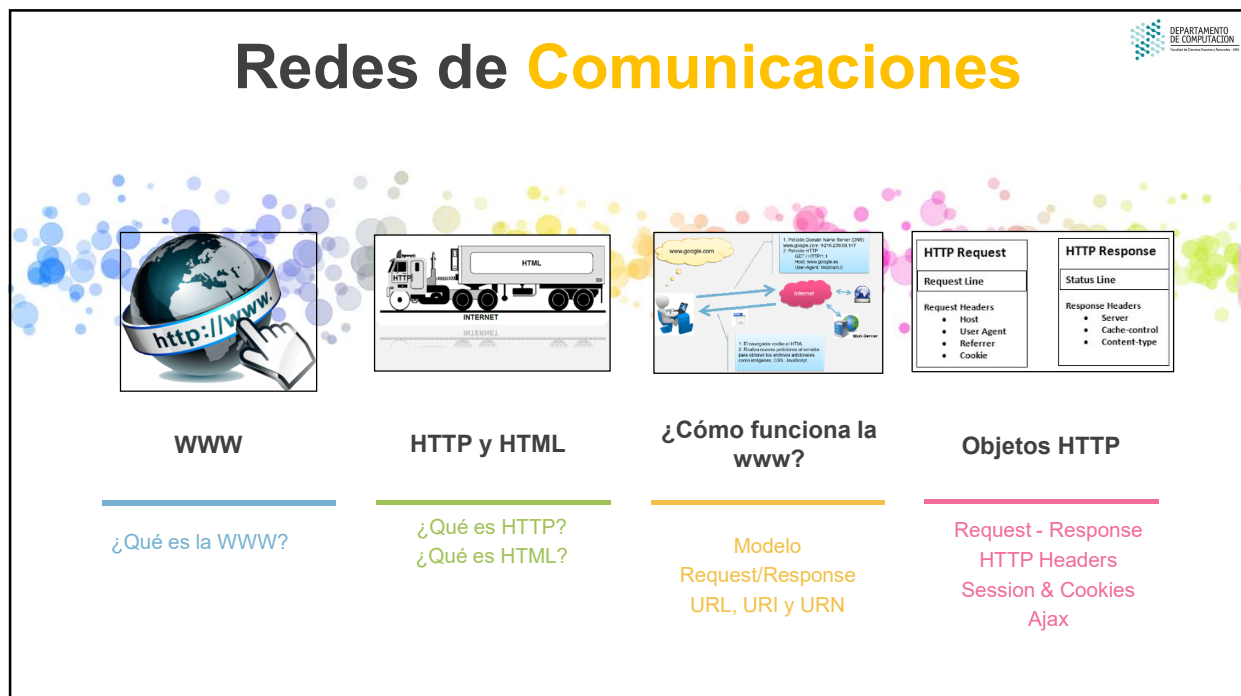
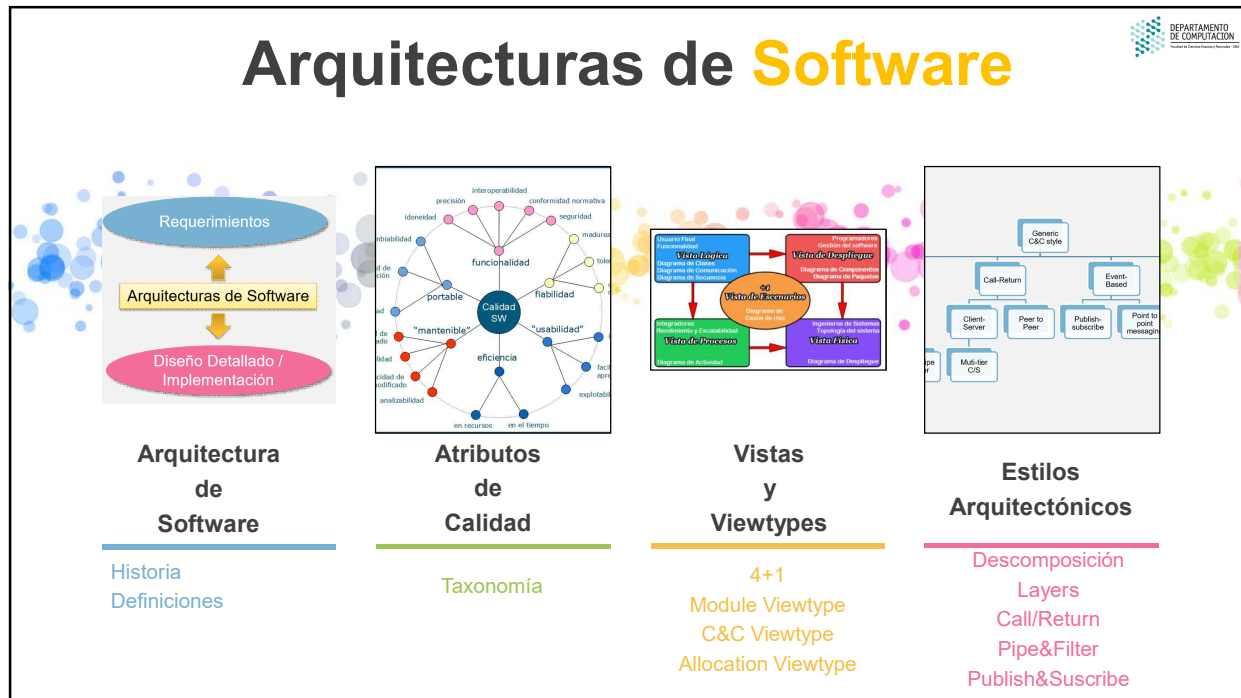
Arquitectura de Aplicaciones Web – 1°C 2018



PREVIOUSLY...

## Retomando...

DEPARTAMENTO DE COMPUTACION  
Facultad de Ciencias Exactas y Naturales - UBA







# Diseño de Software

***“Hay dos formas de construir un diseño del software. Una es hacerlo tan simple que sea obvio que no hay deficiencias y la otra es hacerlo tan complicado que no haya deficiencias obvias. El primer método es mucho más difícil.”***

**C. A. R. Hoare**

Arquitectura de Aplicaciones Web – 1°C 2018



## ¿Por dónde empezamos?

```
graph LR; A[Modelo de Dominio] --> B[Modelo de Análisis]; B --> C[Modelo de Diseño]; C --> D[Implementación];
```

**Modelo de Dominio**

- Diagrama de conceptos
- Relaciones entre conceptos
- Atributos conceptuales

**Modelo de Análisis**

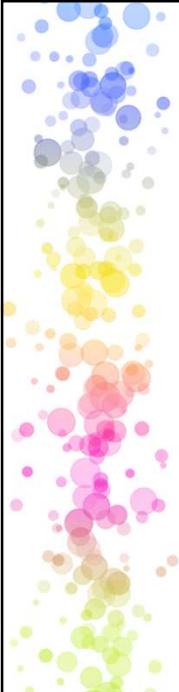
- Responsabilidades
- Colaboraciones entre conceptos

**Modelo de Diseño**

- Interfaces
- Abstracciones
- Generalizaciones
- Descomposición

**Implementación**

Arquitectura de Aplicaciones Web – 1°C 2018

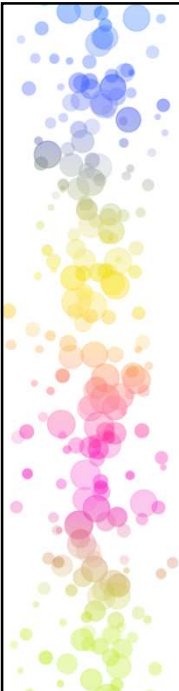


## Ejemplo

Se trata de un sistema simplificado de reserva de vuelos para una agencia de viajes que ofrece los siguientes servicios:

1. Las compañías aéreas ofrecen varios vuelos.
2. Una compañía abre y cierra las reservas para un determinado vuelo.
3. Un cliente puede reservar uno o más vuelos y para diferentes pasajeros.
4. Una reserva implica un único pasajero y un único vuelo.
5. Una reserva puede ser cancelada o confirmada.
6. Un vuelo tiene un aeropuerto de salida y otro de llegada.
7. Un vuelo tiene un día y una hora de salida y un día y hora de llegada.
8. Un vuelo puede implicar escalas en aeropuertos.
9. Una escala tiene una hora de llegada y otra de salida.
10. Un aeropuerto atiende a una o más ciudades.

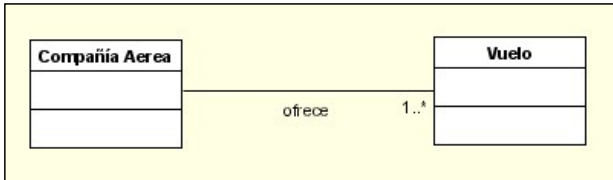
Arquitectura de Aplicaciones Web – 1°C 2018



## Ejemplo

“Las compañías aéreas ofrecen varios vuelos”

- Compañía aérea y Vuelo son conceptos importantes del mundo real con atributos y comportamientos, por lo que son clases candidatas para nuestro modelado estático.



```
classDiagram
    class CompañíaAerea {
        +
        +
        +
    }
    class Vuelo {
        +
        +
        +
    }
    CompañíaAerea "1" -- "1..*" Vuelo : ofrece
```

Arquitectura de Aplicaciones Web – 1°C 2018

## Ejemplo

“Una compañía abre y cierra las reservas para un determinado vuelo”

```

classDiagram
    class CompañíaAerea
    class Vuelo {
        -estado: EstadoDeVuelo
    }
    class EstadoDeVuelo {
        <<enumeration>>
        -abierto: int
        -cerrado: int
    }
    CompañíaAerea "1..*" -- "1" Vuelo : ofrece
  
```

Este no sería un enfoque correcto: cada objeto posee un estado actual por encima de los valores de sus atributos, que pertenece a las propiedades intrínsecas del concepto de objeto. Por lo tanto el concepto de estado no debe aparecer como un atributo en los diagramas de clase y debe ser modelado en la vista dinámica utilizando el diagrama de estado.

