

Modelado de objetos - Patrones de diseño



UML | Modelado

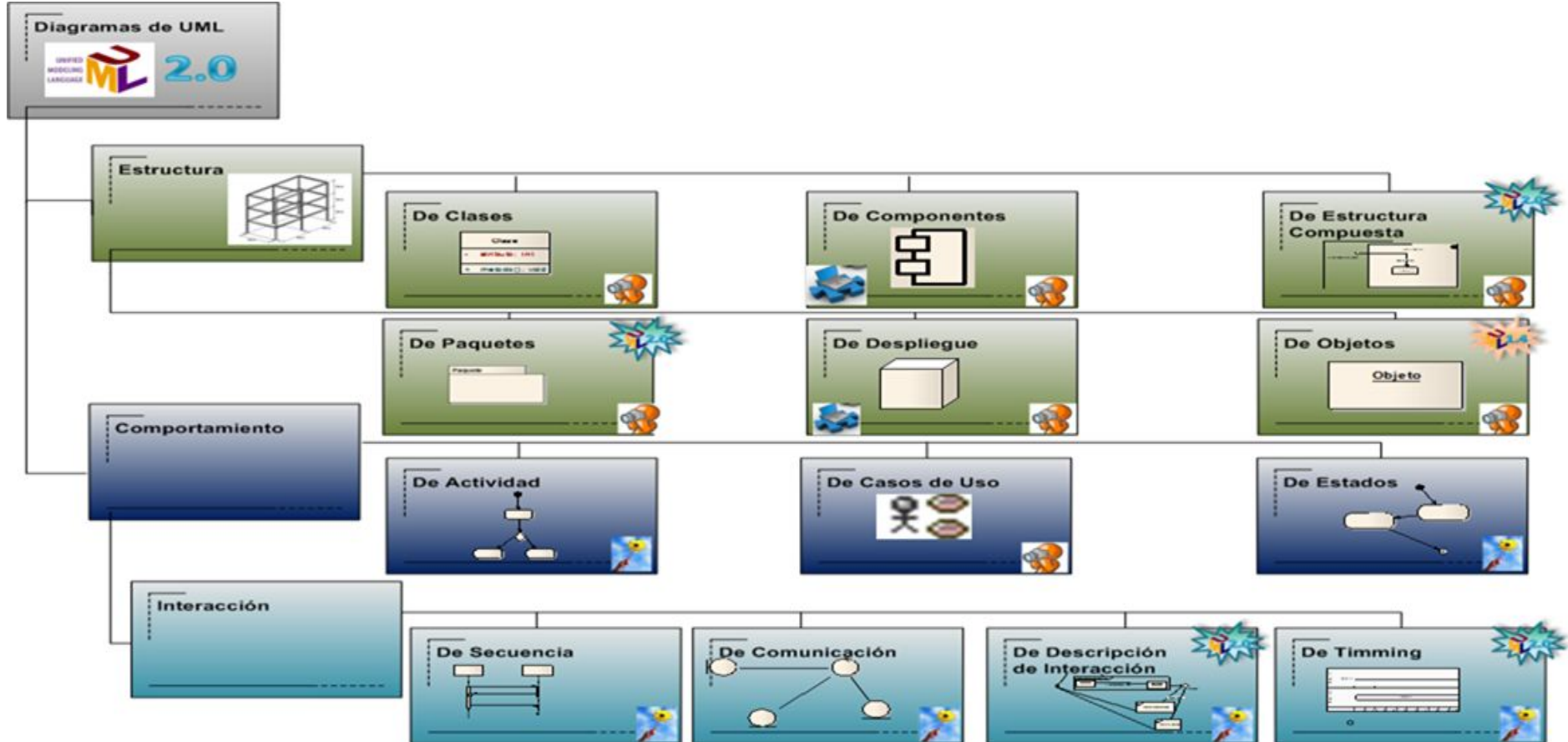


Cuando modelamos creamos una simplificación de la realidad para comprender mejor el sistema que se está desarrollando. UML usa varios diagramas para representar el sistema.

- **Diagramas Estructurales**
- **Diagramas de comportamiento**



UML | Modelado





UML 2.5.1 | Diagramas Estructurales

Cuando se modelan sistemas las *partes estáticas de un sistema* se representarán mediante uno de los diagramas siguientes

- Diagramas de clases
- Diagramas de paquetes
- Diagramas de componentes
- Diagramas de perfiles
- Diagramas de estructura compuesta
- Diagramas de objetos
- Diagramas de despliegue



UML | Diagrama de Clases



Se utiliza el diagrama UML para representar las clases, atributos, operaciones, y la relación entre cada clase. Las clases se agrupan para crear diagramas de clases al crear diagramas de sistemas grandes.

