APLICACIONES DISTRIBUIDAS

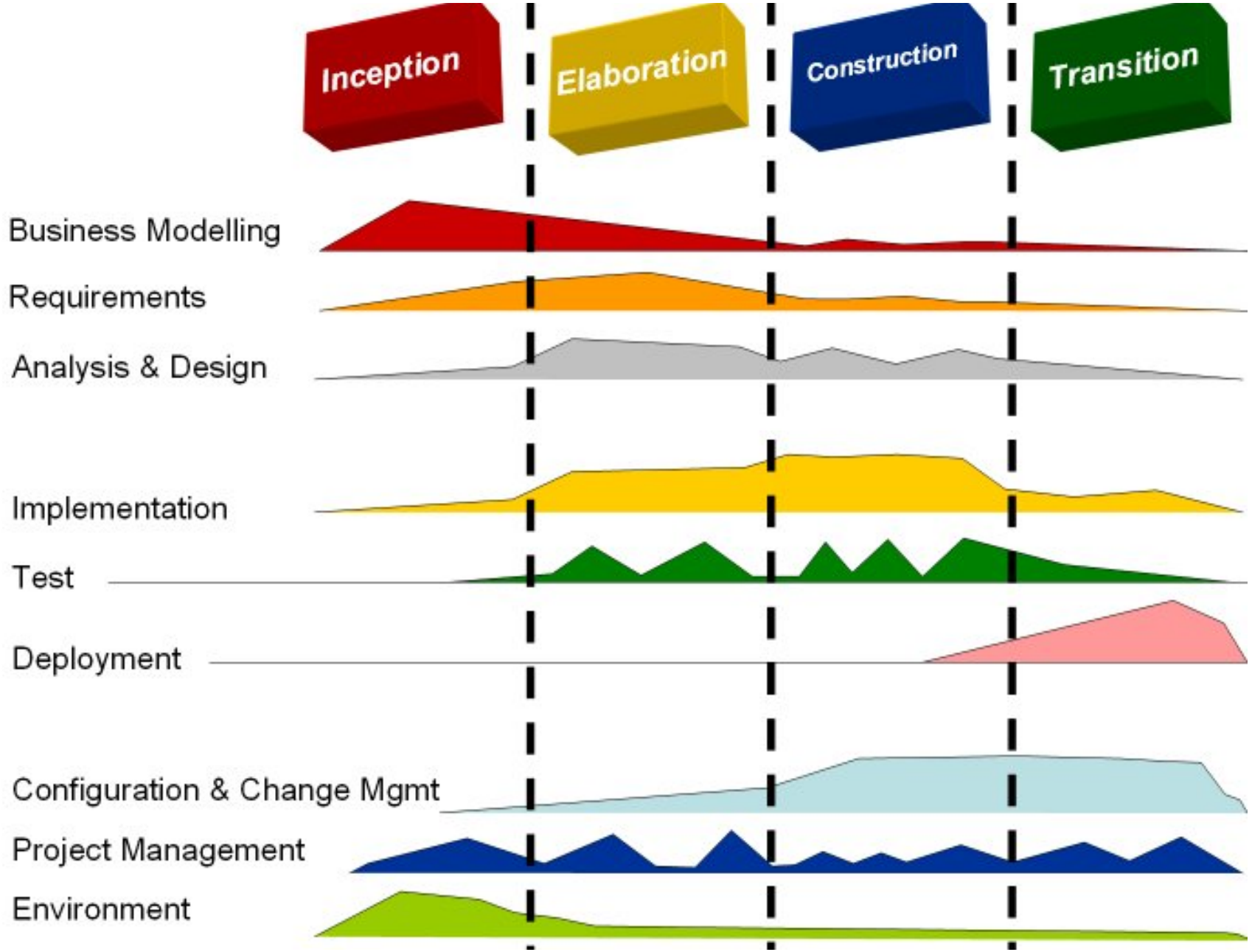
Importante

Desarrollo de software, no es **programar**

El **desarrollo de software** incluye todas las disciplinas asociadas a la **ingeniería de software**

desde el **análisis** hasta la **puesta en producción**

- … el **desarrollo de aplicaciones distribuidas** sugiere por tanto consideraciones durante la ejecución de **todo ciclo de vida**


Inception
Elaboration
Construction
Transition
Business Modelling
Requirements
Analysis & Design
Implementation
Test
Deployment
Configuration & Change Mgmt
Project Management
Environment

¿Qué es una aplicación distribuida?

