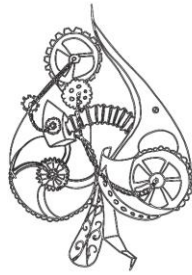
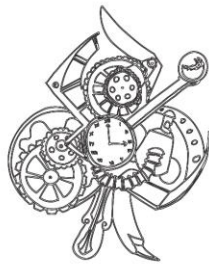


# SISTEMA DE ADMINISTRACIÓN DE CASINOS

- Documento de diseño -



Pablo Esteban Deltell

Miguel Sánchez-Brunete Álvarez

Jonathan Carrero Aranda

Ingeniería del Software

Facultad de Informática

Universidad Complutense de Madrid



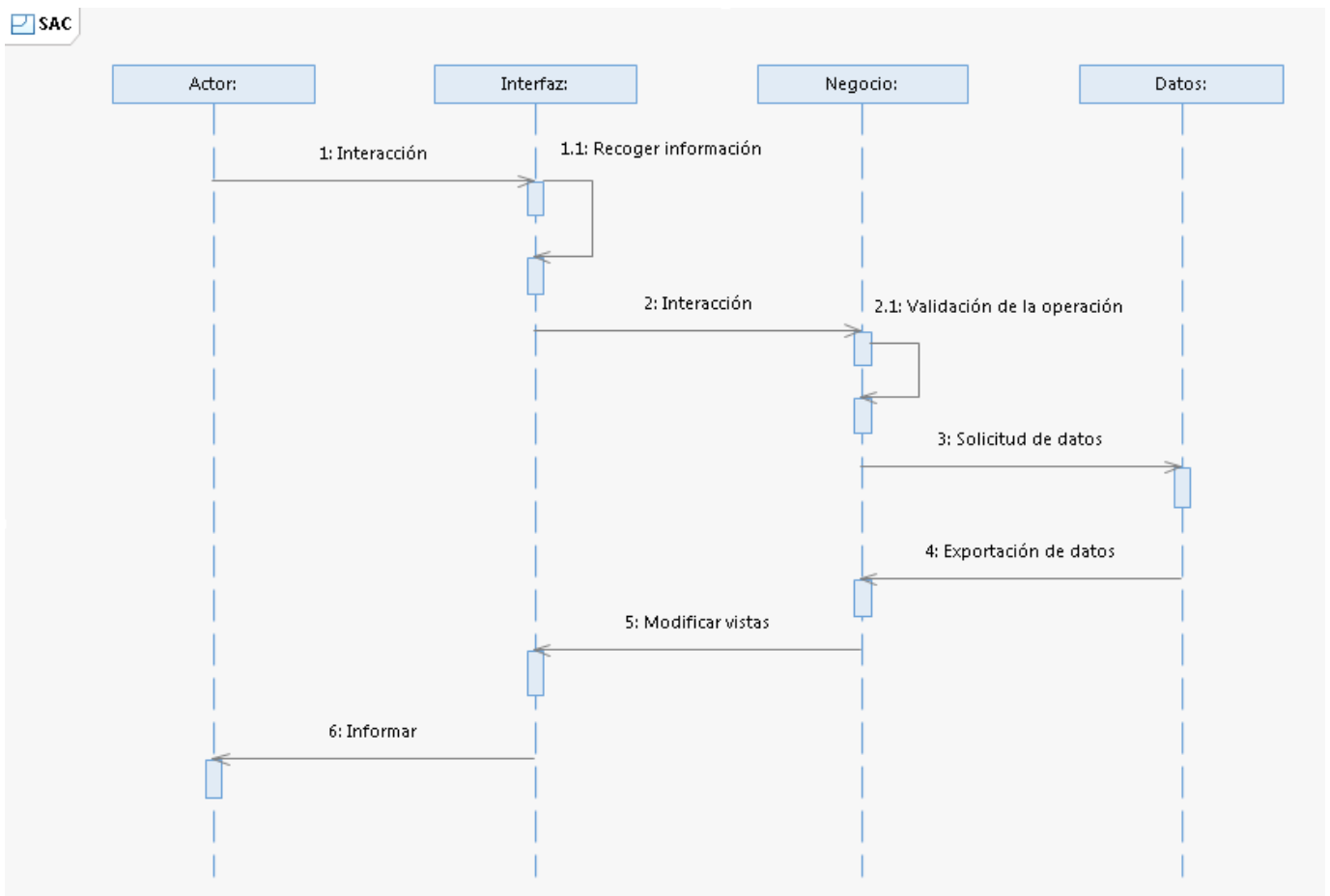
## ÍNDICE

1.	Diagrama de Secuencia por Capas (Multicapa).....	2
2.	Diseño de patrones .....	4
2.1.	Patrón Modelo-Vista-Controlador.....	4
2.2.	Patrón Observer .....	5
2.3.	Patrón Data Access Object.....	7