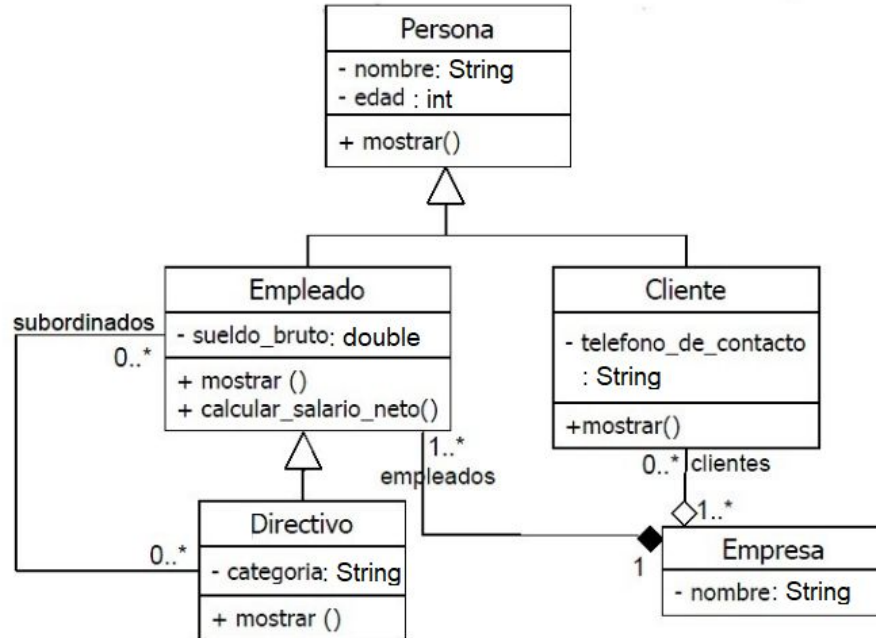
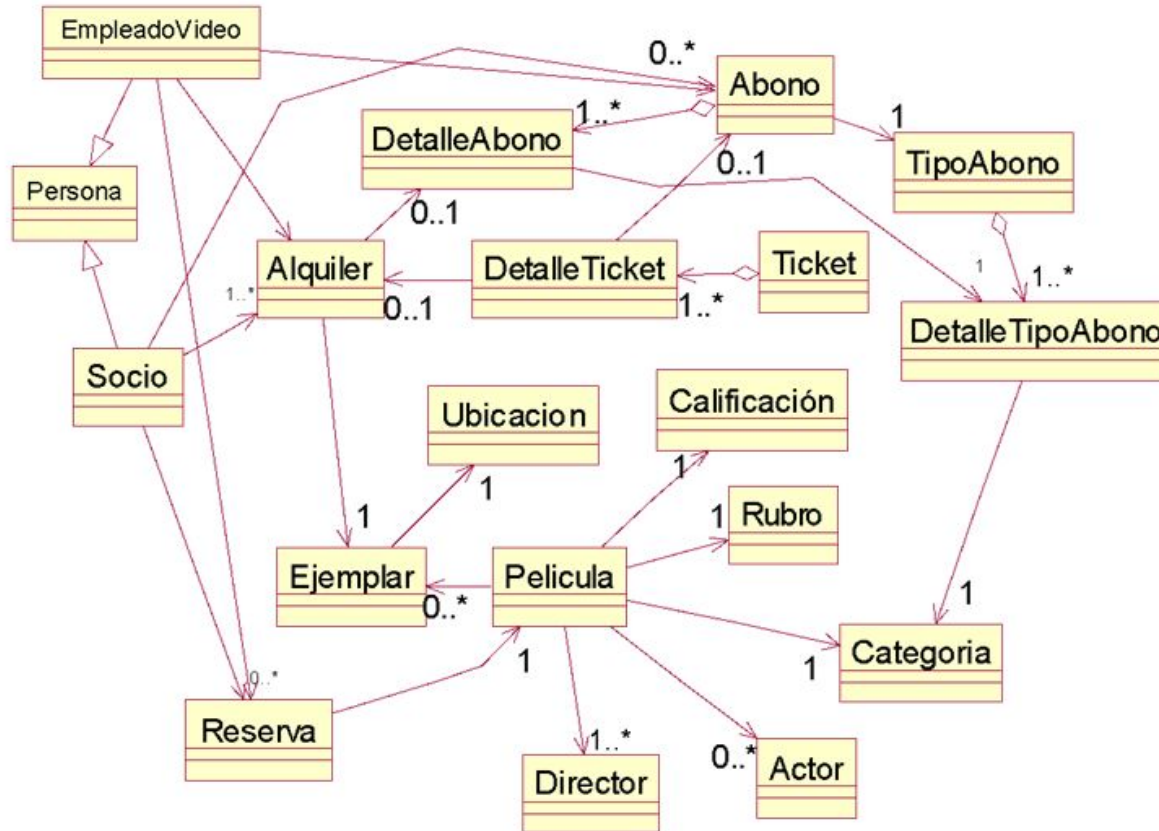