Es una aplicación con distintos **componentes** que se ejecutan en **entorno separados**, normalmente en diferentes plataformas conectadas a través de una red.

¿Cuáles son los **componentes** que se **distribuyen**?

¿Qué criterios se usan para determinar que conforma un **componente**?

¿A través de que mecanismos se realiza la **comunicación** entre **componentes distribuidos**?

- ¿Qué es un **componente**?

Un **componente** es una unidad modular con las interfaces requeridas bien definidas; dicha unidad es reemplazable dentro de su ambiente.

¿Que debería **saberse y/o tenerse en cuenta** para **desarrollar** una **aplicación distribuida**?

Cliente/Servidor, protocolos, Webservices,
Comunicación, Servicios, Objetos, UML,
Interfaces, Interoperabilidad, Lógica de Negocio,
Arquitectura, Paquetes, Acceso a Datos,
Interfaz de Usuario, Componentes, Capas, etc…

La **distribución** se refiere a la construcción de software por **partes**, a las cuales les son asignadas una serie de **responsabilidades** dentro de un sistema.

- Es **distribución**, habla de que las partes o **componentes** se encuentras en **máquinas diferentes**, sin embargo, lo que tiene implícito, es que para realizar esta **separación física** primero debe tenerse clara la **separación lógica** de las partes de una aplicación, esto quiere decir que **programáticamente** existe una forma de **separar** o **agrupar** los **componentes.**

Lógica - física

- **Distribución lógica** se entiende como separación por **capas** (tiers) y cuando hablamos de **distribución física** se usa el térmico separación en **Niveles** (layers).

- La separación por **capas** y **niveles** hace parte de la **arquitectura del sistema** y es definida por el **arquitecto** del sistema.

- Tanto la distribución **lógica** como **física** se hacen con base en las necesidades **técnica,** de **diseño** y/o **negocio.**

- Identificar correctamente estas necesidades necesita de habilidades, conocimiento y experiencia. Pero, por donde comenzamos.

- Las **capas** dentro de una **arquitectura** son un conjunto de **servicios** especializados que pueden ser accesibles por múltiples clientes y que deben ser fácilmente reutilizables.

