





# **PATRONES DE DISEÑO - IMPLEMENTACIÓN**

**Unidad III - Semana 5**

# LOGRO DE LA SESIÓN



*Al término de la sesión de aprendizaje el estudiante aprende el uso de los patrones de diseño, diagrama de clases de diseño con su implementación respectiva.*

# AGENDA



- ❖ Modelo de datos (Modelo de persistencia).
- ❖ Actividad de aprendizaje.
- ❖ Patrones de Diseño.
- ❖ Implementación de Patrones de Diseño
- ❖ Caso propuesto.



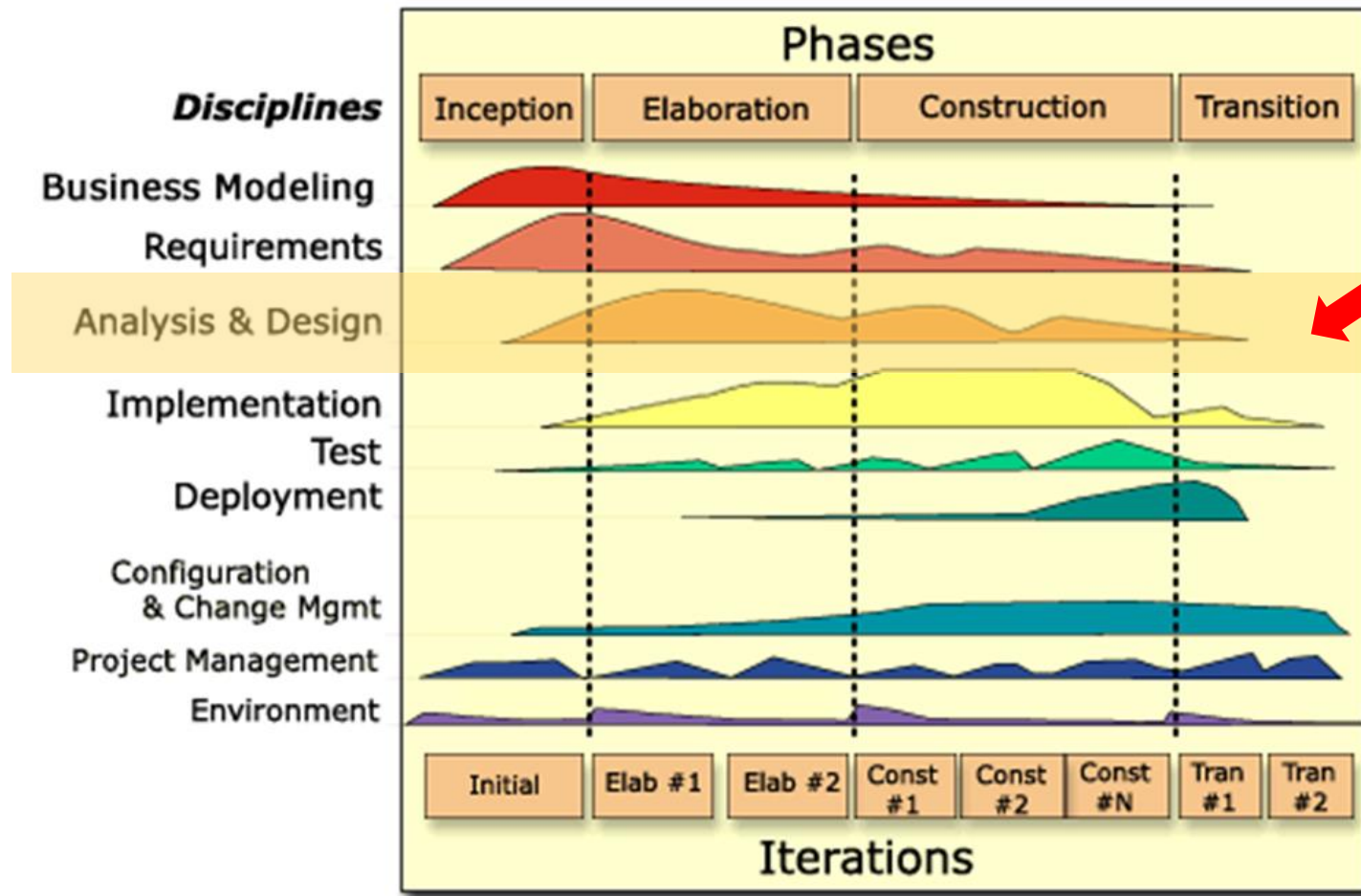


# Patrones de Diseño

**Introducción a los Patrones de  
Diseño**

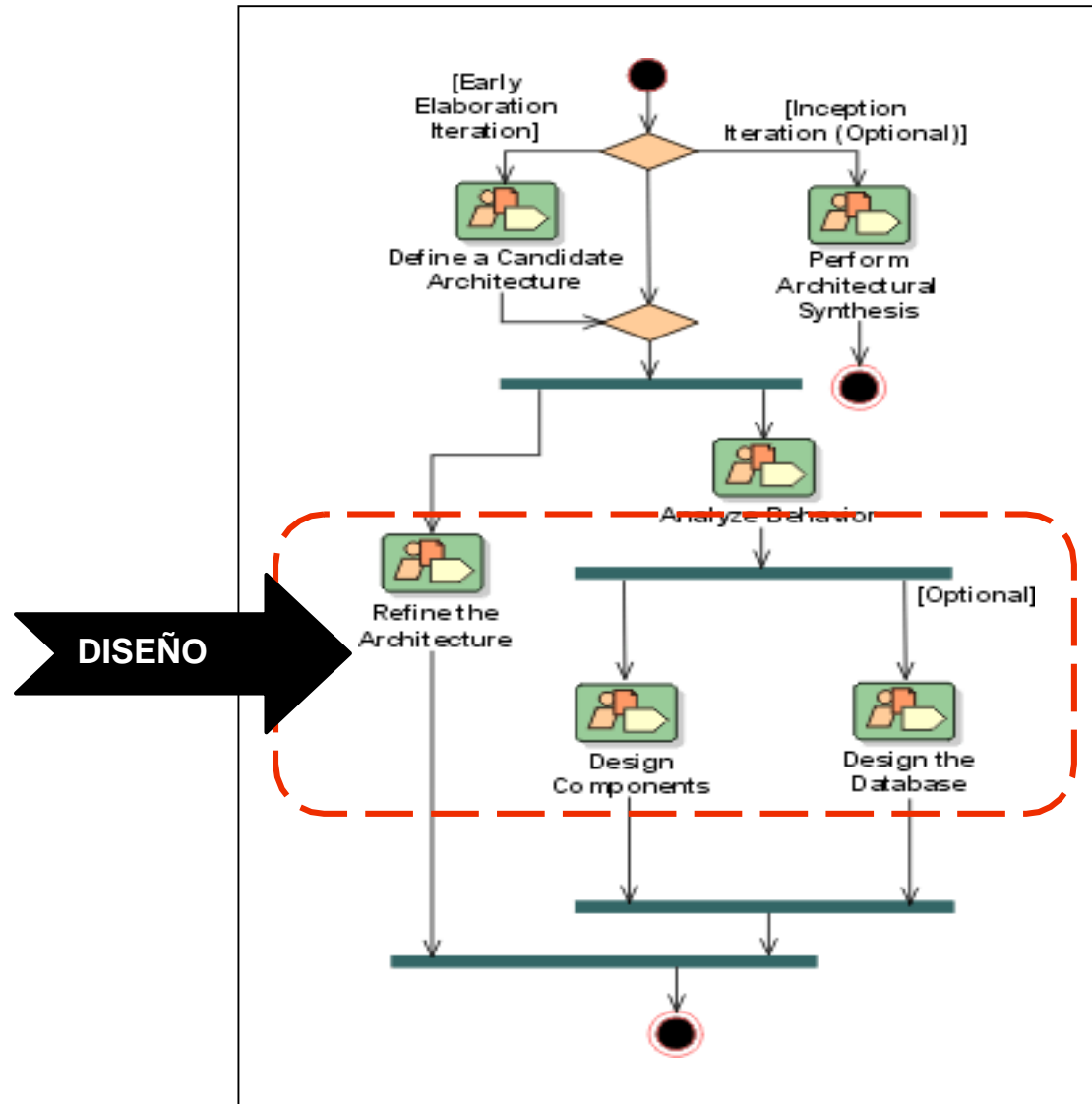
# DISCIPLINA ANÁLISIS Y DISEÑO (RUP)