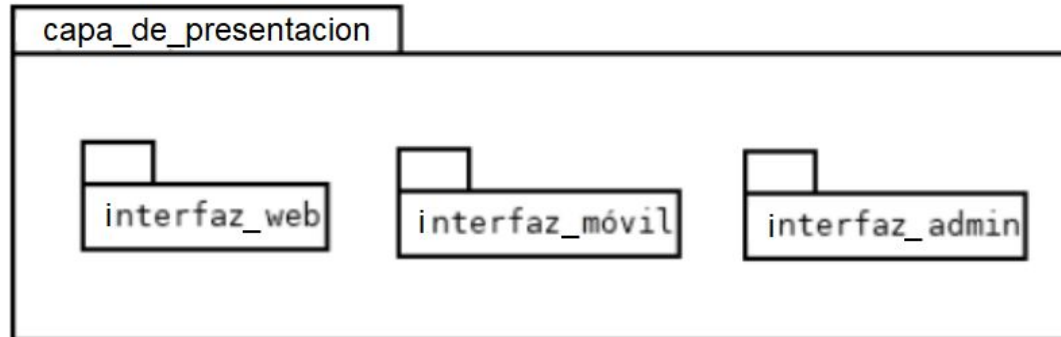
Diagrama de Clases | Ejemplo



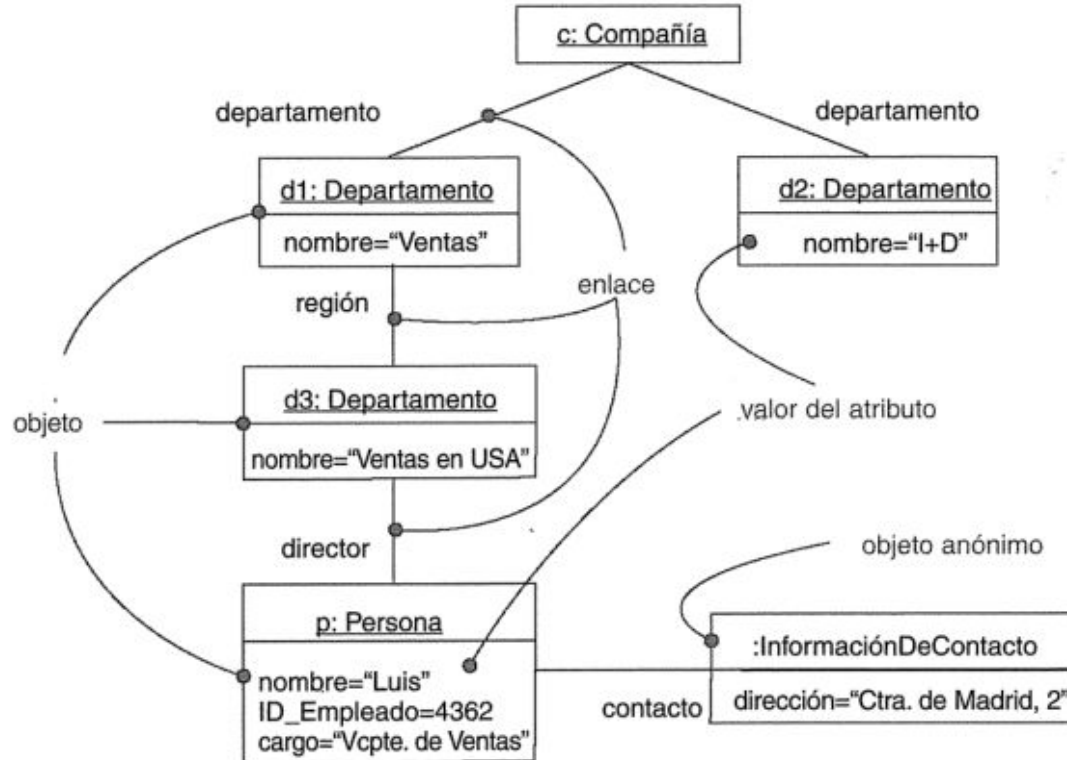
UML | Diagrama de paquetes



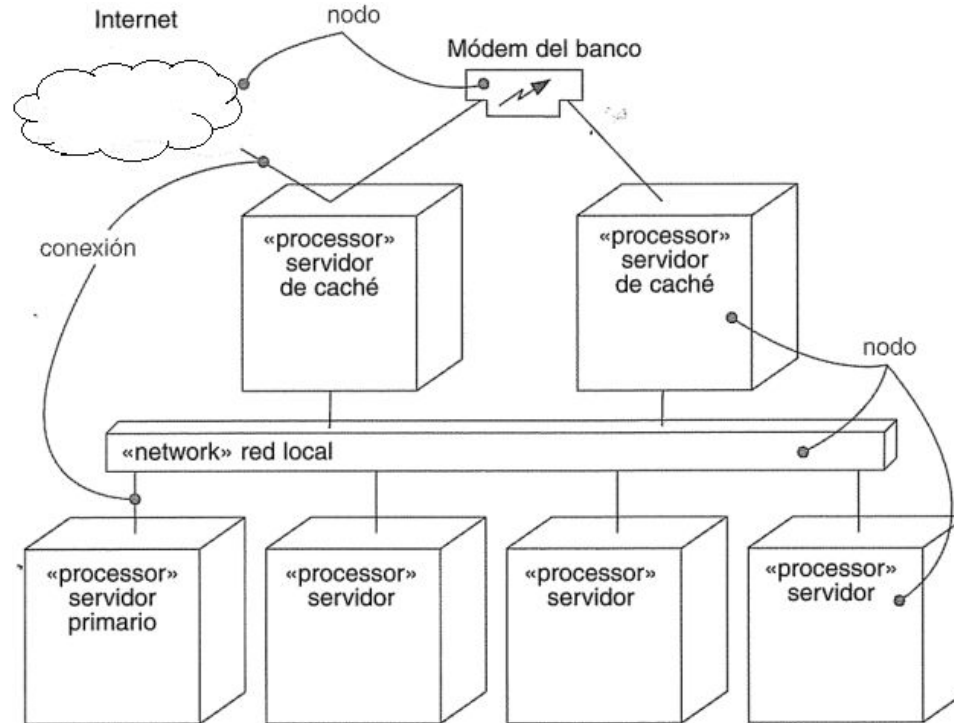
Es utilizado para definir los paquetes a nivel lógico que forman parte de la aplicación y la dependencia entre ellos.