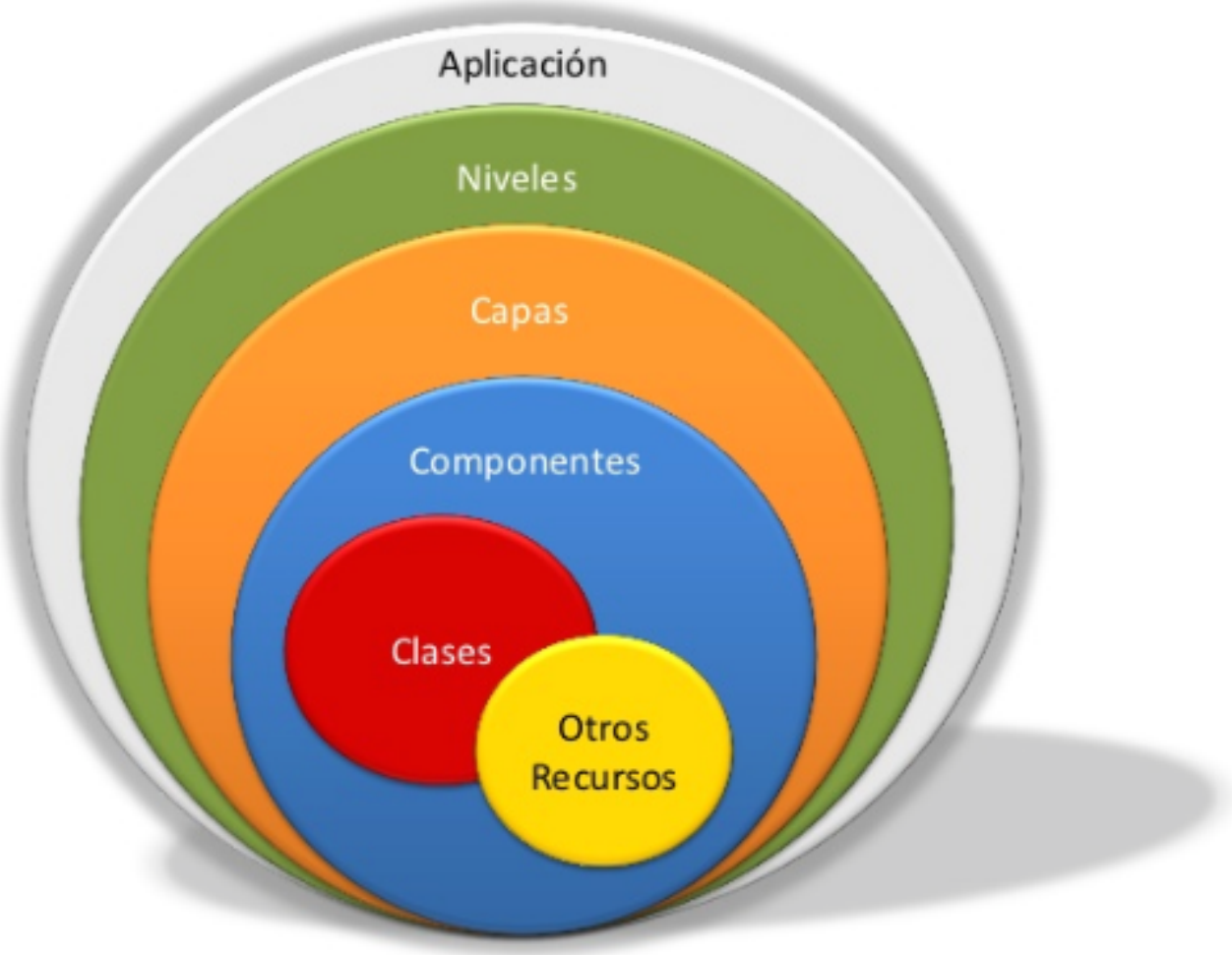
- Además, las **capas**, según el **escenario** y **tipo de aplicación**, están separadas **físicamente**.

- Una **capa** puede ser a su vez un **nivel.**

- Una **capa** puede contener **muchos componentes**, un mismo **componente** puede ubicarse en **varias capas** de acuerdo con su naturaleza y a las consideraciones explícitas de la **arquitectura.**

- Cada **componente** de un sistema puede verse como un **paquete** o **módulo.**

- Un **componente** esta compuesto por elementos que pueden ser **clases** y/o **recursos complementarios** como archivos de configuración, imágenes, entre otros.


Aplicación
Niveles
Capas
Componentes
Clases
Otros
Recursos

Patrones de diseño

- ¿Qué es un Patrón de Diseño?

- “Los patrones de diseño son el esqueleto de las soluciones a problemas en el desarrollo de software”

- Brindan una solución ya aprobada y documentada a problemas de desarrollo de software que están sujetos a contextos similares.

Aspectos de un Patrón

- Nombre del Patrón
- Problema (cuando aplicar un Patrón)
- La solución (descripción abstracta del problema)
- Consecuencias (costos y beneficios)

Clasificación

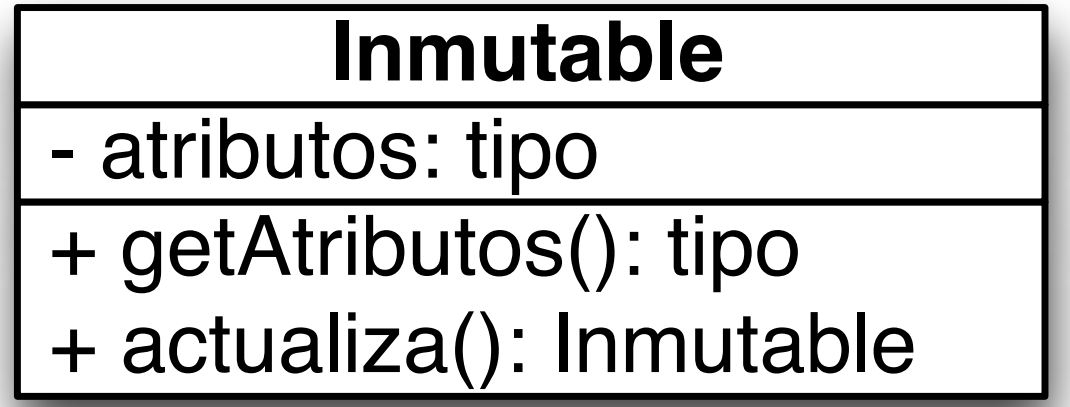
- **Patrones Creacionales**: Inicialización y configuración de objetos.