En un diagrama de clases los únicos conceptos dinámicos son las operaciones.

Arquitectura de Aplicaciones Web – 1°C 2018

## Ejemplo

“Una compañía abre y cierra las reservas para un determinado vuelo”

```

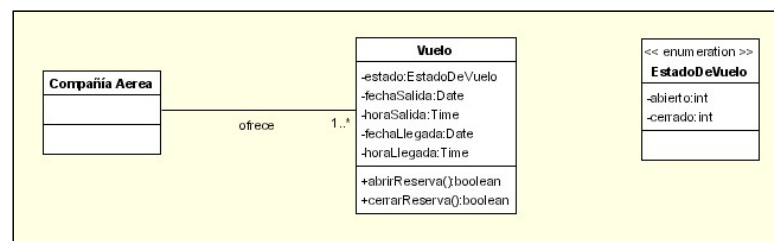
classDiagram
    class CompañíaAerea
    class Vuelo {
        -estado: EstadoDeVuelo
        +abrirReserva(): boolean
        +cerrarReserva(): boolean
    }
    class EstadoDeVuelo {
        <<enumeration>>
        -abierto: int
        -cerrado: int
    }
    CompañíaAerea "1..*" -- "1" Vuelo : ofrece
  
```

Arquitectura de Aplicaciones Web – 1°C 2018

## Ejemplo

“Un vuelo tiene un día y una hora de salida y un día y hora de llegada”

Estas nociones de fechas y horas representan simplemente valores, por lo que los modelaremos como atributos y no como simples objetos.

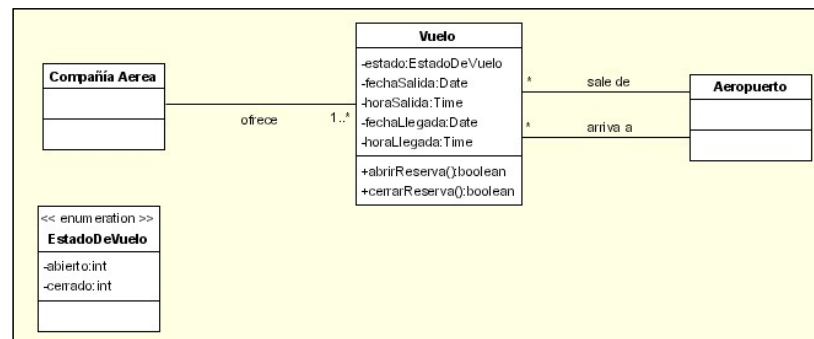


Arquitectura de Aplicaciones Web – 1°C 2018

## Ejemplo

“Un vuelo tiene un aeropuerto de salida y otro de llegada”

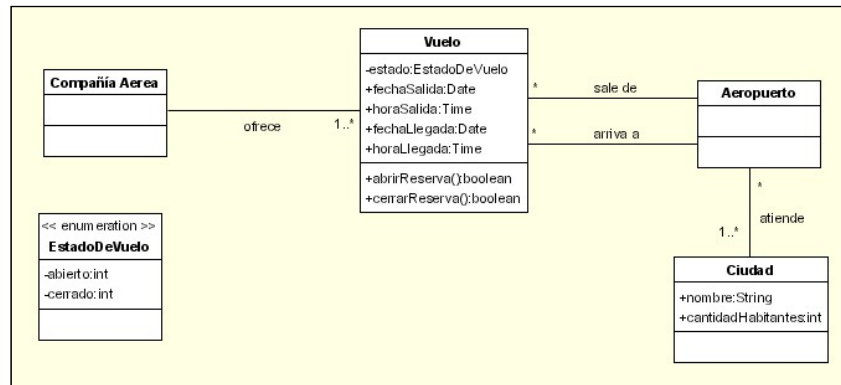
- Un objeto es algo más importante que un atributo.
- La noción de aeropuerto es compleja.



Arquitectura de Aplicaciones Web – 1°C 2018

## Ejemplo

“Cada aeropuerto atiende a una o varias ciudades”

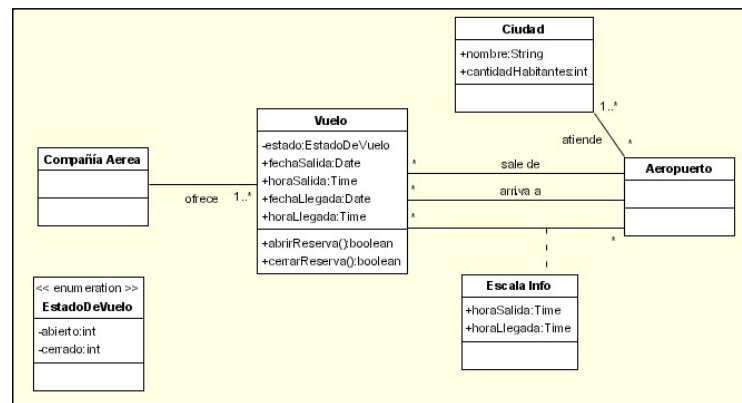


Arquitectura de Aplicaciones Web – 1°C 2018

## Ejemplo

“Un vuelo puede implicar escalas en aeropuertos”

“Una escala tiene una hora de llegada y otra de salida”



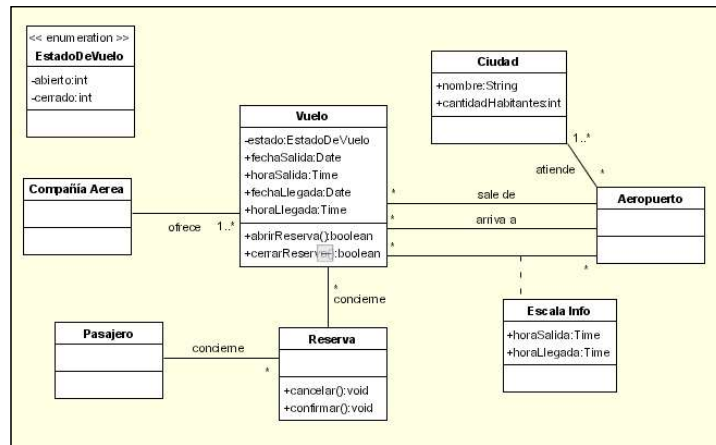
Arquitectura de Aplicaciones Web – 1°C 2018



## Ejemplo

“Una reserva implica un único vuelo y un único pasajero”.

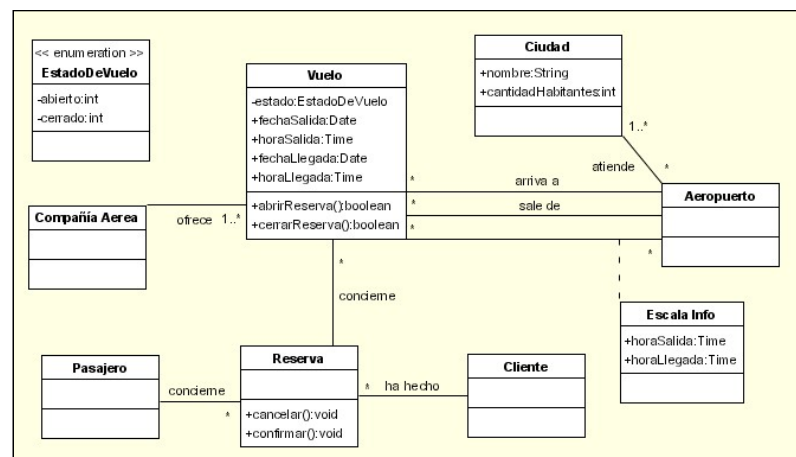
“Una reserva puede cancelarse o confirmarse”.



Arquitectura de Aplicaciones Web – 1°C 2018

## Ejemplo

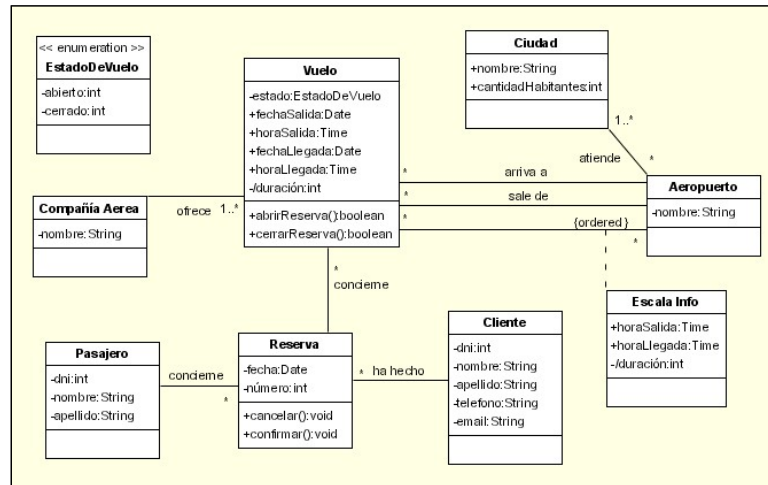
“Un cliente puede reservar uno o más vuelos y para pasajeros diferentes”.