UML | Diagramas de objetos



UML | Diagramas de despliegue



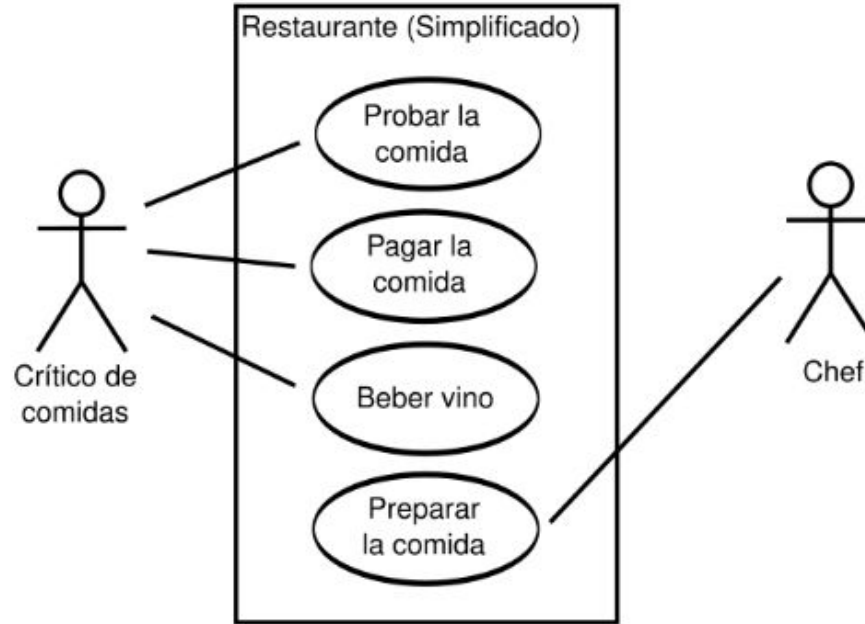
UML | Diagramas de comportamiento



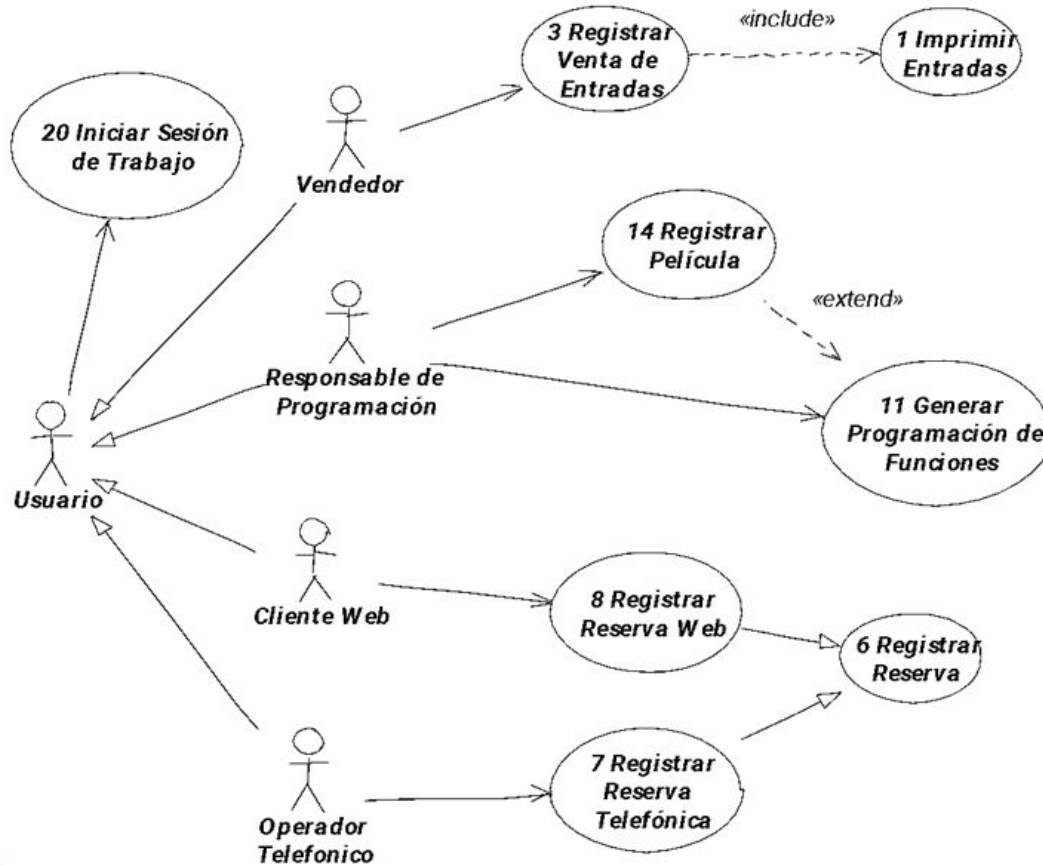
- ❑ Diagramas de casos de uso
- ❑ Diagramas de máquina de estados
- ❑ Diagramas de actividades
- ❑ Diagramas de interacción:
 - ❑ Diagramas de secuencia,
 - ❑ Diagramas de comunicación,
 - ❑ Diagramas de tiempos y
 - ❑ Diagrama global de interacciones.