- **Patrones Estructurales**: Separan la interfaz de la implementación. Se ocupan de cómo las clases y objetos se agrupan, para formar estructuras más grandes.

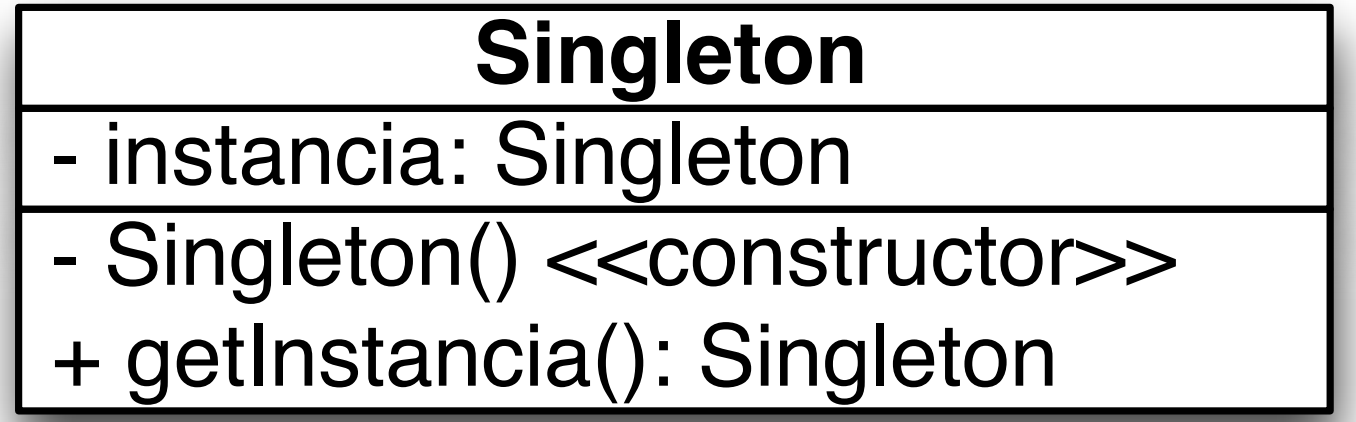
- **Patrones de Comportamiento**: Más que describir objetos o clases, describen la comunicación entre ellos.