Arquitectura de Aplicaciones Web – 1°C 2018

## Ejemplo

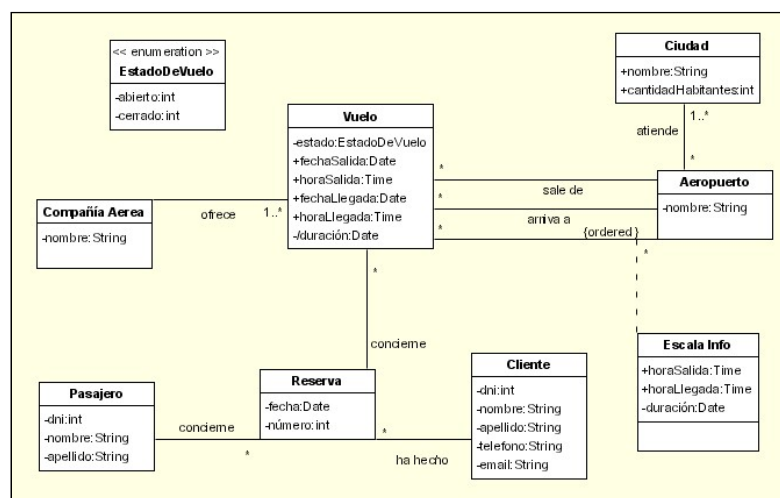
Agregar atributos y restricciones



Arquitectura de Aplicaciones Web – 1°C 2018

## Ejemplo

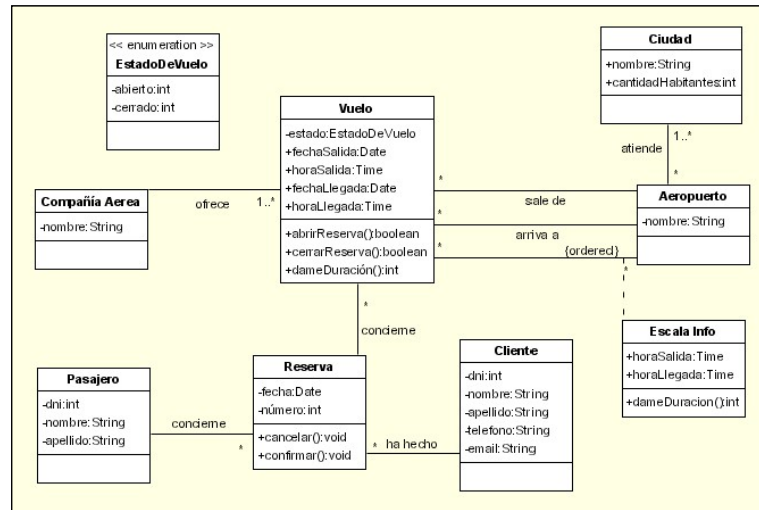
Un posible modelo de dominio...



Arquitectura de Aplicaciones Web – 1°C 2018

## Ejemplo

Se refina el modelo, para construir el modelo de análisis...



Arquitectura de Aplicaciones Web – 1°C 2018

## Ejemplo

Seguir refinando...

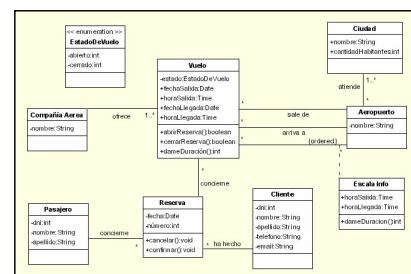
- Identificando interfaces
- Introduciendo generalizaciones (herencias)
- Generando clases “funcionales”
  - Managers
  - Controllers
  - Helpers

• ...

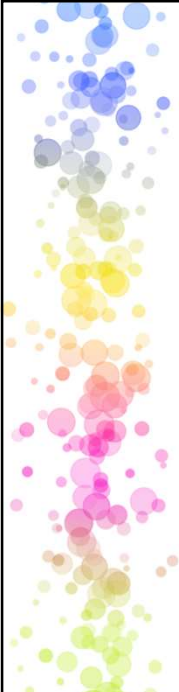
- Aplicando patrones

• ...

- **Respetando los principios de diseño**



Arquitectura de Aplicaciones Web – 1°C 2018

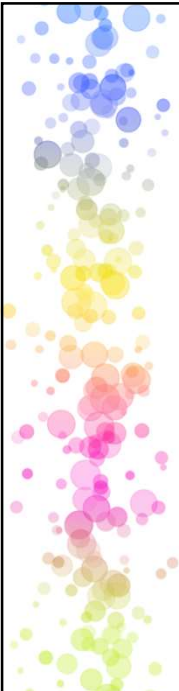


## Principios de Diseño

- ❖ Descomposición
- ❖ Abstracción
- ❖ Alta Cohesión
- ❖ Bajo Acoplamiento
- ❖ Modularidad
- ❖ Encapsulamiento

DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018

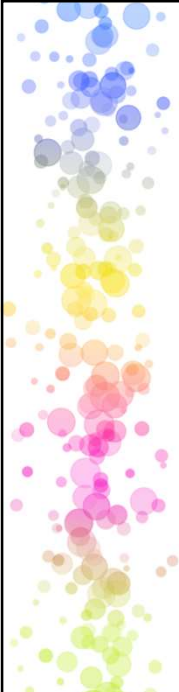


## Descomposición

- ❑ Concepto común a todos los ciclos de vida y técnicas de diseño.
- ❑ El concepto básico es simple:
  - Seleccionar una parte del problema
  - Determinar sus componentes usando cualquier mecanismo: funcional vs. estructuras de datos vs. orientado a objetos
  - Mostrar cómo interactúan los componentes
  - Repetir hasta satisfacer algún criterio de terminación

DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018

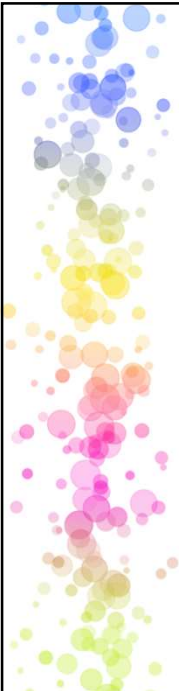


## ¿Cómo **descomponer**?

- Problemas que sabemos resolver
  - – Ej. Control, Visualización, Correspondencia, etc
- Pasos de ejecución
  - Ej. Filtros de procesamiento de imágenes
- Tempo de ejecución
  - Ej. Acumulación vs Utilización de Información
- Funcional
  - Ej. Facturación, Compras y Sueldos
- Modos de Operación
  - Normal vs Excepcional
- Datos
  - Ej. Guiado por el modelo conceptual. Clientes, Ambulancias...

DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018




## **Descomposición** de Software


- ❖ Módulos
  - ❖ Agrupa estructuras de datos y código (y posiblemente otros módulos)
  - ❖ Entidad estática
  - ❖ A veces, separa Interfaz de Implementación
    - ❖ Interfaz bien definida
  - ❖ A veces, es compilable de manera independiente
    - ❖ Es una unidad de trabajo para desarrollo
- ❖ Componentes
  - ❖ Entidades run-time
  - ❖ Descomposición para cumplir con ciertos requerimientos no funcionales distintos a los módulos (performance, distribución, tolerancia a fallas, seguridad, adaptabilidad en run-time, etc.).

DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018

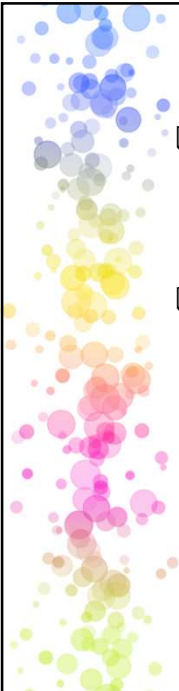


# Abstracción




- ❑ Provee un mecanismo para manejar la complejidad priorizando las características esenciales y suprimiendo los detalles de implementación.
- ❑ Permite posponer ciertas decisiones de detalle que ocurren a distintos niveles de análisis:
  - Representaciones / Algoritmos
  - Arquitectura / Estructura
  - Externo / Funcional

Arquitectura de Aplicaciones Web – 1°C 2018

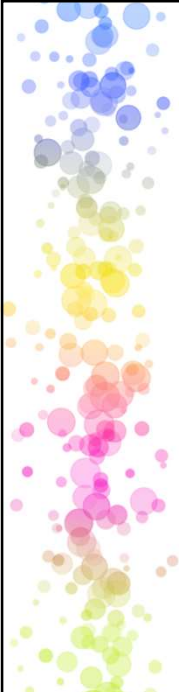


# Abstracción




- ❑ Suprimir detalles de implementación permite
  - ❑ Posponer ciertas decisiones de detalle que ocurren a distintos niveles de análisis.
  - ❑ Simplificar el análisis, comprensión y justificación de la decisión de diseño.
- ❑ Tipos de Abstracción
  - ❑ Procedural
    - ❑ ej. Funciones, métodos, procedimientos
  - ❑ Datos
    - ❑ ej. TADs, modelos de componentes
  - ❑ Control
    - ❑ ej. loops, iteradores, frameworks y multitasking

Arquitectura de Aplicaciones Web – 1°C 2018

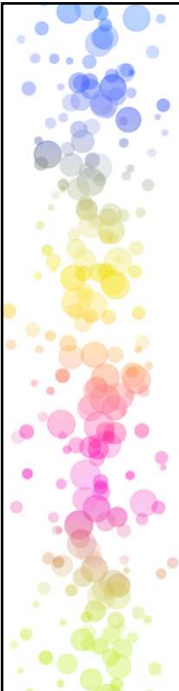


# Acoplamiento




- ❑ Principio de separación entre los distintos aspectos de distintas partes
- ❑ El concepto se aplica según la descomposición y el nivel de abstracción:
  - Relación entre las funcionalidades
  - Relación entre las responsabilidades
  - Relación entre los servicios
  - Relación entre los datos

Arquitectura de Aplicaciones Web – 1°C 2018



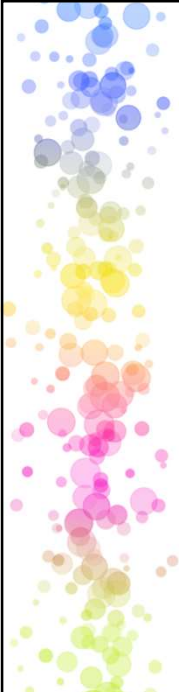
# Acoplamiento



- ❑ Mide el grado de dependencia del módulo sobre otros módulos y en particular las decisiones de diseño que estos hacen.
- ❑ Generalmente correlaciona inversamente con cohesión
  - Bajo/Débil acoplamiento y Alta Cohesión.
- ❑ Alto acoplamiento generalmente conlleva
  - Propagación de cambios cuando se altera un módulo.
  - Módulos son difíciles de entender aisladamente.
  - Reuso y testeo de módulos es difícil ya que se requieren otros módulos.
- ❑ Acoplamiento se incrementa si
  - Un módulo usa un tipo de otro módulo.
  - Si un módulo usa un servicio de otro módulo.
  - Si un módulo es un submódulo de otro.
- ❑ Bajo acoplamiento puede significar peor performance
  - Tradeoff...