UML | Diagramas de casos de uso



Diagramas de casos de uso | Ejemplo Cine



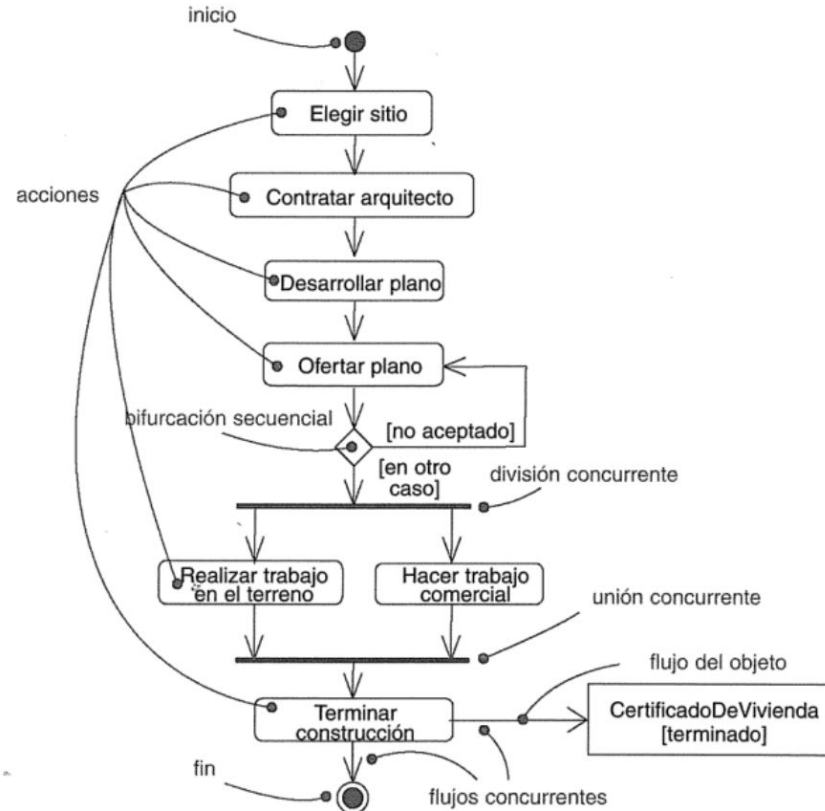
D. Casos de uso | Descripción de actores



Nombre del Actor	Descripción
Cliente Web	Persona que tiene acceso a una computadora con conexión a Internet y puede entrar a la página Web del Complejo de Cines para realizar consultas de programación y reservas para funciones de los diferentes cines del complejo.
Responsable de Programación	Persona que administra toda la funcionalidad vinculada con la obtención de la programación de funciones y la registración de nuevas películas que integrarán las programaciones del complejo.
Vendedor	Persona responsable de la registración y anulación de ventas de entradas para las funciones habilitadas en los cines del complejo.
Operador Telefónico	Persona responsable de la creación consulta y modificación de reservas de lugares para funciones de los cines del complejo
Operador de Reserva	Responsable de la registración y /o modificación de reservas de lugares en una o más funciones de cine del complejo.
Usuario	Persona que está definida como usuario de la aplicación y va a registrarse en la misma para realizar alguna actividad.