# 1. Diagrama de Secuencia por Capas (Multicapa)

Como vemos, en nuestro diagrama de secuencia, se representa, paso a paso, la secuencia que sigue cada una de las tres capas del proyecto. En cada una de ellas queda reflejado cómo se transmite la información y los requerimientos que se exigen de una capa a otra. Es importante destacar que estamos hablando de una arquitectura de tres capas:

- I. **Presentación:** la capa de presentación gestiona las interfaces de usuario.
- II. **Negocio:** en esta capa se implementan las reglas de gestión de datos (pero no tiene datos como tal).
- III. **Integración:** es la capa encargada de comunicar los componentes con sus recursos o peticiones externas.



*Diagrama de Secuencia (Sistema de Administración de Casinos)*

Además del diagrama de secuencia, queremos añadir una pequeña explicación sobre qué acción se lleva a cabo en cada uno de los pasos de dicha secuencia. Para un programador con experiencia, puede resultar fácil entender qué hace **3: SOLICITUD DE**

DATOS. Sabemos que nuestro no está destinado (al menos principalmente) a tales personas, por eso creemos necesario incluir el contenido que se expone a continuación:

1. **Interacción:** el Actor, a través de sus distintos métodos, interacciona con la vista obligando a que sea el Controlador quien recoja la acción ejecutada.
  - 1.1. **Recoger información:** identifica y recoge la información del Actor, porque en función del Actor (no es lo mismo Cliente que Crupier) ejecutará y pedirá un tipo de datos que pueden llegar a ser totalmente distintos.
2. **Interacción:** captura la “señal” emitida por el Actor a través de la Interfaz.
  - 2.1. **Validación de la operación:** comprueba que la operación que realizará el Actor es una operación correcta (por ejemplo, un crupier no puede solicitar una operación que haga referencia a las acciones del gerente).
3. **Solicitud de datos:** solicitud de datos al Modelo (consultas a la BBDD, modificación de registros, operaciones referentes a las apuestas...).
4. **Exportación de datos:** los datos solicitados son exportados hacia la lógica de negocio.
5. **Modificar vistas:** la lógica de negocio tiene los datos, por lo tanto, el siguiente paso es modificar las vistas de cara al Actor.
6. **Informar:** por último, el Actor recibe la información por parte de la Interfaz verificando que, en efecto, se ha realizado con éxito la interacción que se solicitó en PUNTO 1.

## 2. Diseño de patrones

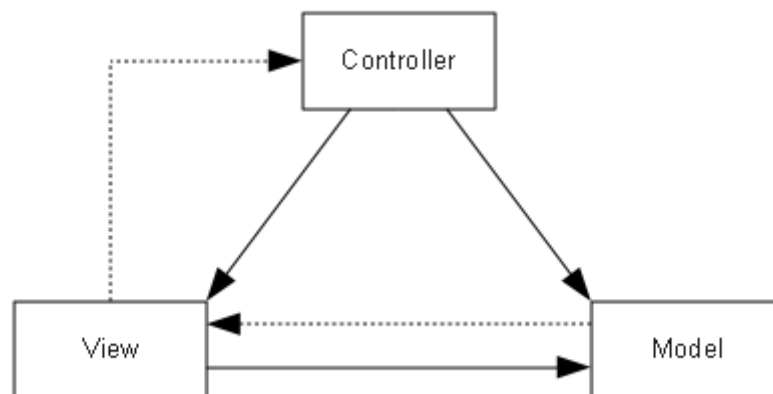
---

Durante el desarrollo de la aplicación se han llevado a cabo, principalmente, tres patrones de diseño. Estos tres patrones conforman y refuerzan completamente la estructura de clases que se tiene a día de hoy. Si bien es cierto que podrían haberse implementado patrones tan sencillos como Decorator, su uso no aporta un gran avance o mejoramiento en el resultado final.