Arquitectura de Aplicaciones Web – 1°C 2018





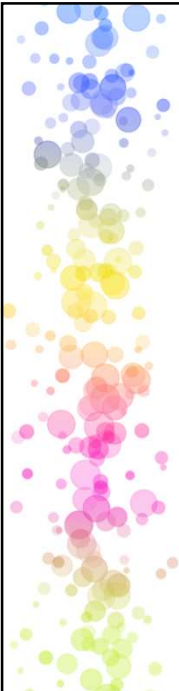
## Tipos de Acoplamiento

Ordenado de mayor a menor (según E. Yourdon y L. Constantine...)

- **Contenido**
  - Cuando un módulo modifica o confía en el lo interno de otro
  - ej. acceso a datos locales o privados
- **Común**
  - Cuando comparten datos comunes
  - ej. una variable global
- **Externo**
  - Cuando comparten aspectos impuestos externamente al diseño.
  - ej. formato de datos, protocolo de comunicación, interfaz de dispositivo.
- **Control**
  - Cuando un módulo controla la lógica del otro
  - ej. pasándole un flag de comportamiento).
- **Estampillado (Stamp)**
  - Cuando comparten una estructura de datos pero cada uno usa sólo una porción
  - Paso de todo un registro cuando el módulo sólo necesita una parte.
- **Datos**
  - Módulos se comunican a través de datos en parámetros
  - ej. llamado de funciones de otro módulo
- **Mensajes**
  - Módulos se comunican a través de mensajes. Posiblemente no se conocen explícitamente.

DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018



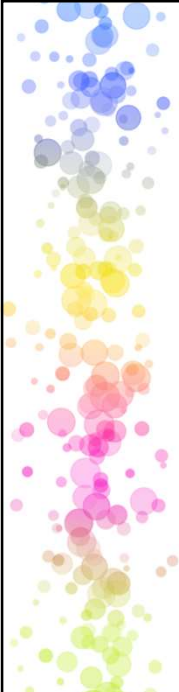
## Cohesión

- ❑ Principio de unión entre los distintos aspectos que incluye una parte.
- ❑ El concepto se aplica según la descomposición y el nivel de abstracción:
  - Relación entre las funcionalidades
  - Relación entre las responsabilidades
  - Relación entre los servicios
  - Relación entre los datos

DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018





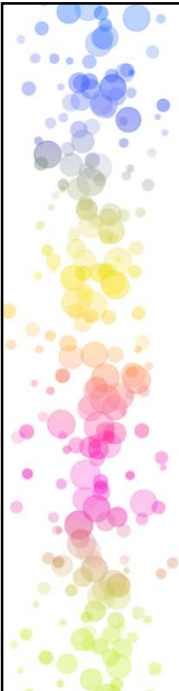
# Cohesión

DEPARTAMENTO DE COMPUTACIÓN

❑ Alta cohesión tiende a proveer...

- Robustez
- Confiabilidad
- Reusabilidad
- Comprensibilidad
- Testeabilidad
- Mantenibilidad

Arquitectura de Aplicaciones Web – 1°C 2018



# Tipos de Cohesión

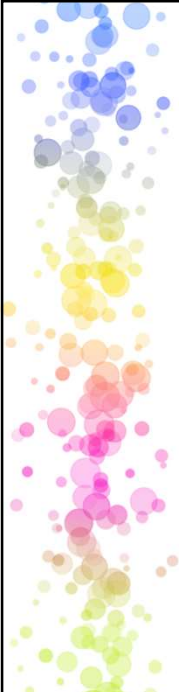
DEPARTAMENTO DE COMPUTACIÓN

Ordenado de peor a mejor (según E. Yourdon y L. Constantine en los 70's)


- Coincidental
  - ej. mis funciones de uso frecuente, utils.lib
- Lógico
  - Existe una categoría lógica que agrupa elementos aunque hagan cosas muy distintas
  - ej. todas las rutinas de I/O
- Temporal
  - Agrupadas por el momento en que se ejecutarán
  - ej. Funciones que atajan un error de output, crean un error en un log y notifican al usuario
- Procedural
  - Agrupadas por pertenecer a una misma secuencia de ejecución o política.
  - ej. funciones que chequean permisos y abren archivos
- Comunicacional
  - Agrupadas por operar sobre los mismo datos.
  - ej. objetos, operaciones sobre clientes.
- Secuencial
  - Agrupadas porque el output de uno es el input de otro
- Funcional
  - Agrupadas porque contribuyen a una tarea bien definida del módulo

**Aceptables**

Arquitectura de Aplicaciones Web – 1°C 2018

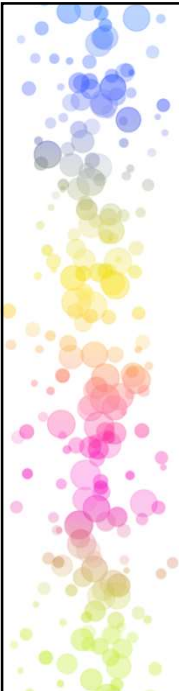


## Modularidad




- ❑ Un sistema modular es un sistema estructurado con abstracción es altamente independientes llamadas módulos.
- ❑ Modularidad es importante tanto para la etapa de diseño como de implementación.
- ❑ Módulos deben tener **interfaces abstractas** bien definidas.
- ❑ Módulos deben tener **alta cohesión** y **bajo acoplamiento**.

Arquitectura de Aplicaciones Web – 1°C 2018



## Encapsulamiento



- ❑ Motivación: detalles de diseño que pueden cambiar se ocultan detrás de interfaces abstractas (módulos).
- ❑ Los módulos se deben comunicar a través de interfaces bien definidas.
- ❑ La información que se oculta incluye:
  - Representaciones de datos
  - Algoritmos
  - Entradas y salidas
  - Interfaces de bajo nivel

Arquitectura de Aplicaciones Web – 1°C 2018

# Principios SOLID



- ❑ Introducidos por Robert C. Martin a principios de la década del 2000.

Inicial	Acrónimo	Concepto
S	SRP	Principio de responsabilidad única.
O	OCP	Principio de abierto/cerrado ( <i>Open/closed principle</i> ).
L	LSP	Principio de sustitución de Liskov ( <i>Liskov substitution principle</i> ).
I	ISP	Principio de segregación de la interfaz ( <i>Interface segregation principle</i> ).
D	DIP	Principio de inversión de la dependencia ( <i>Dependency inversion principle</i> ).

Arquitectura de Aplicaciones Web – 1°C 2018

## S → SRP → Single Responsibility Principle



**“No debería haber nunca más de una razón para cambiar una clase”**

Una clase debería concentrarse sólo en hacer una cosa de tal forma que cuando cambie algún requisito en mayor o menor medida dicho cambio sólo afecte a dicha clase por una razón.

```
interface Modem {
    public void dial(String pno);
    public void hangup();

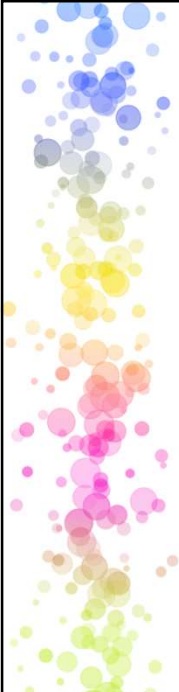
    public void send(char c);
    public char recv();
}
```



```
interface DataChannel {
    public void send(char c);
    public char recv();
}

interface Connection {
    public void dial(String phn);
    public char hangup();
}
```

Arquitectura de Aplicaciones Web – 1°C 2018




**O → OCP → Open / Closed Principle**

**“Las entidades de software deberían estar abiertas a la extensión pero cerradas a la modificación.”**

Sugiere cambiar el comportamiento de una clase mediante herencia, polimorfismo y composición.

DEPARTAMENTO DE COMPUTACION

Arquitectura de Aplicaciones Web – 1°C 2018



**O → OCP → Open / Closed Principle**

```
// Open-Close Principle - Bad example
class GraphicEditor {

    public void drawShape(Shape s) {
        if (s.m_type==1)
            drawRectangle(s);
        else if (s.m_type==2)
            drawCircle(s);
    }

    public void drawCircle(Circle r) {...}
    public void drawRectangle(Rectangle r) {...}

}

class Shape {
    int m_type;
}


class Rectangle extends Shape {
    Rectangle() {
        super.m_type=1;
    }
}

class Circle extends Shape {
    Circle() {
        super.m_type=2;
    }
}
```

- Imposible agregar una nueva *Shape* sin modificar el *GraphicEditor*.
- Hay que entender el *GraphicEditor* para agregar un *Shape*.
- *GraphicEditor* y *Shape* están fuertemente acopladas.
- Dependencias para el testing.
- *If-else* / *Case* se podrían evitar.