## Tercera Disciplina



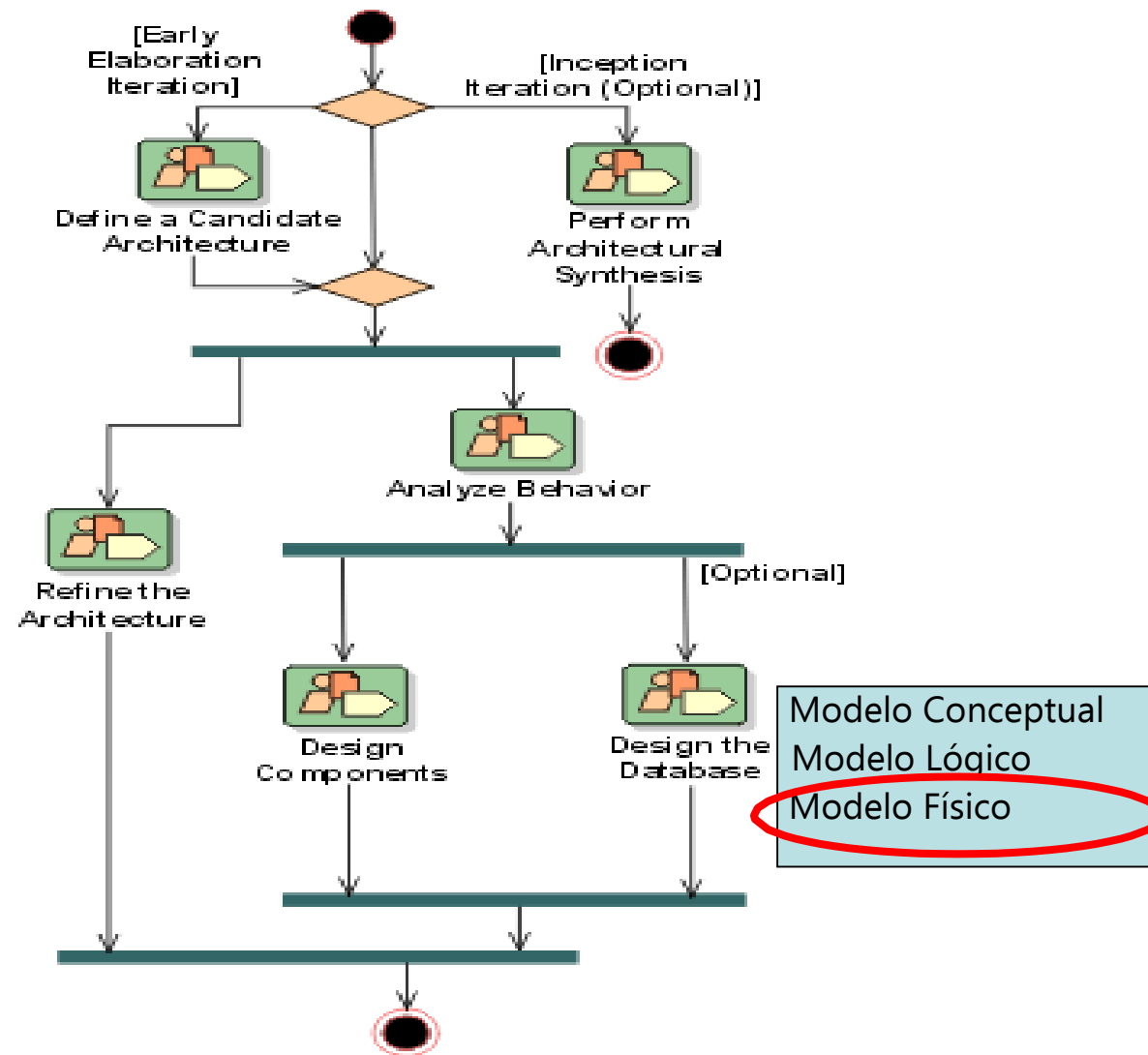
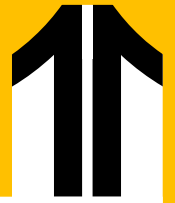
# DISCIPLINA ANÁLISIS Y DISEÑO (RUP)

## Flujo de Trabajo del Análisis y Diseño



# DISCIPLINA ANÁLISIS Y DISEÑO (RUP)

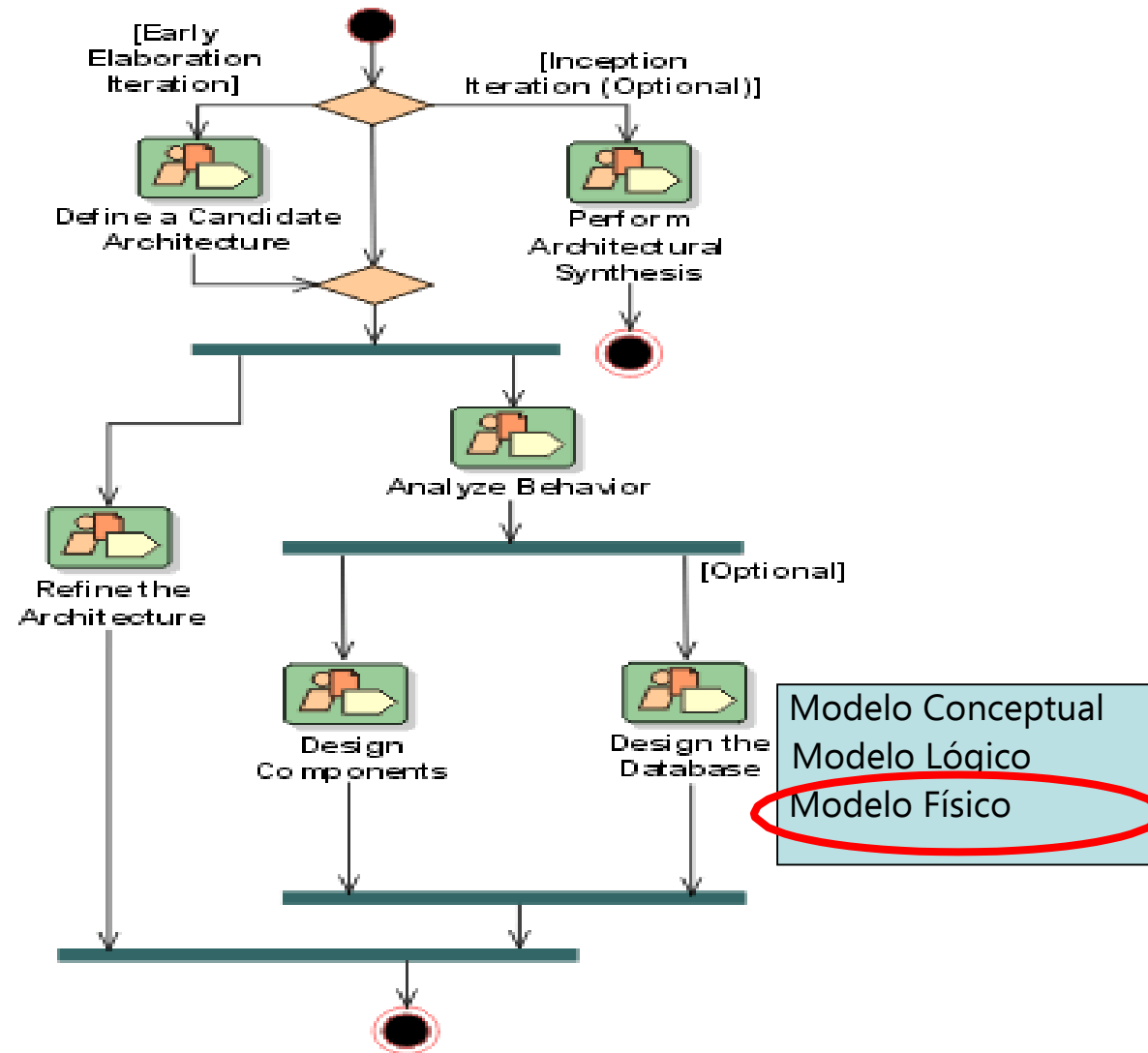
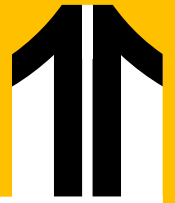
## Flujo de Trabajo del Análisis y Diseño





# DISCIPLINA ANÁLISIS Y DISEÑO (RUP)

## MODELO FÍSICO DE DATOS (MODELO DE PERSISTENCIA)



# MODELO DE DATOS

## Modelo Físico de Datos



Es un **modelo de datos de bajo nivel**. Proporciona conceptos que describen los detalles de cómo se almacenan los datos en el ordenador.

El paso de un **modelo lógico a uno físico** requiere un profundo entendimiento del **manejador de bases de datos** que se desea emplear, incluyendo características como: indexamiento, integridad referencial, restricciones, tipo de datos, parámetros de configuración, versiones, DDL.

El paso de convertir el modelo lógico de datos a tablas hace que las **entidades pasen a ser tablas** (más las derivadas de las relaciones) y los **atributos se convierten en las columnas de dichas tablas**.

# MODELO DE DATOS

## Modelo Físico de Datos



Es un **modelo de datos de bajo nivel**. Proporciona conceptos que describen los detalles de cómo se almacenan los datos en el ordenador.

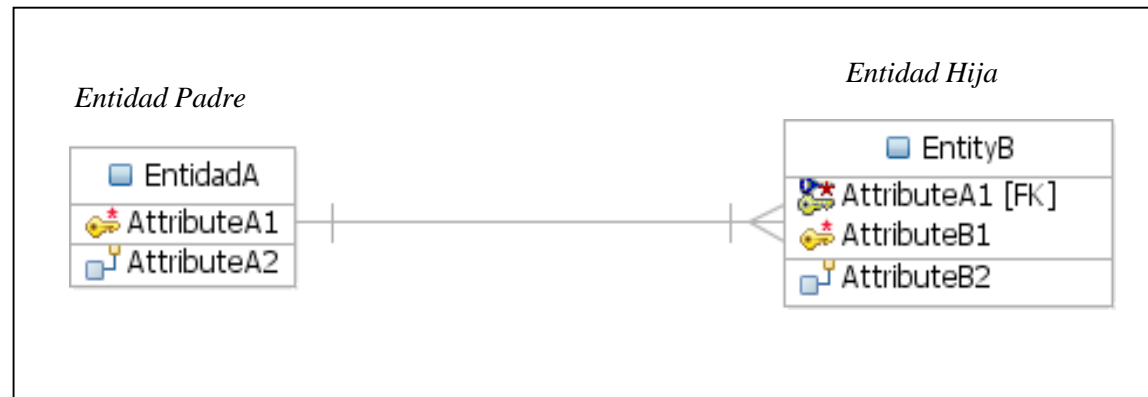
El paso de un **modelo lógico a uno físico** requiere un profundo entendimiento del **manejador de bases de datos** que se desea emplear, incluyendo características como: indexamiento, integridad referencial, restricciones, tipo de datos, parámetros de configuración, versiones, DDL.

El paso de convertir el modelo lógico de datos a tablas hace que las **entidades pasen a ser tablas** (más las derivadas de las relaciones) y los **atributos se convierten en las columnas de dichas tablas**.



### 1. Relación Identificada:

Las relaciones identificadas **migran la llave primaria de la entidad padre a la llave primaria de la entidad hija**. Su representación es una línea nítida.