Patrones creacionales

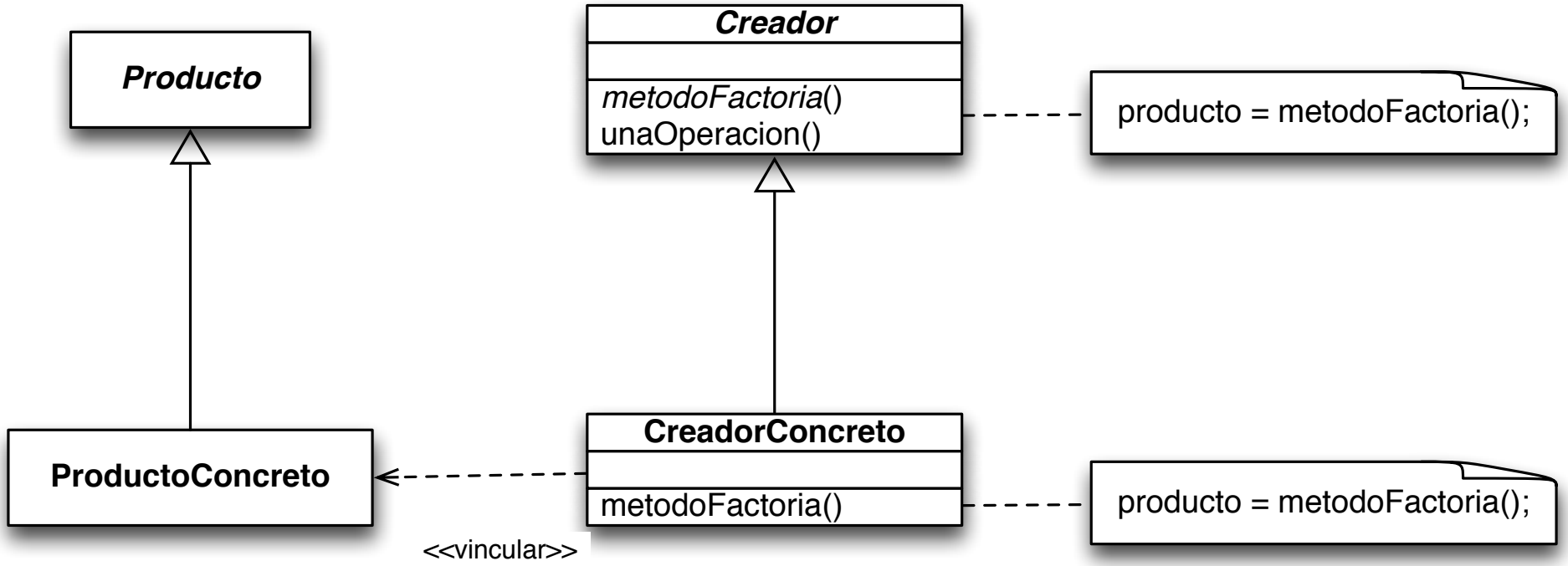
- Inicialización y configuración de objetos.
- Inmutable



- Singleton


Singleton
- instancia: Singleton
- Singleton() <<constructor>>
+ getInstancia(): Singleton

- Factory Method



Patrones Creacionales

- **Fábrica abstracta (Abstract Factory)**

El problema a solucionar por este patrón es el de crear diferentes familias de objetos, como por ejemplo la creación de interfaces gráficas de distintos tipos (venta, menú, botón, etc.).

Método de Fabricación

- **Factory Method**
- Parte del principio de que las subclases determinan la clase a implementar.

```
public class ConcreteCreator extends Creator
  {
  protected Product FactoryMethod()
      {
            return new ConcreteProduct();
      }
}
public interface Product{}
public class ConcreteProduct implements Product{}
      public class Client
      {
            public static void main(String args[])
            {
                  Creator UnCreator;
                  UnCreator = new ConcreteCreator();
                  UnCreator.AnOperations();
            }
      }
```

Prototipado

- **Pototype**

- Se basa en la clonación de ejemplares copiándolos de un prototipo.

- **Singleton**

- Restringe la instanciación de una clase o valor de un tipo a un solo objeto.

```
private static object syncRoot = new Object();
private Singleton()
{
      System.Windows.Forms.MessageBox.Show("Nuevo Singleton");
}
public static Singleton GetInstance
{
      get
      {
            if (instance == null)
            {
                  lock(syncRoot)
                  {
                        if (instance == null)
                              instance = new Singleton();
                  }
            }
            return instance;
```

MVC

- **Model View Controler**

- Este patrón plantea la separación del problema en tres capas: la capa **model**, que representa la realidad; la capa **controler**, que conoce los métodos y atributos del modelo, recibe y realiza lo que el usuario quiere hacer; y la capa **vista**, que muestra un aspecto del modelo y es utilizada por la capa anterior para interaccionar con el usuario.

Patrones Estructurales

- **Adaptador (Adapter):** Convierte una interfaz en otra.
- **Puente (Bridge):** Desacopla una abstracción de su implementación permitiendo modificarlas independientemente.
- **Objeto Compuesto (Composite):** Utilizado para construir objetos complejos a partir de otros más simples, utilizando para ello la composición recursiva y una estructura de árbol.

- **Envoltorio (Decorator):** Permite añadir dinámicamente funcionalidad a una clase existente, evitando heredar sucesivas clases para incorporar la nueva funcionalidad.

- **Fachada (Facade):** Permite simplificar la interfaz para un subsistema.

- **Peso Ligero (Flyweight):** Elimina la redundancia o la reduce cuando tenemos gran cantidad de objetos con información idéntica.

- **Apoderado (Proxy):** Un objeto se aproxima a otro.