UML | Diagramas de actividades



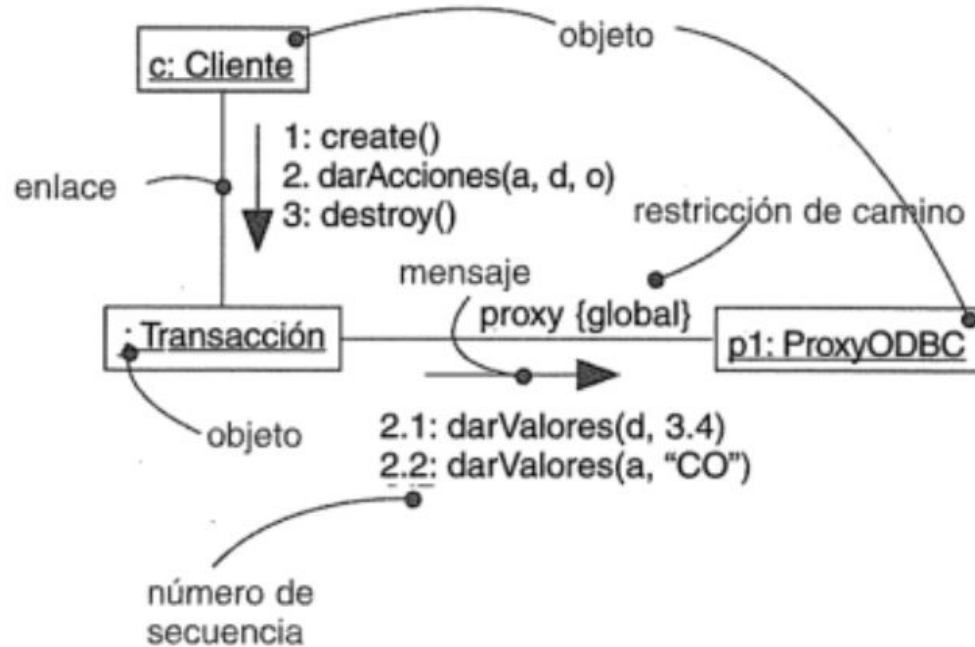
UML | Diagramas de interacción



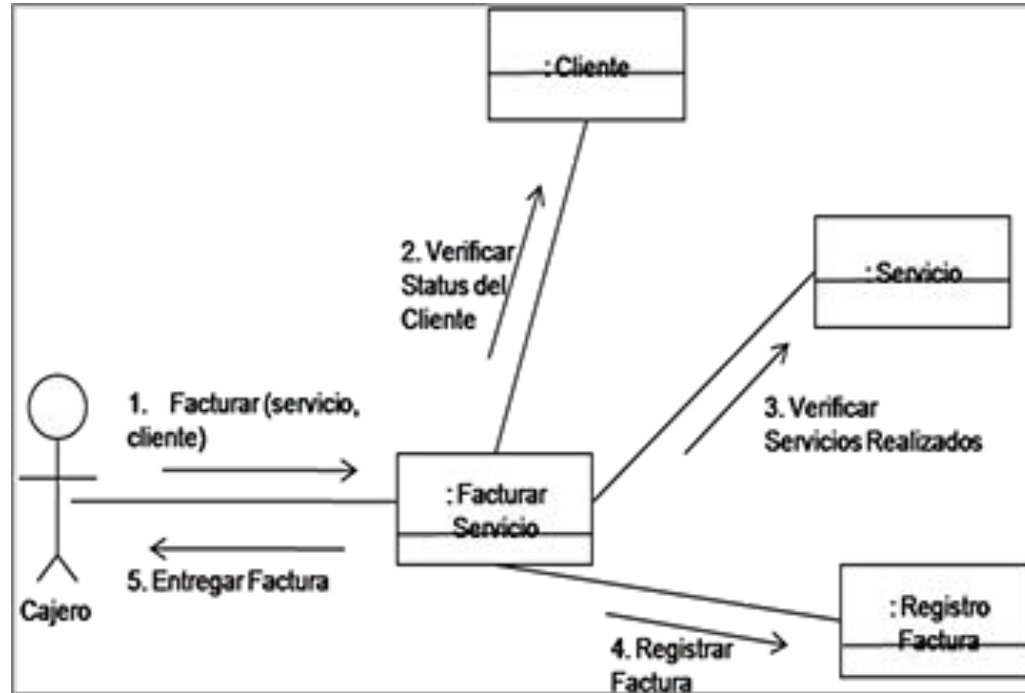
- ❑ *Diagramas de secuencia*
- ❑ *Diagramas de comunicación*
- ❑ *Diagramas de tiempos*
- ❑ *Diagramas global de interacciones*



UML | Diagramas de comunicación



UML | Diagramas global de interacciones



Diseño de Software | Patrones de diseño



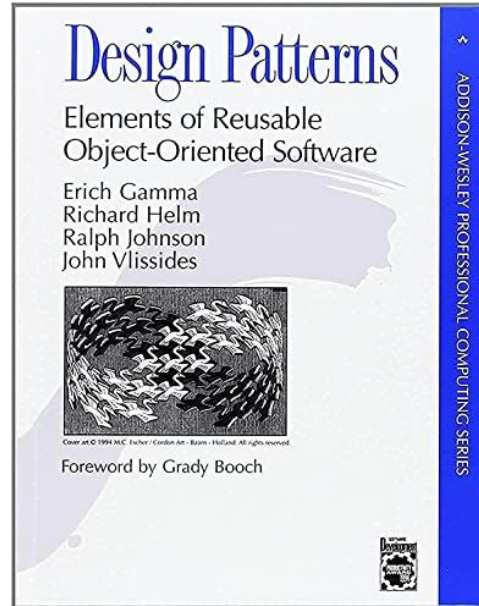
- Solución probada a un problema recurrente.
- Solución reutilizable a un problema común en un contexto dado.
- Cada patrón de diseño tiene sus ventajas y desventajas frente a un determinado problema.
- Ahorran tiempo.



Diseño de Software | Patrones de diseño



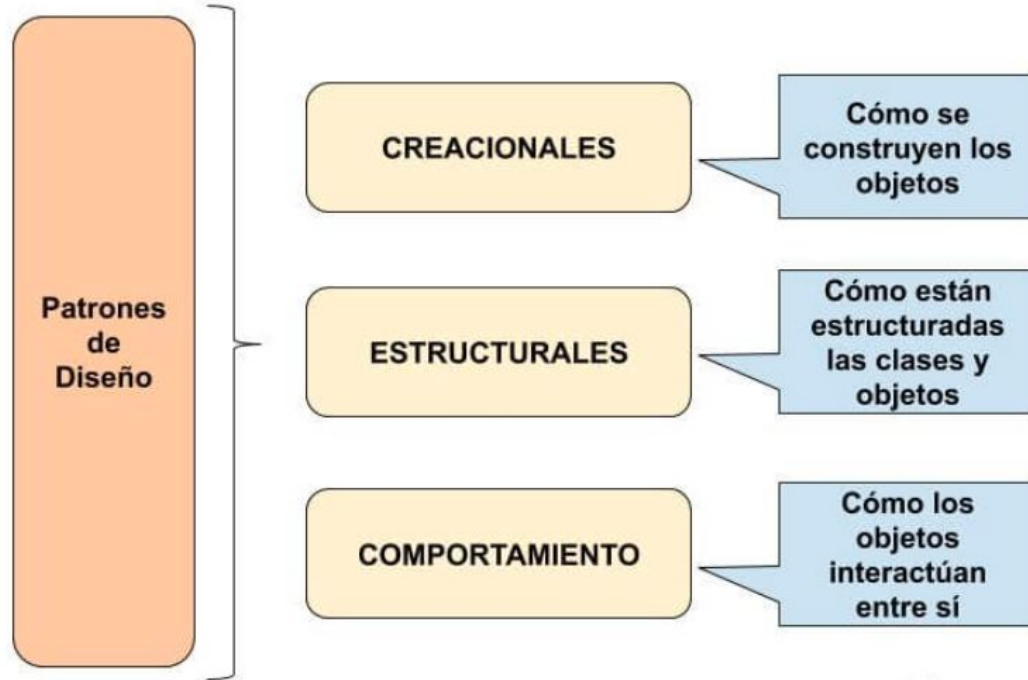
Cobra fuerza en los 90 's cuando “la banda de los cuatro”, publicó el libro “*Design Patterns - Elements of Reusable Object-Oriented Software*” donde se explican 23 patrones de diseño que luego se convirtieron en referentes en el desarrollo de software.