### 2. Relación No Identificada:

En una relación no identificada **la llave primaria de la entidad padre es migrada como un atributo más de la entidad hija**, es decir, no formará parte de su llave primaria. Hay de dos tipos:

- a) Opcional
- b) Obligatoria

# MODELO FÍSICO DE DATOS

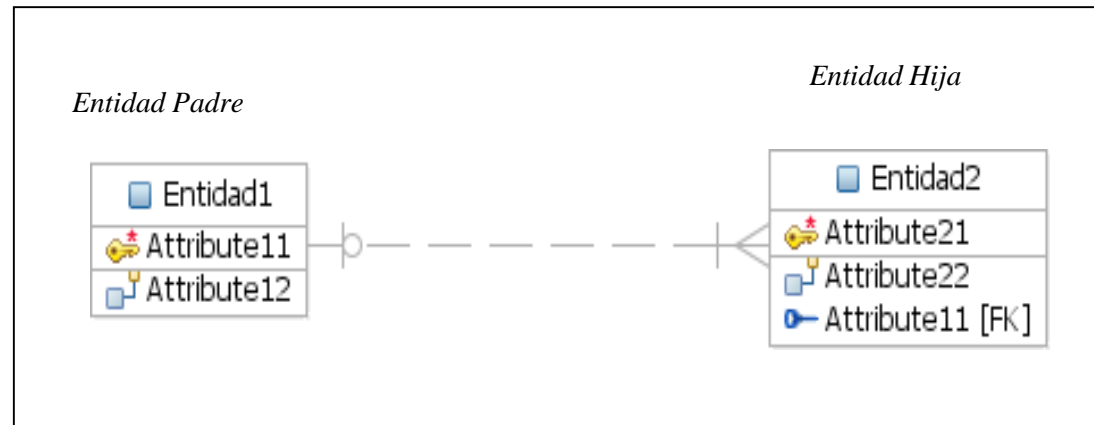
## Tipos de Relaciones



### 2.1 Relación No Identificada Opcional:

Cuando el valor de una llave foránea no es siempre requerido en la entidad hija. Sin embargo, si un valor existe, el valor de la llave foránea debe encontrarse en la llave primaria de la entidad padre.

La representación de la relación No identificada Opcional es una línea entrecortada y en el extremo de la entidad padre aparece una línea con un círculo.

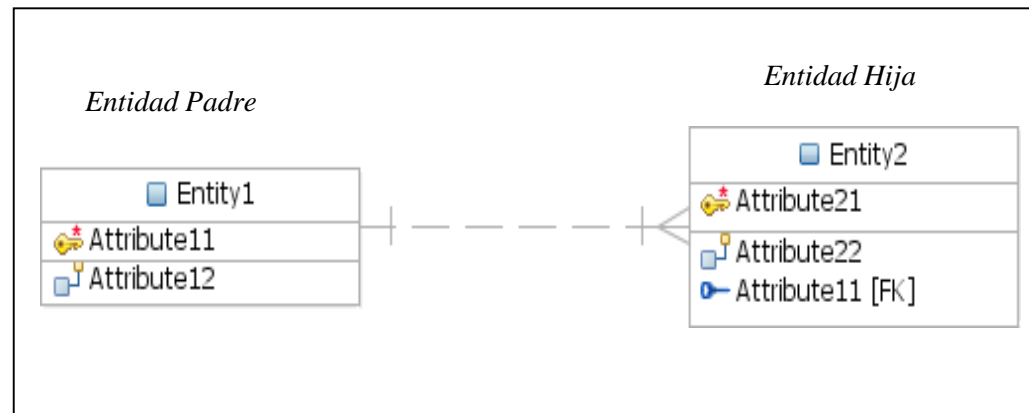




### 2.2 Relación No Identificada Obligatoria:

Cuando el valor de una llave foránea debe existir en la entidad hija y el valor de la llave foránea debe encontrarse en la llave primaria de la entidad padre.

La representación de la relación No identificada Obligatoria es una línea entrecortada y en el extremo de la entidad padre aparece una línea.





### 1. Tipos de Datos:

Los tipos de datos son según los disponibles en el Sistema Manteador de Base de Datos (SMBD). En la mayoría de casos se considera lo siguiente:

- Número de dígitos en números enteros
- La precisión de los flotantes
- Cadenas de caracteres de longitud fija. Ejemplo: CHAR(5)
- Cadenas de caracteres de longitud variable. Ejemplo: VARCHAR(50)
- Para archivos binarios (imágenes) se utiliza el tipo BLOB (Binary Large Objects)
- Para cadenas de longitudes grandes se utiliza el tipo CLOB (Character Large Objects)





## 2. Llaves primarias:

Representa el identificador de una tabla y está formada por uno o más campos de la tabla.

Algunos SMBD poseen la capacidad de "autoincrement" o "identity property" con la cual pueden automáticamente manipular algún atributo para generar llaves primarias de forma incremental. Pero es importante verificar:

- Cómo se manejan internamente.
- Si se pueden reiniciar.
- Si se permite especificar algún valor inicial.



### 3. Orden de los campos o columnas:

Por lo general la secuencia es:

- Columnas de longitud fija que no se actualizan frecuentemente.
- Aquellas que nunca se actualizan, que por lo general tendrán longitud variable.
- Las que se actualizan frecuentemente.



### 4. Integridad Referencial:

- En la medida de lo posible indicar qué columnas brindan o sirven de vínculo entre 2 tablas.
- El administrador de base de datos puede hacerse cargo de esto pero es mejor que el SMBD lo haga.



Un índice es un atajo desde un campo llave hacia la localización real de los datos. Es el punto clave de la optimización de velocidad de toda base de datos.

Si se busca alguna tupla en base a un atributo que no tiene un índice, entonces se realiza un escaneo de la tabla completa lo cual es demasiado costoso, por eso es recomendable usar índices en:

- Llaves primarias
- Llaves foráneas
- Índices de acceso
- Ordenamiento

# MODELO FÍSICO DE DATOS

## Indices

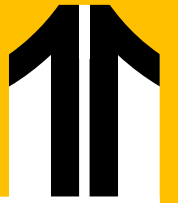


No olvidar que el uso de un índice implica:

- *Overhead* debido a la actualización de los mismos.
- Espacio adicional en disco.
- Procesos *batch* de muchos datos pueden volverse demasiado lentos.
- Manipulación de archivos adicionales por el sistema operativo.

# MODELO FÍSICO DE DATOS

Sistemas Manejadores de Base de Datos (SMBD) más conocidos



SYBASE



ORACLE



mongoDB



Microsoft®  
SQL Server®



Firebird®



PostgreSQL



Informix®  
SOFTWARE



Apache Derby



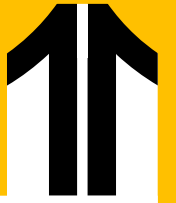
SQLite



MariaDB

# Actividad de aprendizaje

## Individual



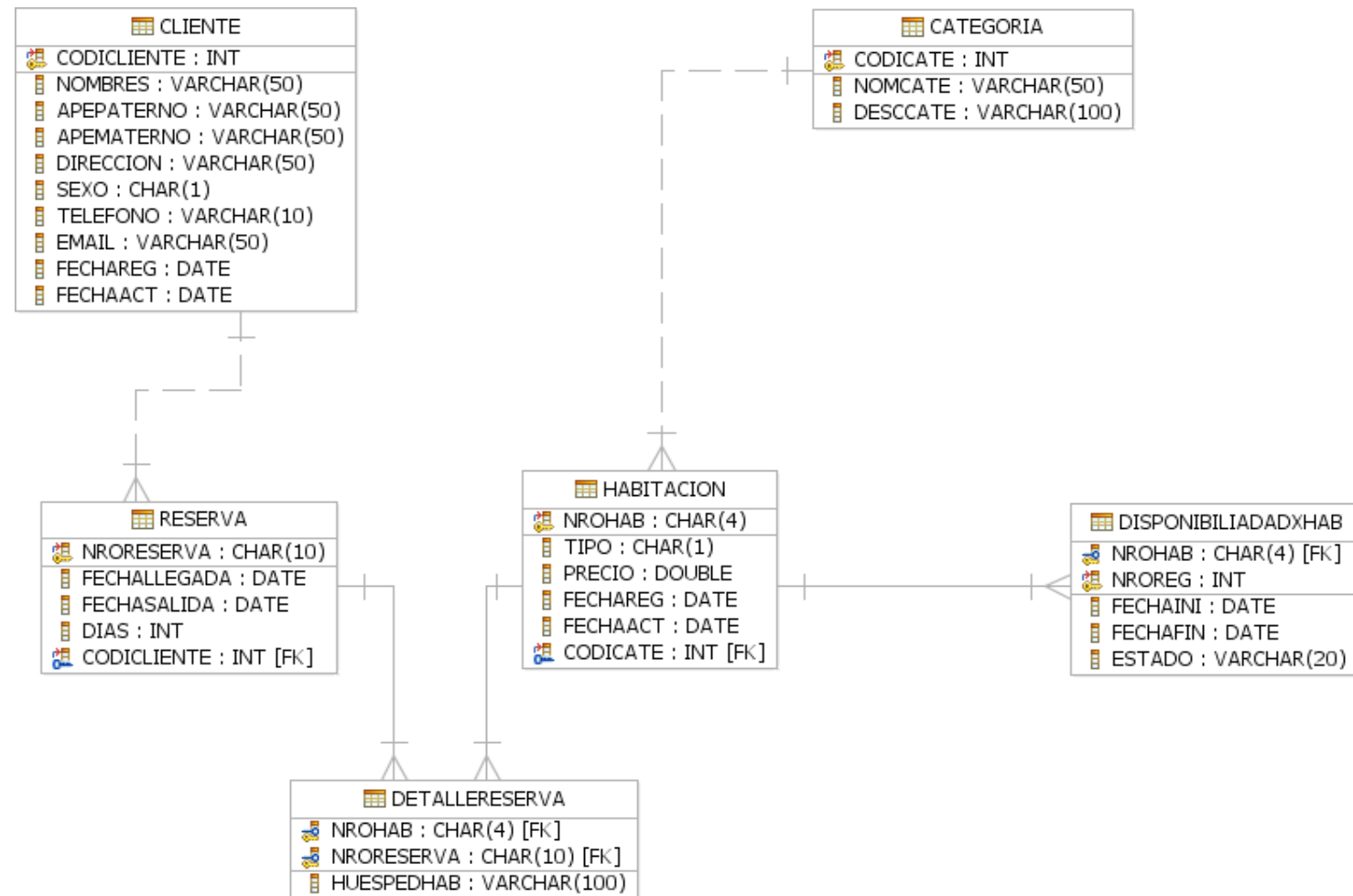
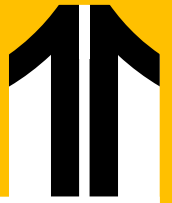
REALIZAR UN MODELO DE DATOS  
PARA LA RESERVA DE UN HOTEL



REPRESENTAR EL MODELO  
IDENTIFICADO EN UN ARCHIVO  
EN PDF

# MODELO FÍSICO DE DATOS

## Modelo Físico de Datos







# Disciplina Análisis y Diseño (RUP)

## Modelo de Diseño

# PATRONES DE DISEÑO

## Interés



¿Cuál es la idea central del Video?