DEPARTAMENTO DE COMPUTACION

Arquitectura de Aplicaciones Web – 1°C 2018


  
 DEPARTAMENTO DE COMPUTACIÓN

## O → OCP → Open / Closed Principle

```
// Open-Close Principle - Good example
class GraphicEditor {
    public void drawShape(Shape s) {
        s.draw();
    }
}

class Shape {
    abstract void draw();
}

class Rectangle extends Shape {
    public void draw() {
        // draw the rectangle
    }
}
```

```
// Open-Close Principle - Bad example
class GraphicEditor {
    public void drawShape(Shape s) {
        if (s.m_type==1)
            drawRectangle(s);
        else if (s.m_type==2)
            drawCircle(s);
    }
    public void drawCircle(Circle r) {...}
    public void drawRectangle(Rectangle r) {...}
}


class Shape {
    int m_type;
}

class Rectangle extends Shape {
    Rectangle() {
        super.m_type=1;
    }
}

class Circle extends Shape {
    Circle() {
        super.m_type=2;
    }
}
```

- Imposible agregar una nueva *Shape* sin modificar el *GraphicEditor*.
- Hay que entender el *GraphicEditor* para agregar un *Shape*.
- *GraphicEditor* y *Shape* están fuertemente acopladas.
- Dependencias para el testing.
- *If-else / Case* se podrían evitar.

Arquitectura de Aplicaciones Web – 1°C 2018


  
 DEPARTAMENTO DE COMPUTACIÓN

## L → LSP → Liskov Substitution Principle

**“Las funciones que utilicen punteros o referencias a clases base deben ser capaces de usar objetos de clases derivadas de éstas sin saberlo.”**

Las subclases deben comportarse adecuadamente cuando sean usadas en lugar de sus clases base.

Arquitectura de Aplicaciones Web – 1°C 2018

**L → LSP → Liskov Substitution Principle**

**// Violation of Liskov's Substitution Principle**

```

class Rectangle
{
    int m_width;
    int m_height;

    public void setWidth(int width){
        m_width = width;
    }

    public void setHeight(int h){
        m_height = ht;
    }

    public int getWidth(){
        return m_width;
    }

    public int getHeight(){
        return m_height;
    }

    public int getArea(){
        return m_width * m_height;
    }
}

```

```

class Square extends Rectangle
{
    public void setWidth(int width){
        m_width = width;
        m_height = width;
    }

    public void setHeight(int height){
        m_width = height;
        m_height = height;
    }
}

```

Arquitectura de Aplicaciones Web – 1°C 2018

**L → LSP → Liskov Substitution Principle**

```

class LspTest
{
    private static Rectangle getNewRectangle()
    {
        // it can be an object returned by some factory ...
        return new Square();
    }

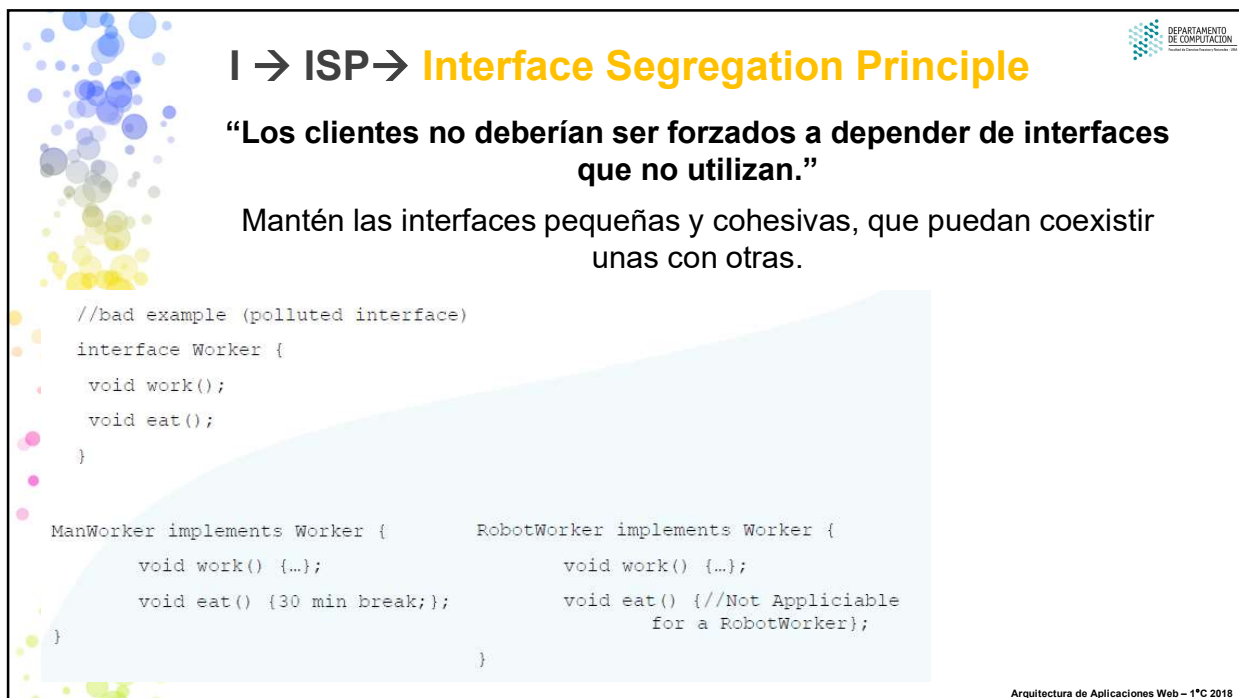
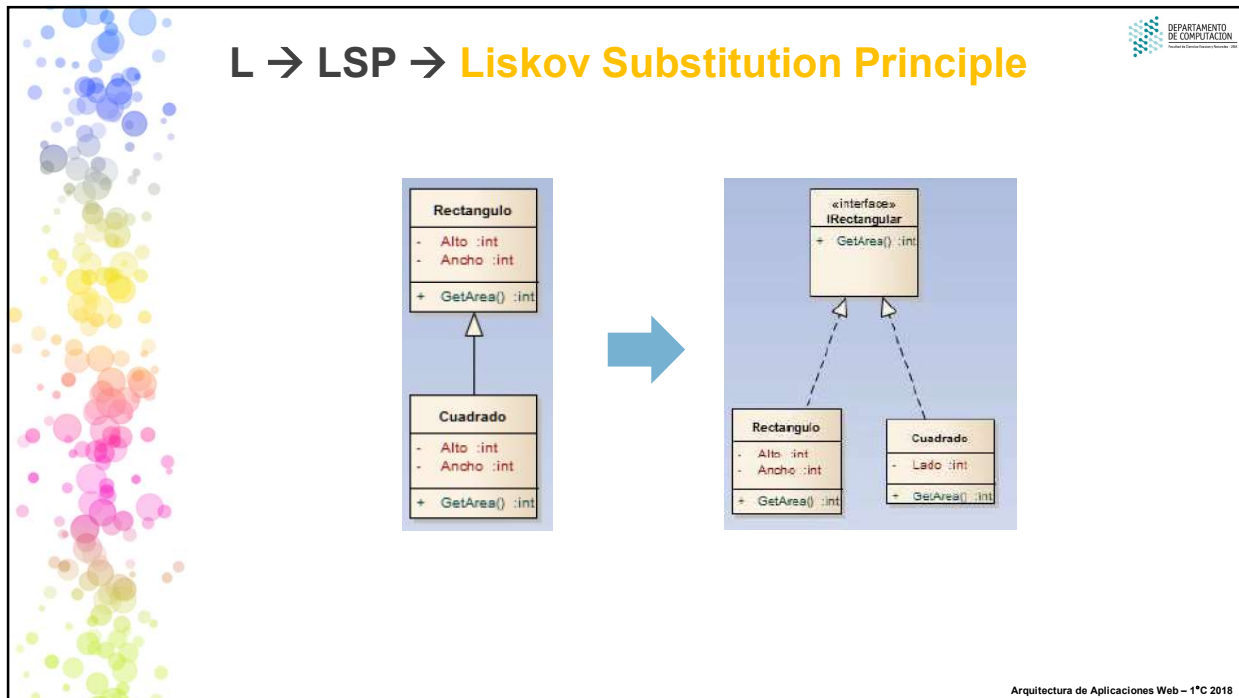
    public static void main (String args[])
    {
        Rectangle r = LspTest.getNewRectangle();
        r.setWidth(5);
        r.setHeight(10);

        // user knows that r it's a rectangle. It assumes that he's able to set the width and
        // height as for the base class


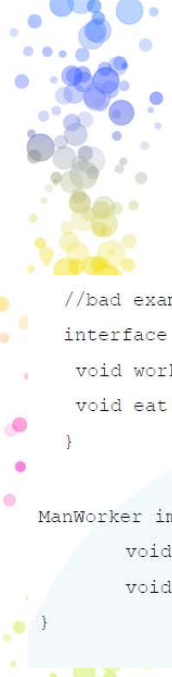
        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100 instead of 50.
    }
}

```

Arquitectura de Aplicaciones Web – 1°C 2018







## I → ISP → Interface Segregation Principle

**“Los clientes no deberían ser forzados a depender de interfaces que no utilizan.”**

Mantén las interfaces pequeñas y cohesivas, que puedan coexistir unas con otras.

```
//bad example (polluted interface)
interface Worker {
    void work();
    void eat();
}



ManWorker implements Worker {
    void work() {...};
    void eat() {30 min break;};
}

RobotWorker implements Worker {
    void work() {...};
    void eat() {//1 min break for a RobotWorker};
}
```

```
interface Workable {
    public void work();
}

interface Feedable {
    public void eat();
}
```

Arquitectura de Aplicaciones Web – 1°C 2018



## D → DIP → Dependency Inversion Principle


**“Módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.”**

**Las abstracciones no deberían depender de los detalles.  
Los detalles deberían depender de las abstracciones”**

Para conseguir robustez y flexibilidad y para posibilitar la reutilización, utilizar interfaces y abstracciones.