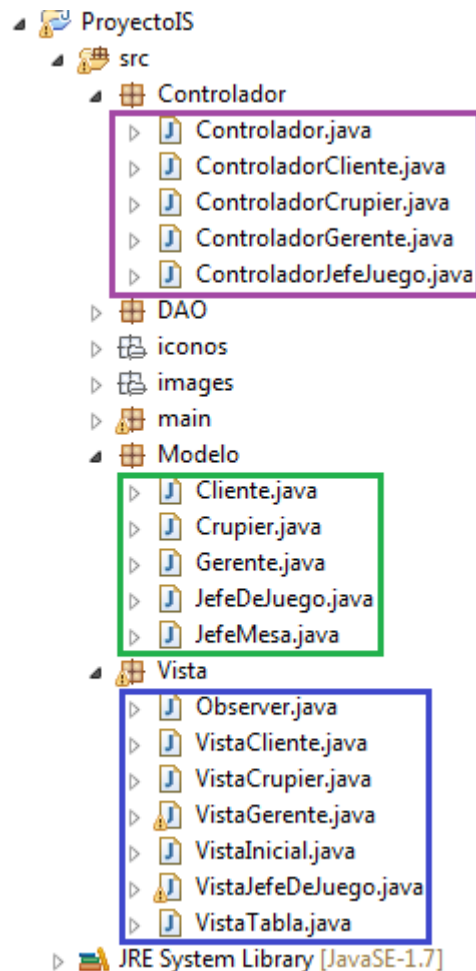
### 2.1. Patrón Modelo-Vista-Controlador

El patrón de arquitectura MVC (**M**odelo **V**ista **C**ontrolador) es un patrón que define la organización independiente del Modelo (lógica de negocio), la Vista (interfaz con el usuario u otro sistema) y el Controlador.

Para entender cómo funciona nuestro patrón Modelo vista controlador, se debe entender la división del conjunto de estos tres elementos y como estos componentes se comunican unos con los otros a través de un flujo de ejecución impuesto por el propio patrón.



**Figura 2.1.1:** flujo de ejecución en el MVC.



**Figura 2.1.2:** patrón MVC en la aplicación SAC. En ella pueden distinguirse los tres aspectos fundamentales: *Modelo*, *Vista*, *Controlador*.

## 2.2. Patrón Observer

La idea principal detrás del patrón Observer es que existe una entidad con estados cambiantes y una o más entidades observándola. Los observadores esperan a que la entidad observada les informe de un cambio de estado a través de un *evento* que puede ser de su interés, por lo que los observadores se registran con la entidad observada.

Cuando ocurre un evento, la entidad observada mira en su lista de observadores y notifica a aquellos que se registraron para recibir eventos de ese tipo. Por tanto, los observadores dejaron instrucciones detalladas de cómo puede la entidad observada ponerse en contacto con ellos para recibir los eventos.

```

Observer.java
1 package Vista;
2
3 public interface Observer {
4
5     // Gestion de juego
6     void onReset(String juego, int mesa);
7     void onCambioJuego();
8
9 }
10

```

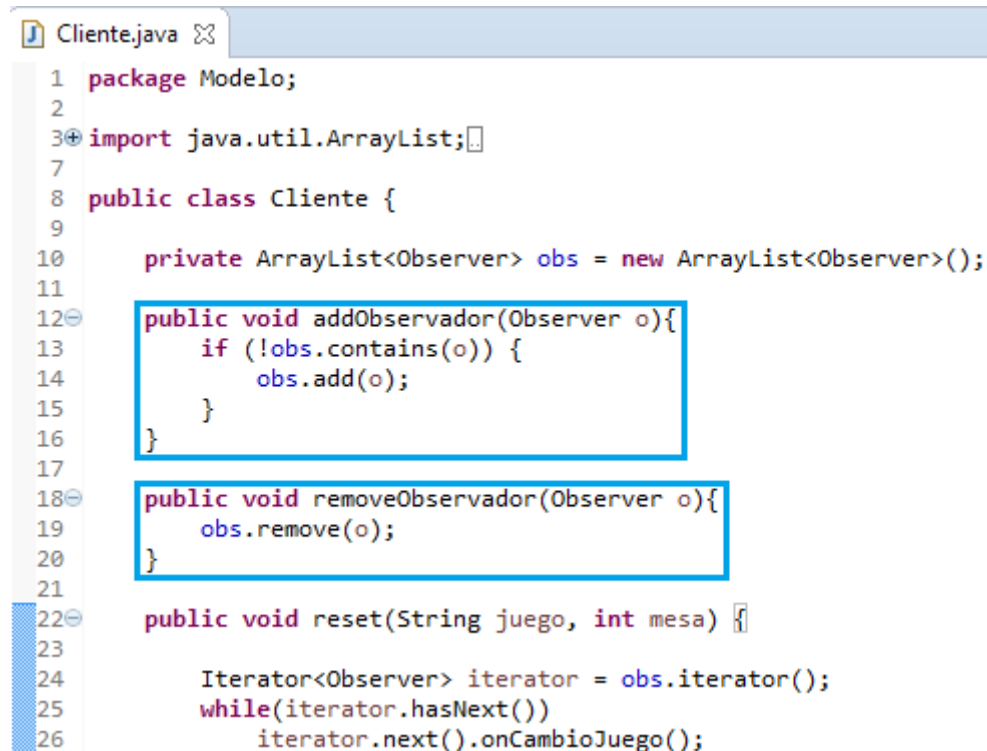
**Figura 2.2.1:** implementación de la interfaz Observer.

```

VistaCliente.java
1 package Vista;
2
3 import java.awt.*;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23 public class VistaCliente extends JFrame implements Observer{
24
25     private static final long serialVersionUID = 6937853572224085296L;
26
27     private static int DEFAULT_WIDTH = 700;
28     private static int DEFAULT_HEIGHT = 500;
29
30     private ControladorCliente ctrl;
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

**Figura 2.2.2:** tanto la clase VistaCliente como las demás clases pertenecientes a las vistas implementan la interfaz Observer.