¿Para qué sirve los Patrones de Diseño?



<https://youtu.be/LxVQ7SsbheE>

# PATRONES DE DISEÑO

Hacer Software no es fácil



Diseñar software orientado a objetos es difícil, y diseñar software orientado a objetos reutilizable es todavía más difícil

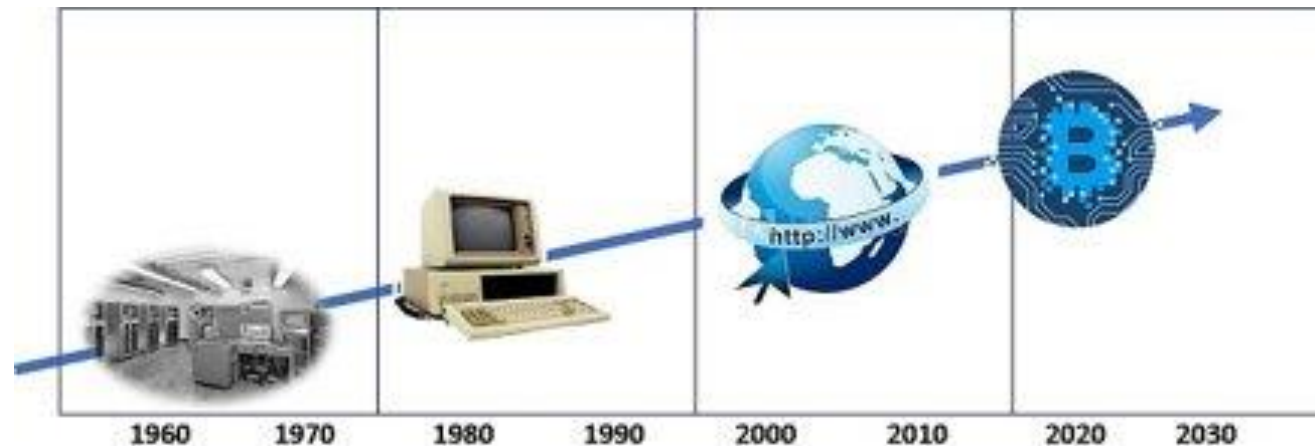
...y un software capaz de evolucionar tiene que ser reutilizable (al menos para las versiones futuras)

# PATRONES DE DISEÑO

## Diseñar para el cambio



- El software cambia.
- Parara anticiparse a los cambios en los requisitos hay que diseñar pensando en qué aspectos pueden cambiar
- Los patrones de diseño están orientados al cambio



# PATRONES DE DISEÑO

## Patrones



### **Cómo llegar a ser un maestro de ajedrez**

- Primero aprender las reglas del juego
  - Nombres de las piezas, movimientos legales, geometría y orientación del tablero, etc.
- A continuación, aprender los principios
  - Relativo valor de las piezas, valor estratégico de las casillas centrales, jaque cruzado, etc.
- Sin embargo, para llegar a ser un maestro, hay que estudiar las partidas de otros maestros
  - Estas partidas contienen patrones que deben ser entendidos, memorizados y aplicados repetidamente
- Hay cientos de estos patrones



# PATRONES DE DISEÑO

## Patrones



Cómo llegar a ser un maestro del software

- Primero aprender las reglas
  - Algoritmos, estructuras de datos, lenguajes de programación, etc.
- A continuación, aprender los principios
  - Programación estructurada, programación modular, programación OO, programación genérica, etc.
- Sin embargo, para llegar a ser un maestro, hay que estudiar los diseños de otros maestros
  - Estos diseños contienen patrones que deben ser entendidos, memorizados y aplicados repetidamente
- Hay cientos de estos patrones

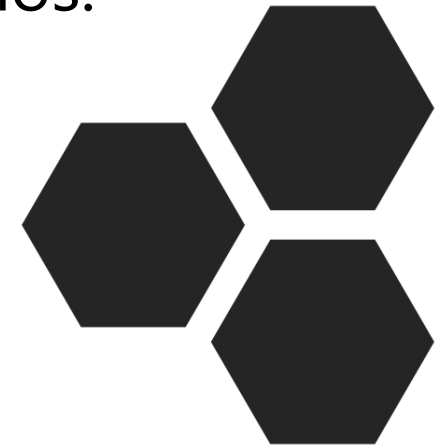


# PATRONES DE DISEÑO

## ¿Qué es un Patrón?



“Un patrón arquitectónico se puede considerar como una descripción abstracta estilizada de buena práctica, que se ensayó y puso a prueba en diferentes sistemas y entornos.”  
(Sommerville, 2011)



### **Propósito:**

- Compartir una solución probada.
- Ampliamente aplicable a un problema particular de diseño.
- Se presenta en una forma estándar que permite que sea fácilmente reutilizado.

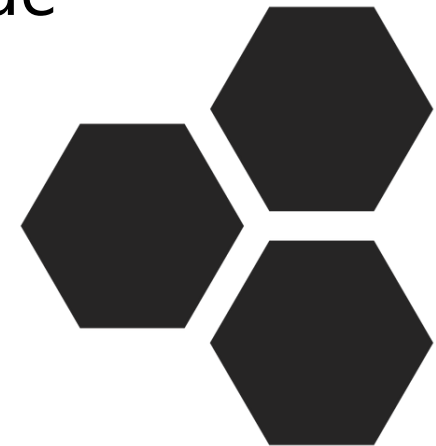
# PATRONES DE DISEÑO

## ¿Qué resuelve un Patrón de Diseño?



Un patrón de diseño es una manera de **resolver un problema** dentro de un **contexto**. En otras palabras, son **plantillas para soluciones a problemas comunes** en el desarrollo de software que se pueden usar en diferentes contextos.

Los patrones de diseño dan soluciones fáciles a problemas complejos sin importar el lenguaje que estemos usando.





# PATRONES DE DISEÑO

## ¿Para qué sirven los Patrones de Diseño?



Los patrones de diseño son **modelos muestra que sirven como guía para que los programadores trabajen sobre ellos.**

Un patrón de diseño debe cumplir al menos con los siguientes objetivos:

- Estandarizar el lenguaje entre programadores.
- Evitar perder tiempo en soluciones a problemas ya resueltos o conocidos.
- Crear código reusable (excelente ventaja).
- Existen muchos patrones, **lo importante es saber que existen y preocuparnos por entenderlos y usarlos.**



# PATRONES DE DISEÑO

## Tipos de Patrones de Diseño



Los patrones de diseño se clasifican en tres tipos diferentes dependiendo del tipo de problema que resuelven. Estos pueden ser:

- Creacionales
- Estructurales
- De comportamiento



# PATRONES DE DISEÑO

## Patrones de Diseño Creacionales



Su objetivo es **resolver los problemas de creación de instancia**. Estos ayudan a delegar responsabilidad de creación de objetos en situaciones necesarias.

Sus pilares fundamentales son encapsular el conocimiento de las clases y Ocultar cómo se crean y se instancian.



# PATRONES DE DISEÑO

## Patrones de Diseño Estructurales



Su nombre es muy descriptivo, **se ocupa de resolver problemas sobre la estructura de las clases**, es decir, se enfocan en cómo las clases y objetos se componen para formar estructuras mayores.



# PATRONES DE DISEÑO

## Patrones de Diseño de Comportamiento



Nos ayuda a **resolver problemas relacionados con el comportamiento de la aplicación**. Ofrece soluciones respecto a la interacción y responsabilidad entre objetos y clases.



# PATRONES DE DISEÑO

## Tipos de Patrones de Diseño



# PATRONES DE DISEÑO

## Tipos de Patrones de Diseño



	CREACIÓN	ESTRUCTURALES	COMPORTAMIENTO
CLASES	Factory Method		Interpreter Template Method
OBJETOS	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

# PATRONES DE DISEÑO

## Patrones de Diseño Creacionales



### **Abstract Factory:**

Permite proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

### **Builder:**

Ayuda a crear objetos complejos de manera sencilla, legible y escalable. Se utiliza en situaciones en las que debe construirse un objeto repetidas veces.

### **Factory Method:**

Nos ayuda a tener instancias de un objeto dado el tipo.





# PATRONES DE DISEÑO

## Patrones de Diseño Creacionales



### Singleton

(Instancia única): nos garantiza la existencia de una única instancia para una clase.

### Prototype (prototipo):

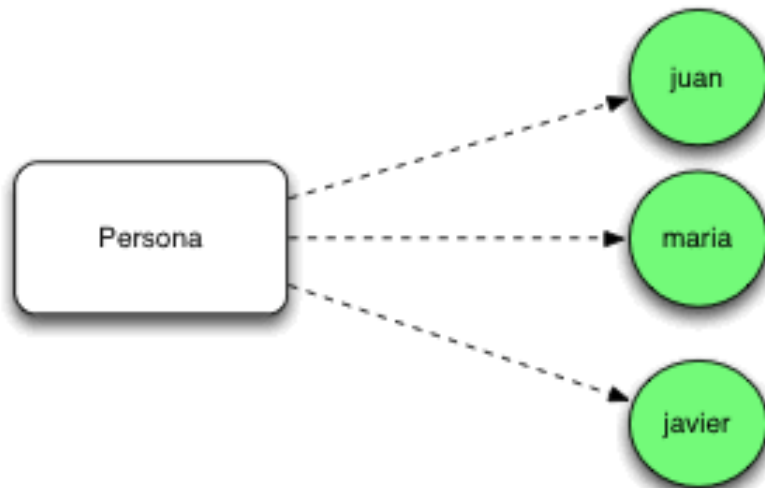
Clona las instancias ya existentes.