Arquitectura de Aplicaciones Web – 1°C 2018






## D → DIP → Dependency Inversion Principle

```
//DIP - bad example
public class EmployeeService {
    private EmployeeFinder emFinder //concrete class, not abstract. Can access a SQL DB for instance
    public Employee findEmployee(...) {
        emFinder.findEmployee(...)
    }
}
```

DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018



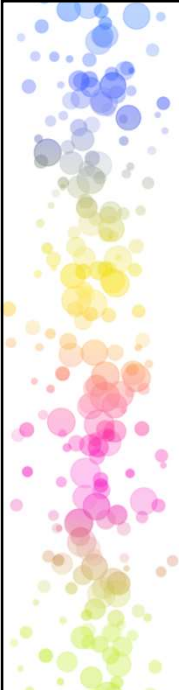
## D → DIP → Dependency Inversion Principle

```
//DIP - bad example
public class EmployeeService {
    private EmployeeFinder emFinder //concrete class, not abstract. Can access a SQL DB for instance
    public Employee findEmployee(...) {
        emFinder.findEmployee(...)
    }
}

//DIP - fixed
public class EmployeeService {
    private IEmployeeFinder emFinder //depends on an abstraction, not an implementation
    public Employee findEmployee(...) {
        emFinder.findEmployee(...)
    }
}
```

DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018



**D → DIP → Dependency Inversion Principle**

Requisito: quiero que la casa tenga puerta y ventana y que sean personalizables.

```
public class Door
{
}

public class Window
{
}

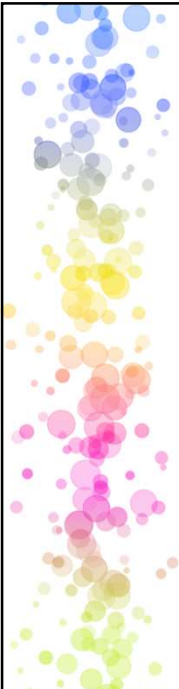
public class House
{
    private Door _door;
    private Window _window;

    public House()
    {
        _door = new Door();
        _window = new Window();
    }
}
```

Al igual que el ejemplo anterior, la dependencia es sobre clases concretas.

DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018



**D → DIP → Dependency Inversion Principle**

Requisito: quiero que la casa tenga puerta y ventana y que sean personalizables.

```
public class House
{
    private IDoor _door;
    private IWindow _window;


    public House()
    {
        _door = new Door();
        _window = new Window();
    }

    public IDoor Door;
    public IWindow Window;
}
```

Para personalizar la puerta y ventana... hay que modificar el constructor.

DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018



**D → DIP → Dependency Inversion Principle**

Requisito: quiero que la casa tenga puerta y ventana y que sean personalizables.

```
public class House
{
    private IDoor _door;
    private IWindow _window;

    public House(){}

    public House(IDoor door, IWindow window)
    {
        _door = door;
        _window = window;
    }

    public IDoor Door;
    public IWindow Window;
}
```

Ejemplo de inyección de dependencias  
(a través del constructor)


```
public interface IDoor
{
    string GetColor();
    void OnOpen();
    void OnClose();
}

public class BrownDoor : IDoor
{
    public void OnOpen() { }
    public void OnClose() { }
    public string GetColor()
    {
        return "Soy una puerta marrón";
    }
}

public interface IWindow
{
    string GetSize();
    void OnOpen();
    void OnClose();
}

public class BigWindow : IWindow
{
    public void OnOpen() { }
    public void OnClose() { }
    public string GetSize()
    {
        return "Soy una ventana grande";
    }
}
```

Arquitectura de Aplicaciones Web – 1°C 2018



**D → DIP → Dependency Inversion Principle**

Requisito: quiero que la casa tenga puerta y ventana y que sean personalizables.

```
public class House
{
    private IDoor _door;
    private IWindow _window;

    public House(){}

    public House(IDoor door, IWindow window)
    {
        _door = door;
        _window = window;
    }

    public IDoor Door;
    public IWindow Window;
}
```

Ejemplo de inyección de dependencias  
(a través del constructor)

```
public interface IDoor
{
    string GetColor();
    void OnOpen();
    void OnClose();
}

public class BrownDoor : IDoor
{
    public void OnOpen() { }
    public void OnClose() { }
    public string GetColor()
    {
        return "Soy una puerta marrón";
    }
}

public interface IWindow
{
    string GetSize();
    void OnOpen();
    void OnClose();
}

public class BigWindow : IWindow
{
    public void OnOpen() { }
    public void OnClose() { }
    public string GetSize()
    {
        return "Soy una ventana grande";
    }
}
```

Arquitectura de Aplicaciones Web – 1°C 2018

## Principios GRASP



- ❑ Introducidos por Craig Larman en 1997  
(*Applying UML and Patterns: An Introduction to Object-Oriented Analysis & Design*)
- ❑ General Responsibility Assignment Software Patterns
- ❑ Son más buenas prácticas que patrones... aunque se presentan como Problema-Solución

Concepto
Experto en información
Creador
Controlador
Alta cohesión y bajo acoplamiento
Polimorfismo
Fabricación pura
Indirección

Arquitectura de Aplicaciones Web – 1°C 2018

## Experto en información


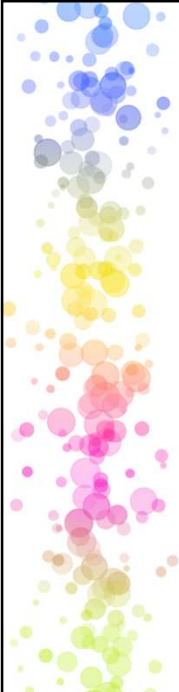


**Problema:** ¿Cuál es el principio fundamental en virtud del cual se asignan las responsabilidades en el diseño orientado a objetos?

**Solución:** asignar una responsabilidad al experto en información: la clase que cuenta con la información para cumplir la responsabilidad.

- La responsabilidad de creación de un objeto o la implementación de un método debe recaer sobre la clase que conoce toda la información necesaria para crearlo o ejecutarlo, contribuyendo a tener mayor cohesión y mayor encapsulamiento y disminuyendo el acoplamiento.
- Relacionado con la S de SOLID.

Arquitectura de Aplicaciones Web – 1°C 2018




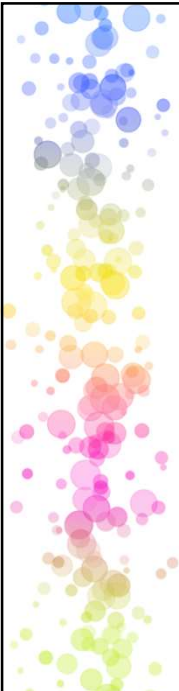
# Creador

**Problema:** ¿Quién debería ser responsable de crear una nueva instancia de alguna clase?

**Solución:** Asignarle a la clase B la responsabilidad de crear una instancia de la clase A en uno de los siguientes casos:

- B agrega los objetos A.
- B contiene los objetos A.
- B registra las instancias de los objetos A.
- B utiliza específicamente los objetos A.
- B tiene los datos de inicialización que serán transmitidos a A cuando este objeto sea creado (así que B es un Experto respecto a la creación de A).
- B es un creador de los objetos A.
- Si existe más de una opción, prefiera la clase B que agregue o contenga la clase A.

Arquitectura de Aplicaciones Web – 1°C 2018




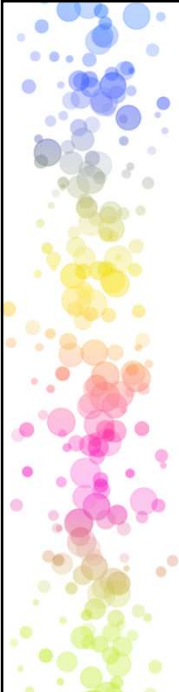
# Controlador

**Problema:** ¿quién debería ser responsable de manejar un evento del sistema?

**Solución:** asignar la responsabilidad al controlador, que será una clase que:

- Representa al sistema completo, a la organización...  
(controlador fachada o facade)
- Representa una parte activa del mundo real que desencadena de tal evento...  
(controlador de rol)
- Representa un manejador artificial de eventos...  
(controlador de caso de uso)

Arquitectura de Aplicaciones Web – 1°C 2018




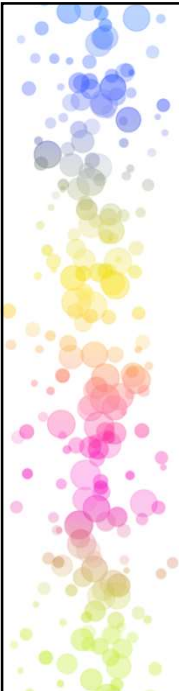
## Bajo Acomplamiento

**Problema:** ¿cómo mantener un bajo acoplamiento para lograr, entre otras cosas, alta reutilización?

**Solución:** asignar las responsabilidades de manera que el acoplamiento permanezca bajo.

- Situaciones de acoplamiento
  - X tiene un miembro o declara una variable de clase Y.
  - X tiene un método que toma como parámetro un objeto de clase Y.
  - X es un descendiente de Y.
- Desventajas del acoplamiento
  - Los cambios de una clase pueden implicar cambios en las clases relacionadas.
  - Dificultad de comprensión.
  - Dificultad de reutilización.
  - ¿Rendimiento?

Arquitectura de Aplicaciones Web – 1°C 2018




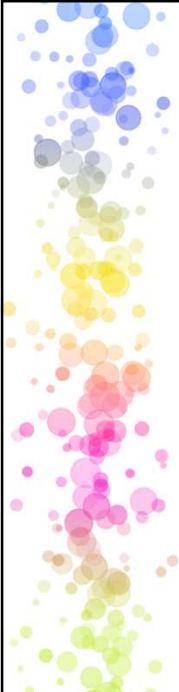
## Alta Cohesión

**Problema:** ¿cómo mantener la complejidad de una clase en niveles manejables?

**Solución:** asignar las responsabilidades de manera que la cohesión se mantenga alta.

- Desventajas de una clase con baja cohesión
  - Difícil de comprender
  - Difícil de reutilizar
  - Difícil de mantener
  - Delicada... sensible a los cambios.