1995



Patrones de diseño | Clasificación



Patrones de Diseño | Ejemplos

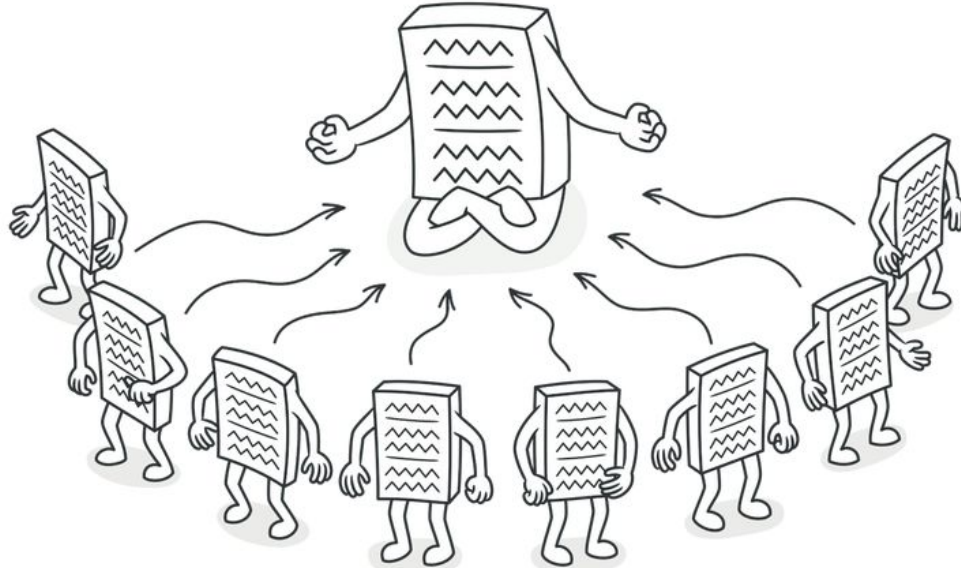


Propósito				
		De Creación	Estructurales	De Comportamiento
Ámbito	Clase	<ul style="list-style-type: none">• Factory Method	<ul style="list-style-type: none">• Adapter (de clases)	<ul style="list-style-type: none">• Interpreter• Template Method
	Objeto	<ul style="list-style-type: none">• Abstract Factory• Builder• Prototype• Singleton	<ul style="list-style-type: none">• Adapter (de objetos)• Bridge• Composite• Decorator• Façade• Flyweight• Proxy	<ul style="list-style-type: none">• Chain of Responsibility• Command• Iterator• Mediator• Memento• Observer• State• Strategy• Visitor

Patrones de Diseño | Singleton



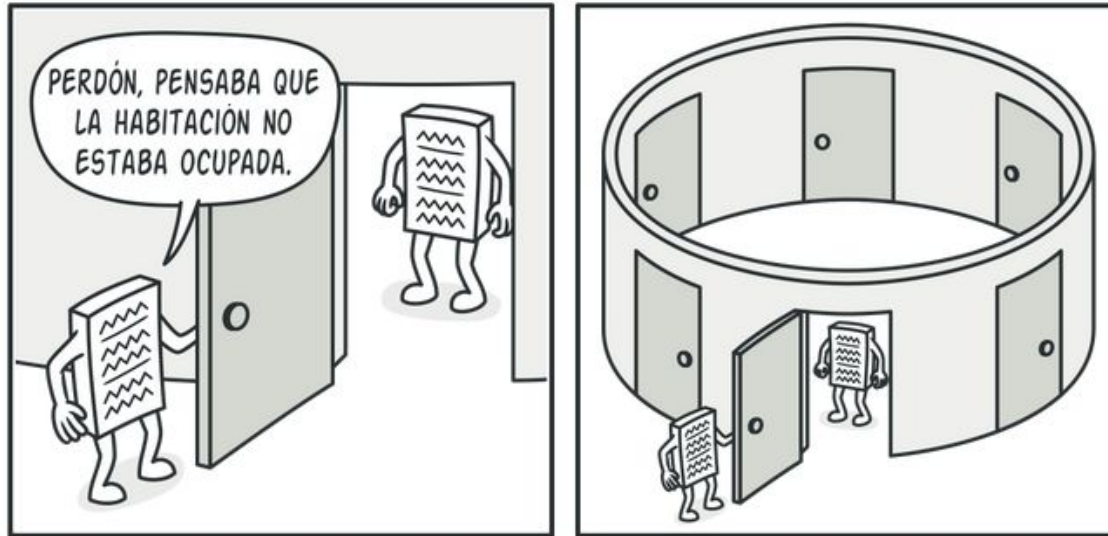
Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.



Patrones de Diseño | Singleton



Garantizar que una clase tenga una única instancia. ¿Por qué querría alguien controlar cuántas instancias tiene una clase? El motivo más habitual es controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo.



Patrones de Diseño | Singleton



Todas las implementaciones del patrón Singleton tienen estos dos pasos en común:

- Hacer *privado* el constructor por defecto para evitar que otros objetos utilicen el operador `new` con la clase Singleton.
- Crear un método de creación estático que actúe como constructor. Este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Si tu código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.