### Ejemplo de Implementación de Patrón Singleton (Java Singleton)

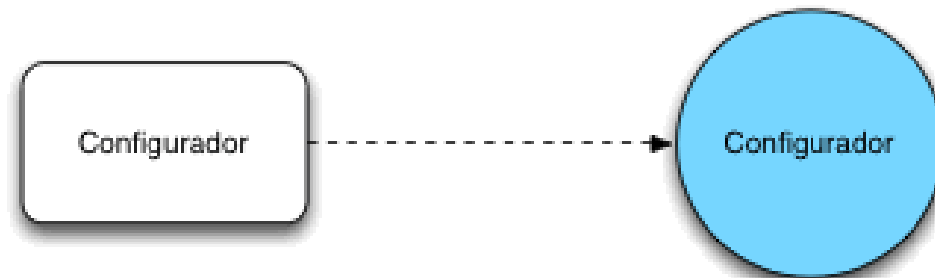
Este patrón de diseño se encarga de que una clase determinada únicamente pueda tener un único objeto. Normalmente una clase puede instanciar todos los objetos que necesite.





### Ejemplo de Implementación de Patrón Singleton (Java Singleton)

Sin embargo una clase que siga el patrón Singleton tiene la peculiaridad de que solo puede instanciar un único objeto. Este tipo de clases son habituales en temas como configurar parámetros generales de la aplicación ya que una vez instanciado el objeto los valores se mantienen y son compartidos por toda la aplicación. Vamos a configurar una clase con el patrón Singleton, a esta clase la llamaremos Configurador.





### Ejemplo de Implementación de Patrón Singleton (Java Singleton)

Una vez que tenemos claro cual es el concepto de Configurador vamos a crearlo en código .En este caso nuestro configurador almacenará dos valores url, y base de datos que serán compartidos por el resto de Clases de la aplicación.



# PATRONES DE DISEÑO

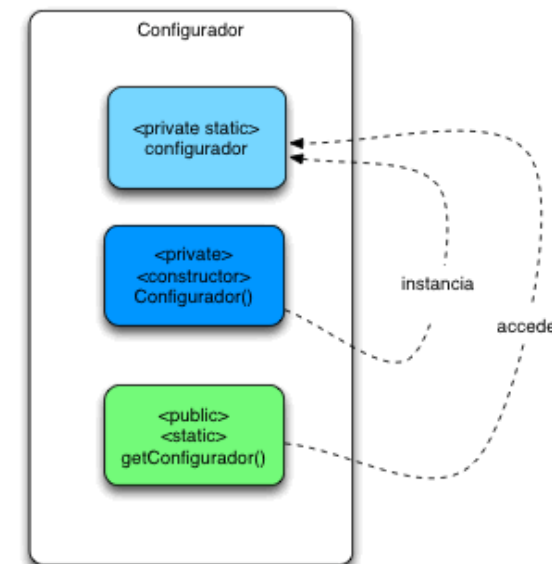
## Patrones de Diseño Creacionales – Ejemplo Singleton (4)



### Ejemplo de Implementación de Patrón Singleton (Java Singleton)

Para conseguir que una clase sea de tipo Singleton necesitamos en primer lugar que **su constructor sea privado**. De esa forma ningún programa será capaz de construir objetos de esta tipo . En segundo lugar necesitaremos disponer de una **variable estática privada** que almacene una referencia al objeto que vamos a crear a través del constructor Por ultimo **un método estático público que se encarga de instancia el objeto la primera vez y almacenarlo en la variable estática**.

```
1 package com.arquitecturajava;
2
3 public class Configurador {
4
5     private String url;
6     private String baseDatos;
7     private static final Configurador miconfigurador;
8
9     public static Configurador getConfigurador(String url,String baseDatos) {
10
11         if (miconfigurador==null) {
12
13             miconfigurador=new Configurador(url,baseDatos);
14         }
15         return miconfigurador;
16     }
17
18     private Configurador(String url,String baseDatos){
19
20         this.url=url;
21         this.baseDatos=baseDatos;
22     }
23
24     public String getUrl() {
25         return url;
26     }
27
28     public void setUrl(String url) {
29         this.url = url;
30     }
31
32     public String getBaseDatos() {
33         return baseDatos;
34     }
35
36     public void setBaseDatos(String baseDatos) {
37         this.baseDatos = baseDatos;
38     }
39 }
40
```





### Ejemplo de Implementación de Patrón Singleton

#### (Java Singleton)

Una vez aclarado como funciona un Singleton es muy sencillo utilizarle desde un programa ya que basta con invocar al método estático.

```
1 package com.arquitecturajava;
2
3 public class Principal {
4
5     public static void main(String[] args) {
6
7         Configurador c= Configurador.getConfigurador("miurl", "mibaseDatos");
8
9         System.out.println(c.getUrl());
10
11         System.out.println(c.getBaseDatos());
12
13     }
14
15 }
```



# PATRONES DE DISEÑO

## Patrones de Diseño Estructurales



### **Bridge (Punto):**

Permite desacoplar una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.

### **Decorator (Decorador):**

Agrega funcionalidades a una clase de forma dinámica.

### **Facade (Fachada):**

Nos provee una interfaz unificada y simple para acceder a un sistema más complejo.

### **Adapter:**

Cuando dos clases no se entienden, el adapter es mediador y adapta una clase para que la otra la entienda.



Estructurales

# PATRONES DE DISEÑO

## Patrones de Diseño Estructurales



### **Composite:**

Ayuda a construir objetos complejos a partir de otros más simples.

### **Flyweight:**

Se refiere a los objetos que queremos reutilizar para crear objetos más ligeros.

### **Proxy:**

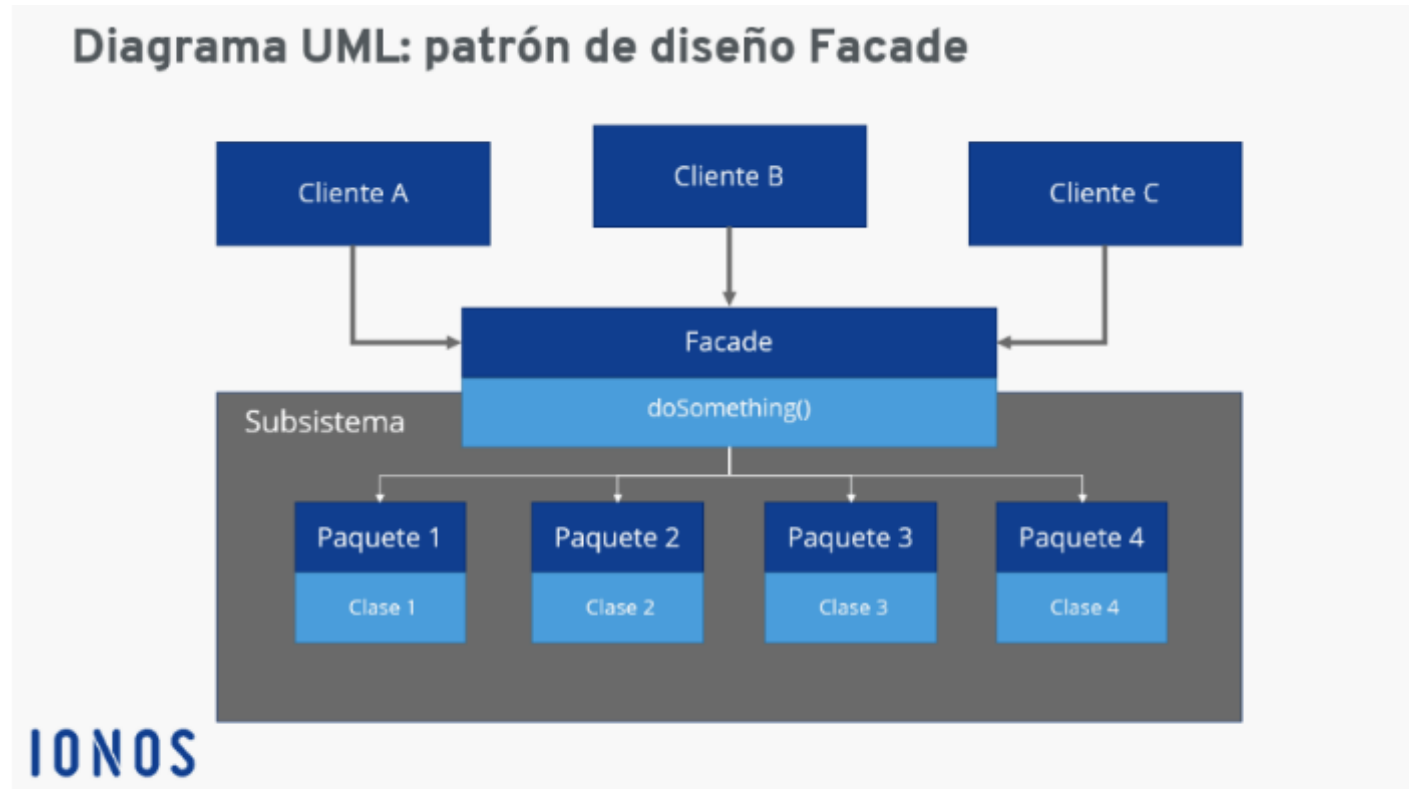
Es un elemento que se encarga de introducir un nivel de acceso a una clase. Ese nivel de acceso puede ser por seguridad o por complejidad.





# PATRONES DE DISEÑO

## Patrones de Diseño Estructurales – Ejemplo Facade (1)



El número de cuatro paquetes de clases en el diagrama UML es solo un ejemplo. Teóricamente, cualquier número de clases en el subsistema puede ser controlado a través de la clase de fachada.



# PATRONES DE DISEÑO

## Patrones de Diseño Estructurales – Ejemplo Facade (1)



### Ejemplo: Sacar Dinero de la cuenta de un banco

Ilustraremos el patron Facade mediante un ejemplo muy común en la vida real: sacar dinero de la cuenta del banco. Para la retirada de efectivo en un cajero automático, se requiere un gran número de operaciones para garantizar la seguridad y la integridad.