Arquitectura de Aplicaciones Web – 1°C 2018




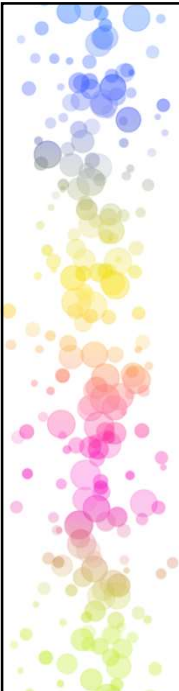
## Fabricación Pura

**Problema:** ¿cómo proceder cuando las soluciones encontradas comprometen la cohesión y el acoplamiento?

**Solución:** asignar un conjunto cohesivo de responsabilidades a una clase artificial (no representa ningún concepto del dominio del problema).

- Ej: Objetos “Persistidores”, etc...

Arquitectura de Aplicaciones Web – 1°C 2018



## Indirección

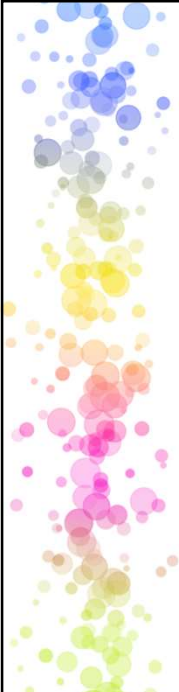
**Problema:** ¿dónde asignar responsabilidades para evitar/reducir el acoplamiento directo entre elementos y mejorar la reutilización?

**Solución:** asigne la responsabilidad a un objeto que medie entre los elementos. Resultado: acoplamiento indirecto.

- Ej: Adapters, Facade...

Arquitectura de Aplicaciones Web – 1°C 2018






## Variaciones Protegidas


**Problema:** ¿cómo me protejo a futuros cambios?

**Solución:** todos lo que se prevea susceptible de modificaciones (en un análisis previo) debe envolverse en interfaces, utilizando polimorfismo para crear varias implementaciones y posibilitar implementaciones futuras.




Arquitectura de Aplicaciones Web – 1°C 2018

## Otros Principios



- ❑ KISS (Keep it simple, Stupid!)
  - Evitar o eliminar la complejidad innecesaria.
- ❑ DRY (*Don't repeat yourself*)
  - No hay que escribir código duplicado. Si estás repitiendo código, extrae ese código a una función para encapsularlo.
- ❑ YAGNI (*You ain't gonna need it*)
  - No debemos implementar algo si no estamos seguros de necesitarlo.



Arquitectura de Aplicaciones Web – 1°C 2018



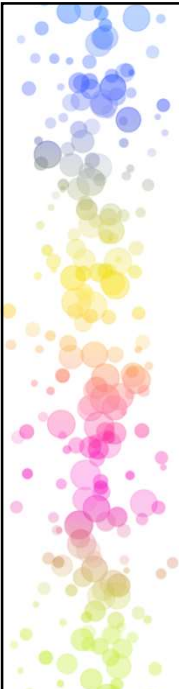


## Patrones de Diseño

- Gamma, Helm, Johnson & Vlissides, 1995 (Aka The gang of four - GoF)
- Soluciones esquemáticas (buen diseño) a problemas recurrentes en el desarrollo de software OO.
- 1er Catálogo tenía 23 patrones:
  - fenómeno de definición terminológica
- Los Design Patterns se suponen que incorporan los principios de diseño que vimos.



Arquitectura de Aplicaciones Web – 1°C 2018


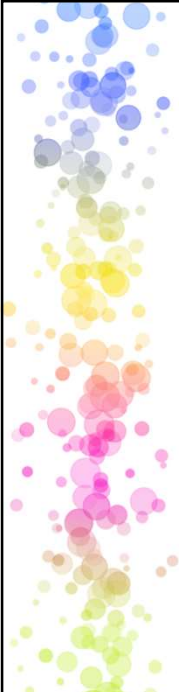


## Patrones de Diseño

### ¿Qué es un patrón de diseño?

- Ante un problema reiterado ofrece una solución contrastada que lo resuelve.
- Describe el problema en forma sencilla.
- Describe el contexto en que ocurre.
- Describe los pasos a seguir.
- Describe los puntos fuertes y débiles de la solución.
- Describe otros patrones asociados.

Arquitectura de Aplicaciones Web – 1°C 2018


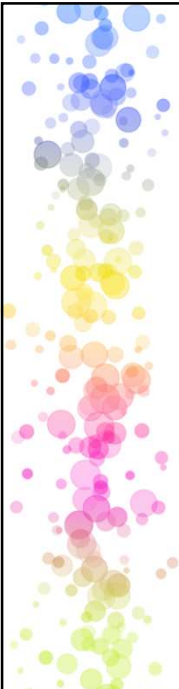


# Patrones de Diseño

## ¿Por qué usarlos?

- Mejora en la comunicación y documentación
  - “Hay que hacer un Factory Method”
  - Facilita la documentación interna del proyecto.
- Mejora la ingeniería de software.
  - Eleva el nivel del grupo de desarrollo.
- Previene “reinventar la rueda” en diseño
  - Son soluciones ya probadas.
- Mejora la calidad y estructura
  - “¿Cuán grande debe ser una clase?”

Arquitectura de Aplicaciones Web – 1°C 2018

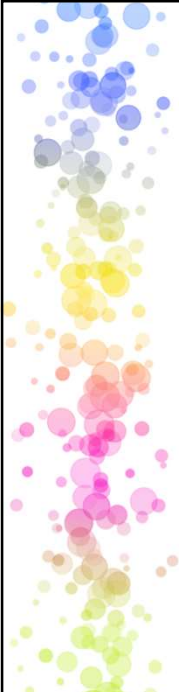


# Patrones de Diseño

## Tipos de Patrones

- **De Creación:** abstraen el proceso de creación de instancias.
- **Estructurales:** se ocupan de cómo clases y objetos son utilizados para componer estructuras de mayor tamaño.
- **De Comportamiento:** atañen a los algoritmos y a la asignación de responsabilidades entre objetos.

Arquitectura de Aplicaciones Web – 1°C 2018



## Patrones de Diseño

### 1er Catálogo

Ámbito	Creación	Estructurales	Comportamiento
Clase	Factory Method	Adapter	Interpreter Template Method
Objeto	Abstract Factory	Adapter	Chain of Responsibility
	Builder	Bridge	Command
	Prototype	Composite	Iterator
	Singleton	Decorator	Mediator
		Facade	Memento
		Flyweight	Observer
		Proxy	State
			Strategy
			Visitor

Arquitectura de Aplicaciones Web – 1°C 2018



## Patrones de Diseño

### Más patrones...

- Data Transfer Object Pattern
- Open Session In View
- Model View Controller
- MV\* Pattern
- Y más...

Arquitectura de Aplicaciones Web – 1°C 2018



## Antipatrones de Diseño

Los antipatrones (antipatterns) son descripciones de situaciones, o soluciones, recurrentes que producen consecuencias negativas. Un antipatrón puede ser el resultado de una decisión equivocada sobre cómo resolver un determinado problema, o bien, la aplicación correcta de un patrón de diseño en el contexto equivocado.

- **Patrón: Problema → Solución**
- **Antipatrón: Solución (problemática) y Solución (refactorizada)**

Arquitectura de Aplicaciones Web – 1°C 2018

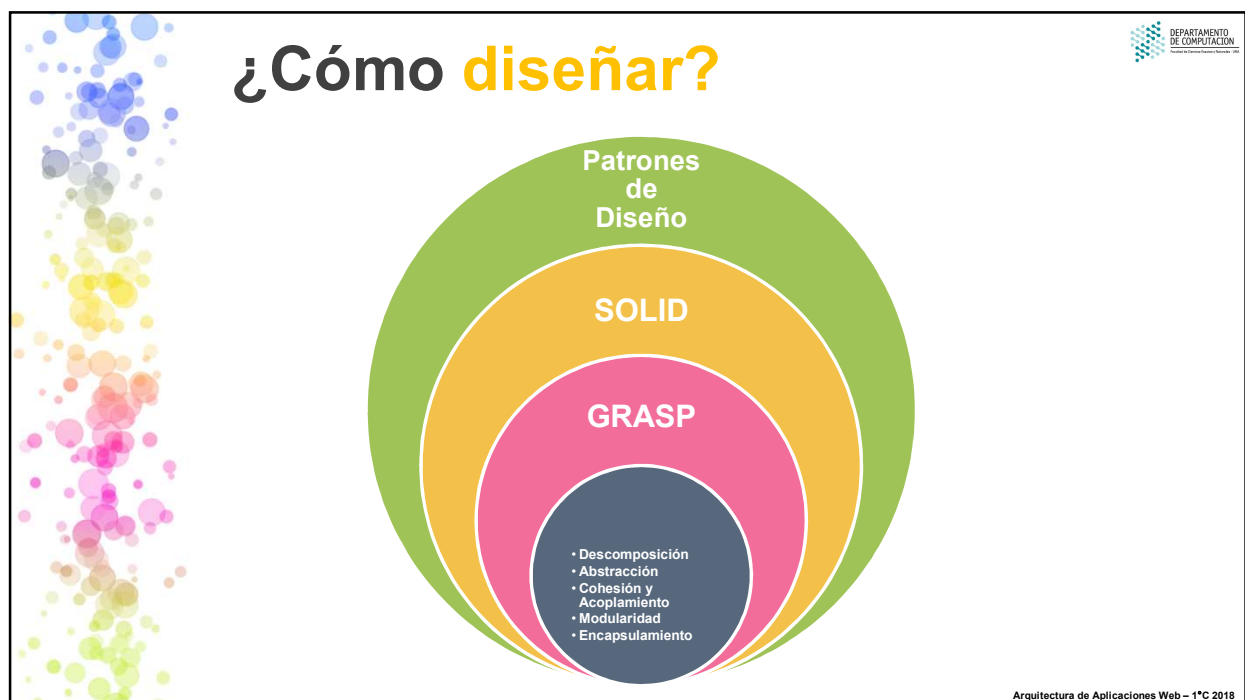
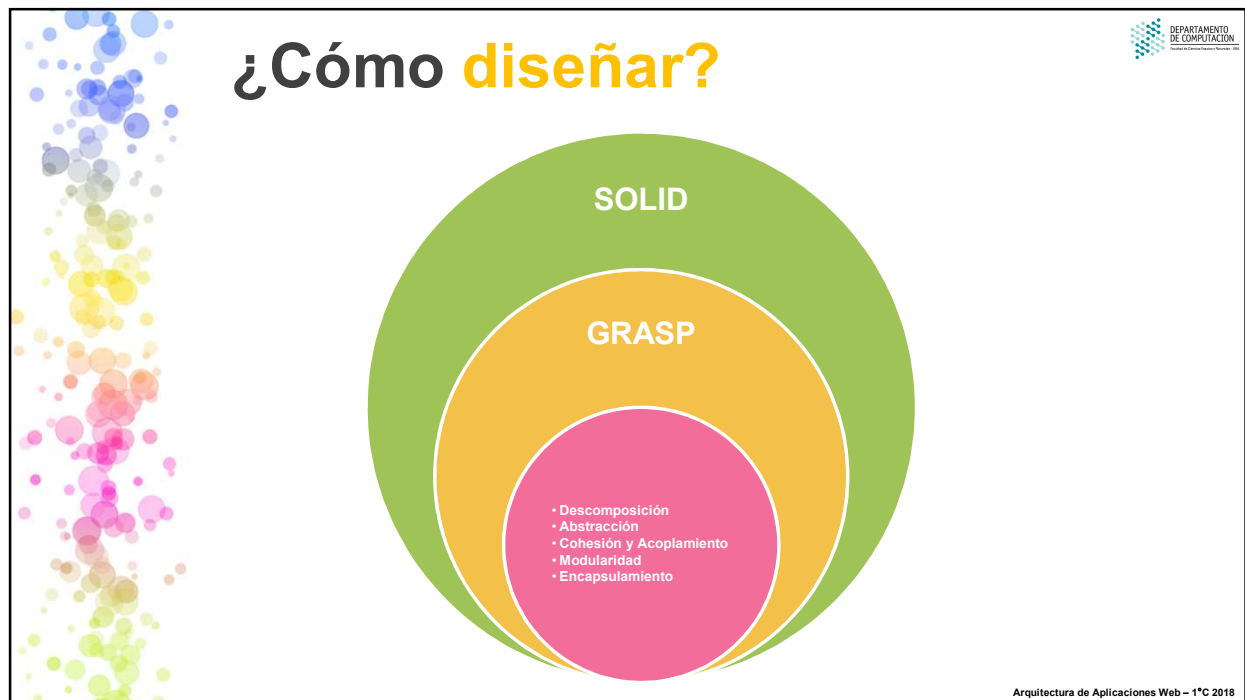