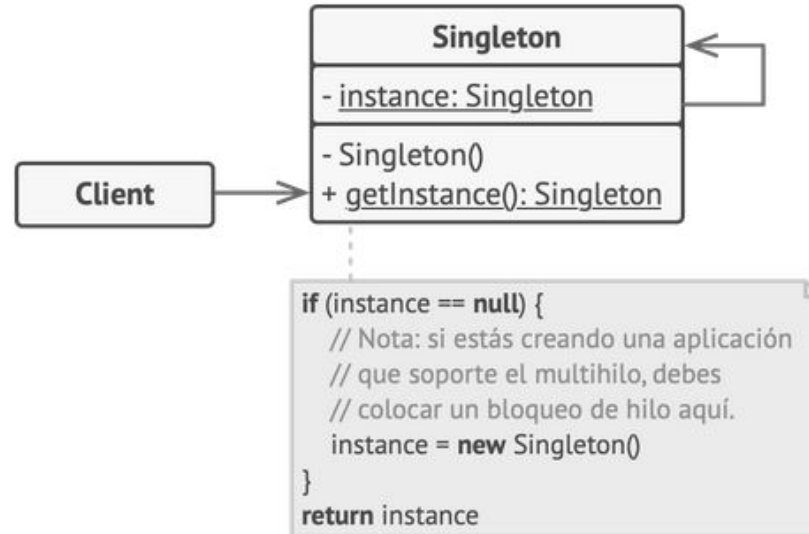


Singleton | Estructura




La clase **Singleton** declara el método estático `getInstance` (obtener instancia) que devuelve la misma instancia de su propia clase.


El constructor del Singleton debe ocultarse del código cliente. La llamada al método `obtenerInstancia` debe ser la única manera de obtener el objeto de Singleton.



Singleton | Aplicabilidad



 Utiliza el patrón Singleton cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes; por ejemplo, un único objeto de base de datos compartido por distintas partes del programa.

 El patrón Singleton deshabilita el resto de las maneras de crear objetos de una clase, excepto el método especial de creación. Este método crea un nuevo objeto, o bien devuelve uno existente si ya ha sido creado.

 Utiliza el patrón Singleton cuando necesites un control más estricto de las variables globales.

Fuente: <https://refactoring.guru/es/design-patterns/>



Singleton | Ejemplo



```
1 package principal;
2 public class Conexion {
3     private static Conexion instancia;
4
5     private Conexion() {
6
7     }
8
9     public static Conexion getInstancia() {
10         if (instancia == null) {
11             instancia = new Conexion();
12         }
13         return instancia;
14     }
15
16     public void conectar() { System.out.println("Me conecté a la BD"); }
17
18     public void desconectar() { System.out.println("Me desconecté de la BD"); }
19 }
```



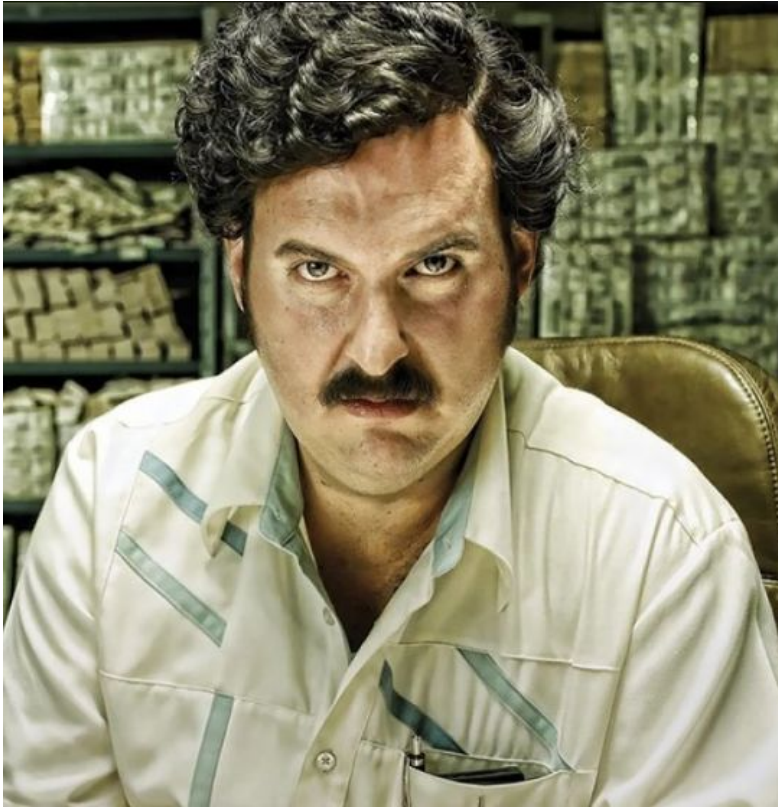
Singleton | Ejemplo



```
1  package principal;
2
3  public class Principal {
4
5      public static void main(String[] args) {
6          //Conexion c = new Conexion(); Instanciación prohibida
7          Conexion c = Conexion.getInstance();
8          c.conectar();
9          c.desconectar();
10
11          boolean rpta = c instanceof Conexion;
12          System.out.println(rpta);
13      }
14 }
```



Patrones de Diseño | Que pasa si no los uso



- Le encuentro bugs.
- Errores de diseño.
- Y si su programa funciona, hago que el cliente le cambie los requerimientos.