En primer lugar vamos a ver los prototipos de las clases (con implementación dummy, ya que no es relevante) que intervienen en el proceso de retirada de efectivo de un cajero automático:



# PATRONES DE DISEÑO

## Patrones de Diseño Estructurales – Ejemplo Facade (2)



### Ejemplo: Sacar Dinero de la cuenta de un banco

```
public class Autenticacion{  
    /* ... */  
    public boolean leerTarjeta(){}  
    public String introducirClave(){}  
    public boolean comprobarClave(String clave){}  
    public Cuenta obtenerCuenta(){}  
    public void alFallar(){}  
}  
  
public class Cajero{  
    /* ... */  
    public int introducirCantidad(){}  
    public boolean tieneSaldo(int cantidad){}  
    public int expedirDinero{}  
    public String imprimirTicket(){}  
}
```

```
public class Cuenta{  
    /* ... */  
    public double comprobarSaldoDisponible(){}  
    public boolean bloquearCuenta(){}  
    public boolean desbloquearCuenta{}  
    public void retirarSaldo(int cantidad){}  
    public boolean actualizarCuenta(){}  
    public void alFallar(){}  
}
```



Estructurales

# PATRONES DE DISEÑO

## Patrones de Diseño Estructurales – Ejemplo Facade (3)



Mediante estas operaciones, el cliente tendría que acceder a 3 subsistemas y realizar una inmensa cantidad de operaciones. Sin embargo, crearemos una fachada para ofrecer una interfaz mucho más amigable

```
public class FachadaCajero{

    private Autenticacion autenticacion = new Autenticacion();

    private Cajero cajero = new Cajero();

    private Cuenta cuenta = null;

    public void introducirCredenciales(){

        boolean tarjeta_correcta = autenticacion.leerTarjeta();

        if(tarjeta_correcta){

            String clave = autenticacion.introducirClave();

            boolean clave_correcta = autenticacion.comprobarClave(clave);

            if(clave_correcta){

                cuenta = autenticacion.obtenerCuenta();

                return;

            }

        }

        autenticacion.alFallar();

    }

}
```

```
public void sacarDinero(){

    if(cuenta != null){

        int cantidad = cajero.introducirCantidad();

        int tiene_dinero = cajero.tieneSaldo(cantidad);

        if(tiene_dinero){

            boolean hay_saldo_suficiente = ((int)cuenta.comprobarSaldoDisponible()) >= cantidad;

            if(hay_saldo_suficiente){

                cuenta.bloquearCuenta();

                cuenta.retirarSaldo(cantidad);

                cuenta.actualizarCuenta();

                cuenta.desbloquearCuenta();

                cajero.expedirDinero();

                cajero.imprimirTicket();

            }

            else{

                cuenta.alFallar();

            }

        }

    }

}
```



Estructurales



### Ejemplo: Sacar Dinero de la cuenta de un banco

De esta manera, hemos conseguido que para que un cliente use el cajero, no tenga que realizar todas las operaciones de sus subsistemas. En lugar de ello proporcionamos una interfaz mucho más simple que facilita enormemente su uso

```
public static void main(String[] args){  
  
    FachadaCajero cajero_automatico = new FachadaCajero();  
  
    cajero_automatico.introducirCredenciales();  
  
    cajero_automatico.sacarDinero();  
  
}
```



# PATRONES DE DISEÑO

## Patrones de Diseño de Comportamiento



### **Observer (Observador):**

Un objeto le pasa el estado interno a muchos objetos que están interesados.

### **Chain of Responsibility:**

Simplifica las interconexiones de objetos.

### **Command:**

Separa acciones que pueden ser ejecutadas desde varios puntos diferentes de la aplicación a través de una interfaz sencilla.



Comportamiento

# PATRONES DE DISEÑO

## Patrones de Diseño de Comportamiento



### **Iterator:**

Este se utiliza en relación a objetos que almacenan colecciones de otros objetos.

### **Mediator:**

Define un objeto que media entre otros objetos.

### **Memento:**

Se utiliza para restaurar el estado de un objeto a un estado anterior.

### **State:**

Permite a un objeto alterar su comportamiento cuando su estado interno cambia.



# PATRONES DE DISEÑO

## Patrones de Diseño de Comportamiento



### **Strategy:**

Permite que un objeto tenga parte o todo su comportamiento definido en términos de otro objeto que sigue una interfaz particular.

### **Template Method:**

Se centra en la reutilización del código para implementar pasos para resolver problemas.

### **Visitor:**

Se utiliza para separar la lógica y las operaciones realizadas sobre una estructura compleja.





# PATRONES DE DISEÑO

## Patrones de Diseño de Comportamiento



### **Strategy:**

Permite que un objeto tenga parte o todo su comportamiento definido en términos de otro objeto que sigue una interfaz particular.

### **Template Method:**

Se centra en la reutilización del código para implementar pasos para resolver problemas.

### **Visitor:**

Se utiliza para separar la lógica y las operaciones realizadas sobre una estructura compleja.



# PATRONES DE DISEÑO

## Patrones de Diseño de Comportamiento – Ejemplo Strategy (1)

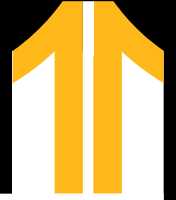


### Patrón Strategy en Java

El patrón de diseño Strategy en Java ayuda a definir diferentes comportamientos o funcionalidades que pueden ser cambiadas en tiempo de ejecución.

En el patrón Strategy se crean diferentes clases que representan estrategias y que se pueden usar según alguna variación o input.





### Patrón Strategy en Java

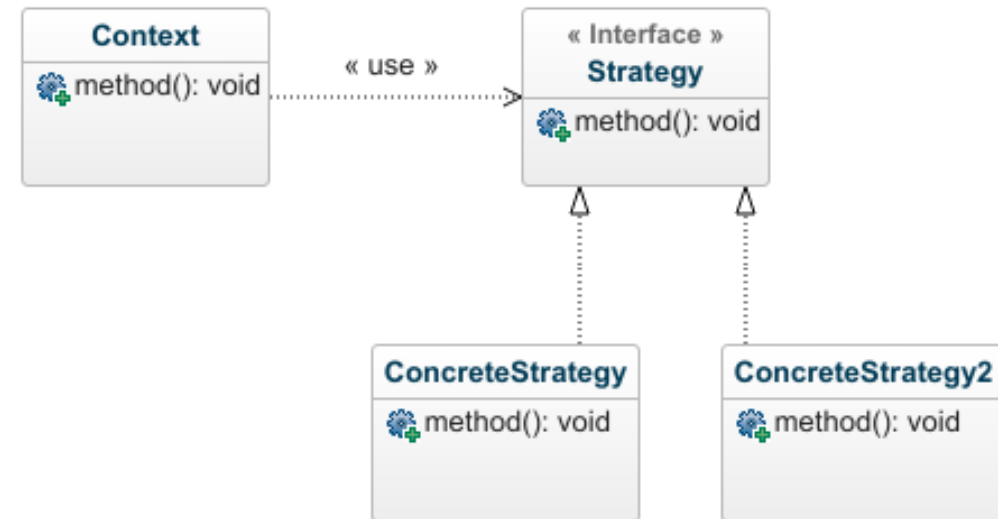
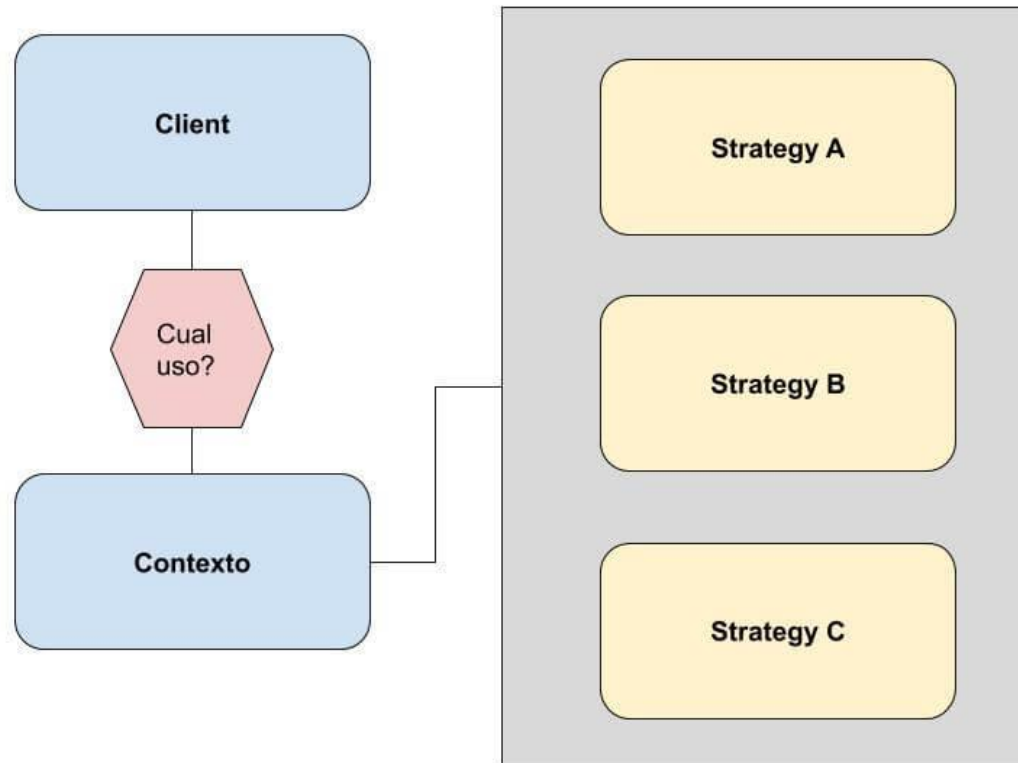
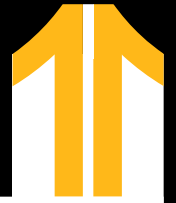
Los componentes del patrón Strategy son:

- Interfaz Strategy: es la interfaz que define cómo se conformará el contrato de la estrategia.
- Clases Concretas Strategy: son las clases que implementan la interfaz y donde se desarrolla la funcionalidad.
- Contexto: donde se establece que estrategia se usará.



# PATRONES DE DISEÑO

## Patrones de Diseño de Comportamiento – Ejemplo Strategy (3)



# PATRONES DE DISEÑO

## Patrones de Diseño de Comportamiento – Ejemplo Strategy (4)



Usaremos un ejemplo muy simple para explicar el patrón Strategy. Tenemos diferentes comisiones que se pagan según un monto de venta. Si el monto de venta es mayor se paga mayor comisión.