```

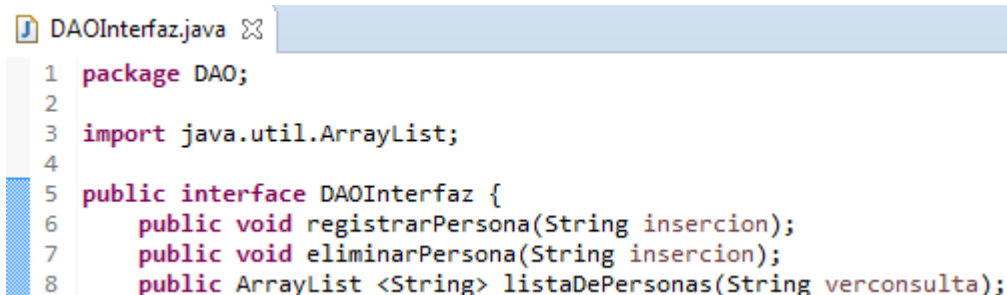
1 package Modelo;
2
3 import java.util.ArrayList;
4
5 public class Cliente {
6
7     private ArrayList<Observer> obs = new ArrayList<Observer>();
8
9     public void addObservador(Observer o){
10         if (!obs.contains(o)) {
11             obs.add(o);
12         }
13     }
14
15     public void removeObservador(Observer o){
16         obs.remove(o);
17     }
18
19     public void reset(String juego, int mesa) {
20         Iterator<Observer> iterator = obs.iterator();
21         while(iterator.hasNext())
22             iterator.next().onCambioJuego();
23     }
24 }

```

**Figura 2.2.3:** en el Modelo se añaden dos métodos, uno para añadir observador y otro para eliminarlo.

## 2.3. Patrón Data Access Object

El patrón DAO (Data Access Object) es un patrón de diseño estructural que actúa como una interfaz a la hora de conectarse a la base de datos. La principal ventaja es que no hace falta que se sepa qué tipo de base de datos es, por lo que a la hora de tener que pasar a otro sistema de base de datos es mucho más sencillo y más rápido.



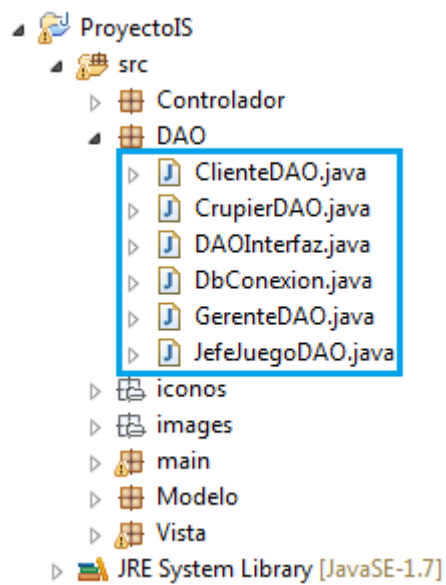
```

1 package DAO;
2
3 import java.util.ArrayList;
4
5 public interface DAOInterfaz {
6     public void registrarPersona(String insercion);
7     public void eliminarPersona(String insercion);
8     public ArrayList <String> listaDePersonas(String verconsulta);
9 }

```

**Figura 2.3.1:** implementación de la interfaz DAO





**Figura 2.3.2:** *clases DAO en la aplicación.*