## ¿Cómo evaluar diseños?



- Descomposición
- Abstracción
- Cohesión y Acoplamiento
- Modularidad
- Encapsulamiento

Arquitectura de Aplicaciones Web – 1°C 2018





## A los patrones vamos a volver...

Veremos...

- Patrones para aplicaciones JEE o .NET (basados en sus plataformas)
- Patrones para JavaScripts
- Patrones para aplicaciones en la Nube
  - Habiendo hablado de temas de infraestructura
  - Habiendo hablado de temas relacionados a SOA
- Diferentes tecnologías / frameworks
  - ¿Qué patrones implementan?
  - ¿Sobre que principios se apoyan o hacen foco?

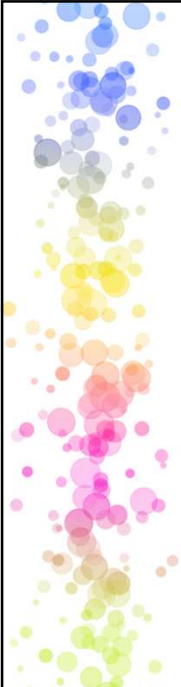
DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018




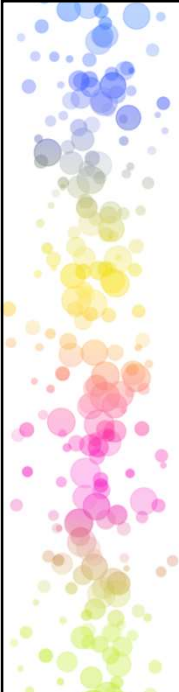
**Lo que viene...** evolución de la web y de las aplicaciones web

DEPARTAMENTO DE COMPUTACION  
Facultad de Ciencias Exactas y Naturales - UBA



**Hablemos del TP**  
Enunciado y Objetivos





## TP Nro 1 - Adelanto

**Objetivo**  
Implementar una aplicación de chat, que satisfaga los siguientes requerimientos (en orden de importancia):


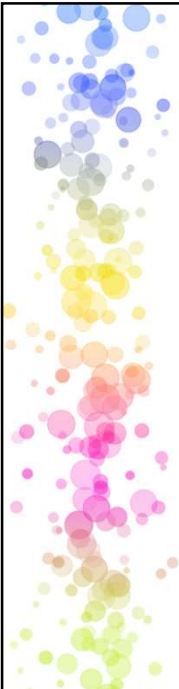
- Permitir que un grupo de usuarios compartan un grupo o salón de chat público.
- Permitir que dos usuarios puedan tener una conversación privada.
- Permitir que un usuario pueda compartir una imagen en un chat (público y/o privado).
- Permitir que un usuario pueda compartir un audio en un chat (público y/o privado).

**Posibles Tecnologías Agrupadas según su generación o tipo de aplicación**  
(lo vemos la próxima clase)

- JSP-Servlets / ASP / ASP.NET / PHP (sin frameworks)
- JSF / ASP.MVC / Ruby on Rails / Spring MVC / Django
- Flex / GWT / Silverlight
- AngularJS / React / Angular5

En este TP no haremos foco en el *backend* (sí en el TP 2).

Arquitectura de Aplicaciones Web – 1°C 2018



## TP Nro 1 - Adelanto

**Objetivo**  
Implementar una aplicación de chat, que satisfaga los siguientes requerimientos (en orden de importancia):

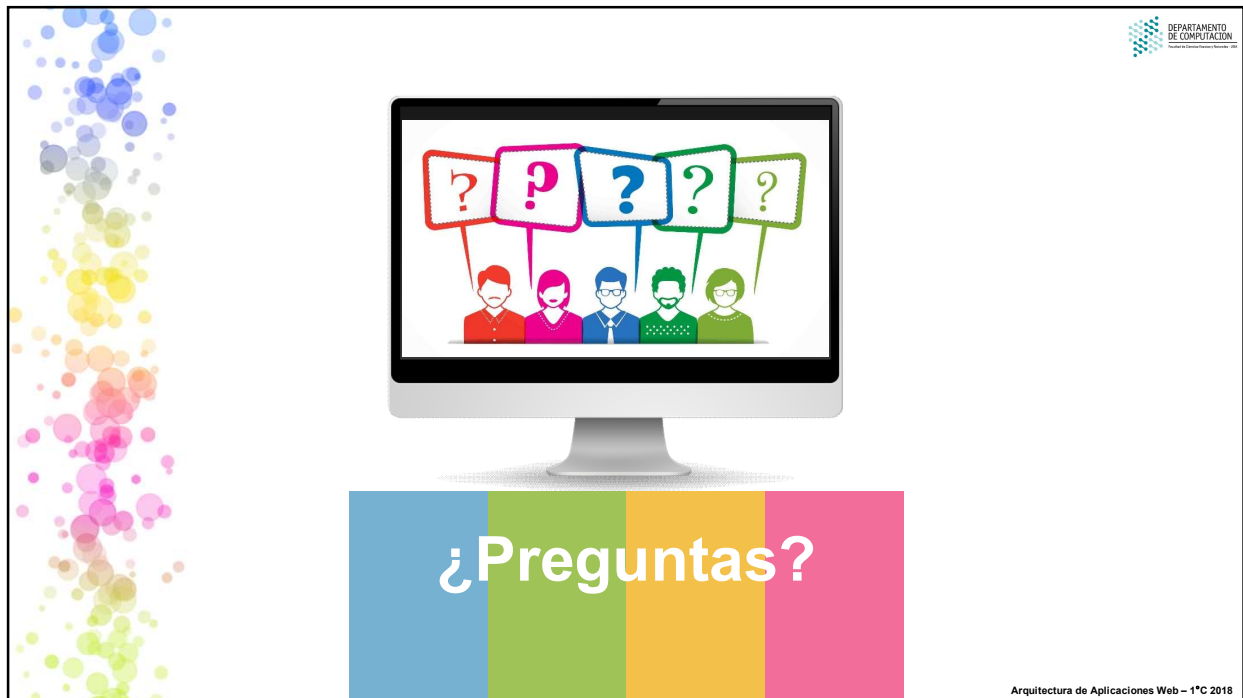
- Permitir que un grupo de usuarios compartan un grupo o salón de chat público.
- Permitir que dos usuarios puedan tener una conversación privada.
- Permitir que un usuario pueda compartir una imagen en un chat (público y/o privado).
- Permitir que un usuario pueda compartir un audio en un chat (público y/o privado).

**Objetivos de la Presentación**

- 30 / 40 minutos máximo.
- Ver demo funcionando
- Describir la arquitectura definida
  - Atributos de calidad considerados
  - Trazabilidad arquitectura - diseño
- Analizar el framework / tecnología utilizada
  - Principios y patrones de diseño
  - Relación con los atributos de calidad
  - Ciclo de vida
  - Puntos de Extensión

Arquitectura de Aplicaciones Web – 1°C 2018





## Bibliografía

- Design Principles and Design Patterns Robert C. Martin [http://www.objectmentor.com/resources/articles/Principles\\_and\\_Patterns.pdf](http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf)
- LARMAN, Craig. Applying UML and Patterns. Prentice Hall. 1998.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns. Addison Wesley. 1994.

DEPARTAMENTO DE COMPUTACIÓN

Arquitectura de Aplicaciones Web – 1°C 2018