Definimos la interfaz Strategy de la cual crearemos estrategias puntuales para cada caso.

```
package patterns.strategy;  
  
public interface CommissionStrategy {  
    double applyCommission(double amount);  
}
```



Comportamiento

# PATRONES DE DISEÑO

## Patrones de Diseño de Comportamiento – Ejemplo Strategy (4)



Pensamos en las implementaciones de cada Strategy.  
Implementación para un pago de una comisión full.

```
package patterns.strategy;

public class FullCommission implements CommissionStrategy {
    @Override
    public double applyCommission(double amount) {
        // do complicated formula of commissions.
        return amount * 0.50d;
    }
}
```

Implementación para un pago de una comisión normal.

```
package patterns.strategy;

public class NormalCommission implements CommissionStrategy {
    @Override
    public double applyCommission(double amount) {
        // do complicated formula of commissions.
        return amount * 0.30;
    }
}
```



Comportamiento

# PATRONES DE DISEÑO

## Patrones de Diseño de Comportamiento – Ejemplo Strategy (4)



Implementación para un pago de una comisión regular.

```
package patterns.strategy;

public class RegularCommision implements CommissionStrategy {
    @Override
    public double applyCommission(double amount) {
        // do complicated formula of commissions.
        return amount * 0.10;
    }
}
```

Creamos el contexto para las estrategias.

```
package patterns.strategy;

public class Context {

    private CommissionStrategy commissionStrategy;

    public Context(CommissionStrategy commissionStrategy){
        this.commissionStrategy = commissionStrategy;
    }

    public double executeStrategy(double amount){
        return commissionStrategy.applyCommission(amount);
    }
}
```



Comportamiento

# PATRONES DE DISEÑO

## Patrones de Diseño de Comportamiento – Ejemplo Strategy (4)



Probamos la estrategia con diferentes montos de venta.

```
package patterns.strategy;

public class StrategyPatternExample {

    public static void main(String[] args) {

        CommissionStrategy commissionStrategy = getStrategy(1000d);
        Context context = new Context(commissionStrategy);
        System.out.println("Commission for 1000d = " + context.executeStrategy(1000d));

        commissionStrategy = getStrategy(500d);
        context = new Context(commissionStrategy);
        System.out.println("Commission for 500d = " + context.executeStrategy(500d));

        commissionStrategy = getStrategy(100d);
        context = new Context(commissionStrategy);
        System.out.println("Commission for 100d = " + context.executeStrategy(100d));
    }

    private static CommissionStrategy getStrategy(double amount) {
        CommissionStrategy strategy;
        if (amount >= 1000d) {
            strategy = new FullCommission();
        } else if (amount >= 500d && amount <= 999d) {
            strategy = new NormalCommission();
        } else {
            strategy = new RegularCommision();
        }
        return strategy;
    }
}
```





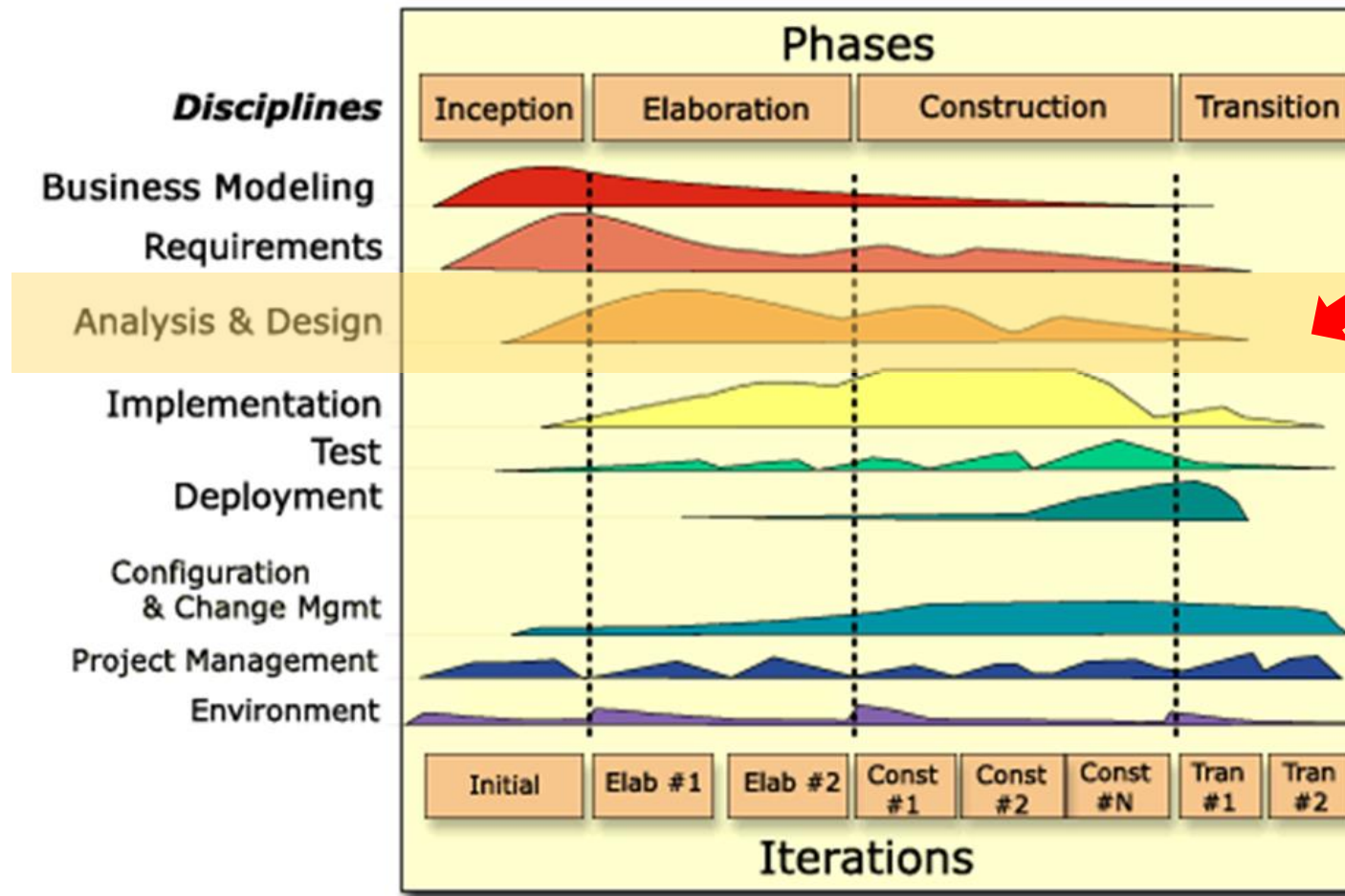
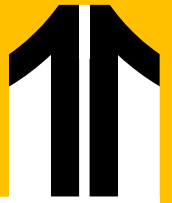


# Modelo de Diseño

**Tercera Disciplina de RUP**

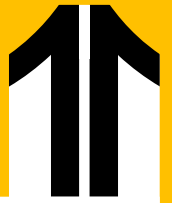
# DISCIPLINA ANÁLISIS Y DISEÑO (RUP)

## Tercera Disciplina

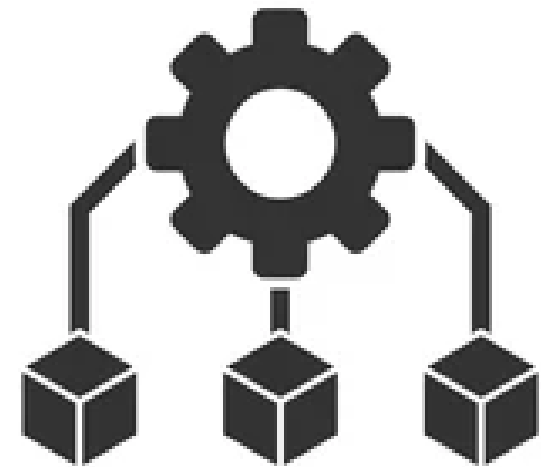


# Modelo de Diseño

## Diseño Orientado a Objetos

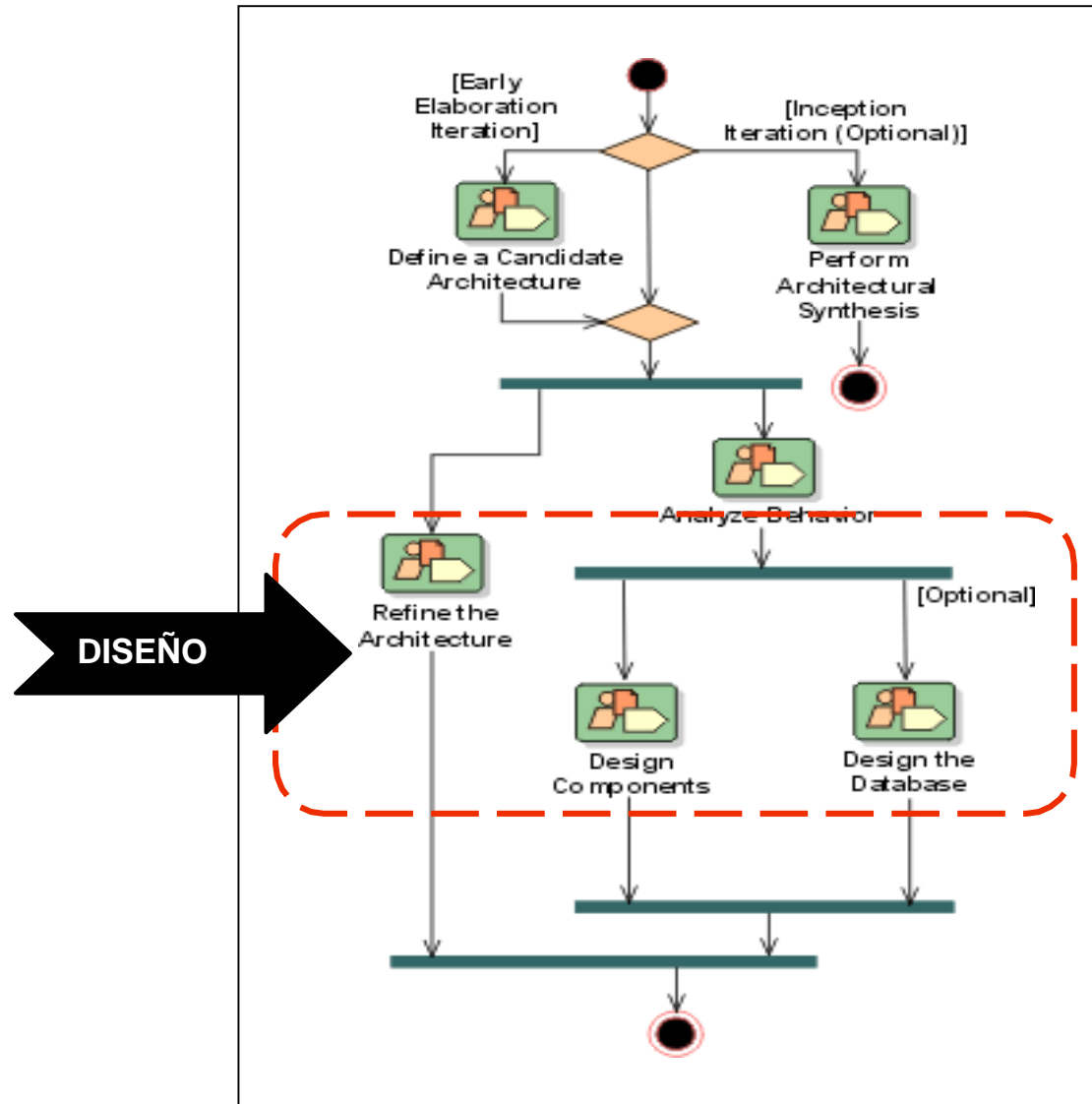


El objetivo del **diseño** es **entender la solución** refinando el modelo de análisis con la intención de desarrollar un **modelo de diseño** que permita una **transición** sin problemas a la **fase de construcción**. En el diseño, nos adaptamos al entorno de implementación y despliegue.



# DISCIPLINA ANÁLISIS Y DISEÑO (RUP)

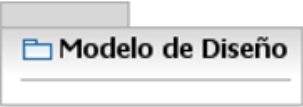
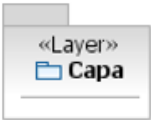
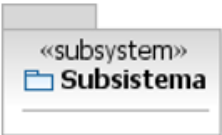

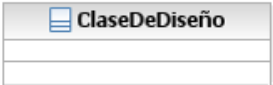
## Flujo de Trabajo del Análisis y Diseño



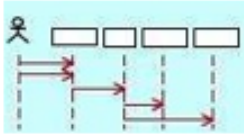


# Artefactos del Diseño

## Artefactos



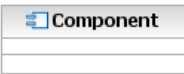

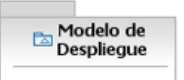
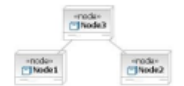


Artefacto	Descripción
 Modelo de Diseño	Representa la vista interna del sistema. Conjunto formado por las clases de diseño y las realizaciones de casos de uso. El modelo de diseño se convertirá en la materia prima que nuestra disciplina de implementación transformará en código ejecutable.
 Capa	Representan un medio para organizar los artefactos del modelo de diseño. Dependiendo del estilo arquitectónico, una capa agrupa un conjunto de subsistemas junto con sus clases de diseño.
 Subsistema	Un subsistema contiene las clases de diseño de un grupo de casos de uso. Tiene correspondencia directa con los paquetes de análisis.
 Librería	Una librería contiene clases utilitarias, adicionales a las del API del lenguaje de programación, implementadas por el equipo de desarrollo.
 Clases de diseño	Son abstracciones de clase directamente utilizadas en la implementación, es decir, estas clases junto con sus atributos y operaciones se mapean directamente en el lenguaje de programación.

 Realización de Diseño de Caso de Uso	Describe cómo un caso de uso se lleva a cabo en términos de clases de diseño y sus objetos. Hay una correspondencia directa entre Realización de Diseño de Casos de Uso y Realización de Análisis de Casos de Uso.
 Diagrama de Clases	El diagrama de clases describe la estructura de un caso de uso. Contiene las clases que participan en el caso de uso, aunque algunas de ellas puedan participar en varios.
 Diagramas de Secuencia	Muestra una secuencia detallada de interacción entre los objetos de diseño. Visualizan el intercambio de mensajes entre objetos. Se crea un diagrama de secuencia por cada flujo de trabajo del caso de uso: flujo básico, subflujos y flujos alternativos.

# Artefactos del Diseño

## Artefactos (Continuación)



Artefacto	Descripción
 Componente	Un componente representa una pieza del software reutilizable. Por ejemplo, si se está desarrollando una aplicación web estas piezas pueden ser recursos estáticos (páginas HTML) y recursos dinámicos (JSP y <del>servlets</del> <b>servlets</b> ) representados como componentes.
 Diagrama de componentes	Un diagrama de componentes muestra la estructura de un sistema <i>software</i> , el cual describe los componentes <i>software</i> , sus interfaces y sus dependencias.
 Modelo de Despliegue	Describe la distribución física del sistema en términos de cómo las funcionalidades se distribuyen entre los nodos de computación sobre los que se va a instalar el sistema.
 Diagrama de Despliegue	Un diagrama de despliegue se puede utilizar para visualizar la topología actual del sistema y la distribución de componentes.
 Artefacto	Los artefactos son elementos que representan las entidades físicas de un sistema <i>software</i> . Los artefactos representan unidades de implementación física como archivos ejecutables, librerías, componentes de <i>software</i> , documentos, y bases de datos.
 Nodo	Representa un recurso de computación. Los nodos tienen relaciones entre ellos que representan los medios de comunicación que hay entre éstos como una Intranet o Internet. La funcionalidad de un nodo viene representada por los componentes que se ejecutan en él.

# Modelo de Diseño

## Modelo de Análisis Vs Modelo de diseño



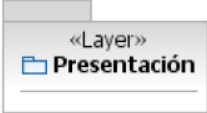
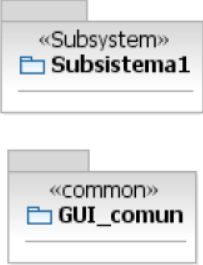

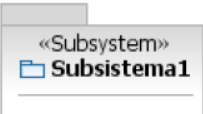
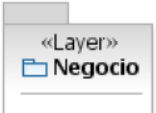
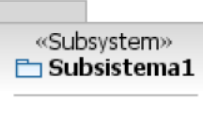

<i><b>Modelo de Análisis</b></i>	<i><b>Modelo de Diseño</b></i>
Es un modelo conceptual y genérico, es una abstracción del sistema.	Es un modelo físico y concreto, es un plano de la implementación.
Es menos formal.	Es más formal.
Es un bosquejo del diseño del sistema.	Es una realización del diseño del sistema.
Puede no mantenerse durante todo el ciclo de vida del <i>software</i> .	Debe ser mantenido durante todo el ciclo de vida del <i>software</i> .
Define una estructura para modelar el sistema.	Da forma al sistema.

# Modelo de Diseño

## Capas Lógicas de la Arquitectura



### Capas, subsistemas, librerías y elementos de diseño según patrón arquitectónico MVC

Capa	Subsistema/Librerías	Componentes
		Clases estereotipadas: <ul style="list-style-type: none"><li>• Páginas HTML: &lt;&lt;Client Page&gt;&gt; y &lt;&lt;HTML Form&gt;&gt;</li><li>• Páginas JSP: &lt;&lt;Server Page&gt;&gt;, &lt;&lt;Client Page&gt;&gt; y &lt;&lt;HTML Form&gt;&gt;</li></ul>
		Clase estereotipada para servlets: <<Http Servlet>>
		Clases de diseño: <i>beans</i> .
		Clases de diseño: clases utilitarias.



# EXPERIENCIAS

INDIVIDUALES



REALIZAR DIFERENCIAS ENTRE  
MODELO DE ANALISIS Y MODELO DE  
DISEÑO



REPRESENTAR LO IDENTIFICADO  
MEDIANTE UN GRÁFICO O  
TEXTO Y PRESENTARLO EN PDF  
O PNG  
LA ACTIVIDADE DURA 25 MIN

# CONCLUSIONES



- Un patrón de diseño es una manera de resolver un problema dentro de un contexto. Son plantillas para soluciones a problemas comunes en el desarrollo de software que se pueden usar en diferentes contextos.
- Los patrones de diseño dan soluciones fáciles a problemas complejos sin importar el lenguaje que estemos usando.
- Los patrones de diseño se clasifican en tres tipos diferentes dependiendo del tipo de problema que resuelven: creacionales, estructurales, de comportamiento.
- Un **modelo de diseño** permite una **transición** sin problemas a la **fase de construcción**. En el diseño, nos adaptamos al entorno de implementación y despliegue.

# REFERENCIAS



- Booch, G., Rumbaugh, J., Jacobson, I. (1999). "El Lenguaje Unificado de Modelado. Manual de Referencia". Madrid, España: Pearson Educación.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., (2003), Patrones de diseño. Addison-Wesley.
- Jacobson, I., Booch, G., Rumbaugh, J. (2000). "El Proceso Unificado de Desarrollo de Software". Madrid, España: Pearson Educación.
- Larman, C. (2003). "UML y Patrones" (2da ed.). Madrid, España: Pearson Educación.
- OMG.(2017). Unified Modeling Language. <https://www.omg.org/spec/UML/2.5.1/PDF>.
- Pressman, R. (2005). Ingeniería del Software. Un enfoque práctico (6a ed.). D.F., México:McGraw-Hill.
- Somerville, Ian. (2011). Ingeniería de Software (9a ed.). Madrid: Pearson Education.
- Bass, L., Clements, P., & Kazman, R. (2012). Software architecture in practice,3rd Ed. Addison-Wesley Professional

# REFERENCIAS



- Booch, G., Rumbaugh, J., Jacobson, I. (1999). "El Lenguaje Unificado de Modelado. Manual de Referencia". Madrid, España: Pearson Educación.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., (2003), Patrones de diseño. Addison-Wesley.
- Jacobson, I., Booch, G., Rumbaugh, J. (2000). "El Proceso Unificado de Desarrollo de Software". Madrid, España: Pearson Educación.
- Larman, C. (2003). "UML y Patrones" (2da ed.). Madrid, España: Pearson Educación.
- OMG.(2017). Unified Modeling Language. <https://www.omg.org/spec/UML/2.5.1/PDF>.
- Pressman, R. (2005). Ingeniería del Software. Un enfoque práctico (6a ed.). D.F., México:McGraw-Hill.
- Somerville, Ian. (2011). Ingeniería de Software (9a ed.). Madrid: Pearson Education.
- Bass, L., Clements, P., & Kazman, R. (2012). Software architecture in practice,3rd Ed. Addison-Wesley Professional



Equipo Docente  
Facultad de Ingeniería  
Universidad Privada del Norte