Patrones de Comportanmiento

- **Cadena de responsabilidad (Chain of responsability):** La base es permitir que más de un objeto tenga la posibilidad de atender una petición.

- **Orden (Command):** Encapsula una petición como un objeto dando la posibilidad de “deshacer” la petición.

- **Intérprete (Interpreter):** Intérprete de lenguaje para una gramática simple y sencilla.

- **Mediador (Mediator):** Coordina las relaciones entre sus asociados. Permite la interacción de varios objetos, sin generar acoples fuertes en esas relaciones.
- **Recuerdo (Memento):** Almacena el estado de un objeto y lo restaura posteriormente.
- **Observador (Observer):** Notificaciones de cambio de estado de un objeto.

```
Public Class Articulo
    Delegate Sub DelegadoCambiaPrecio(ByVal unPrecio As Object)
    Public Event CambiaPrecio As DelegadoCambiaPrecio
    Dim _cambiaPrecio As Object
    Public WriteOnly Property Precio()
        Set(ByVal value As Object)
            _cambiaPrecio = value
            RaiseEvent CambiaPrecio(_cambiaPrecio)
        End Set
    End Property
End Class
Public Class ArticuloObservador
    Public Sub Notify(ByVal unObjecto As Object)
        Console.WriteLine("El nuevo precio es:" & unObjecto)
    End Sub


End Class
```

- **Estado (Server):** Se utiliza cuando el comportamiento de un objeto cambia dependiendo del estado del mismo.
- **Estrategia (Strategy):** Utilizado para manejar la selección de un algoritmo.
- **Método plantilla (Template Method):** Algoritmo con varios pasos suministrados por una clase derivada.
- **Visitante (Visitor):** Operaciones aplicadas a elementos de una estructura de objetos heterogénea.

- **Design Patterns (Elements of Reusable Object-Oriented Software)**, por Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

Frameworks

Se refiere a un “ambiente de trabajo y ejecución”. Por ejemplo, .Net es considerado un *framework* para desarrollo de aplicaciones (Windows).

En general los *frameworks* son soluciones completas que contemplan herramientas de apoyo a la construcción (ambiente de desarrollo) y motores de ejecución (ambiente de ejecución).

- Son diseñados con la intención de facilitar el desarrollo de software, permitiendo a los diseñadores y programadores pasar más tiempo identificando requerimientos de software que tratando con los tediosos de bajo nivel de proveer funcionalidad.

- FrameWork puede ser algo tan grande como “.NET” o Java, pero también el concepto se aplica a ámbitos mas específicos, por ejemplo; dentro de Java en el ámbito especifico de aplicaciones Web tenemos los framework: Struts, “Java Server Faces”, o Spring.

Struts, “Java Server Faces”, o Spring, en la practica son conjuntos de librerías (API’s) para desarrollar aplicaciones Web , más librerías para su ejecución (o motor), y más un conjunto de herramientas para facilitar esta tarea (debuggers, ambientes de desarrollo como Eclipse, etc).

Spring

Un *framework* de código abierto de desarrollo de aplicaciones para la plataforma Java.

- No obliga a usar un modelo de programación en particular.
- Se ha popularizado en la comunidad de programadores en Java.
- Considerado una alternativa y sustituto del modelo de Enterprise JavaBean.
- Por su diseño el framework ofrece mucha libertad a los desarrolladores en Java y soluciones muy bien documentadas y fáciles de usar para las prácticas comunes en la industria.

Hibernate

Es una herramienta de Mapeo objeto-relacional (ORM) para la plataforma Java. que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones en los beans de las entidades que permiten establecer estas relaciones.

- Hibernate es software libre.
- Permite al desarrollador detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen.
- Manipula los datos de la base operando sobre objetos, con todas las características de la POO.
- Converte los datos entre los tipos utilizados por Java y los definidos por SQL.
- Genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todos los motores de bases de datos con un ligero incremento en el tiempo de ejecución.

JSF

Tecnología y framework para aplicaciones Java basadas en web que simplifica el desarrollo de interfaces de usuario en aplicaciones Java EE. JSF usa JavaServer Pages (JSP) como la tecnología que permite hacer el despliegue de las páginas, pero también se puede acomodar a otras tecnologías como XUL. JSF incluye:

- Un conjunto de APIs para representar componentes de una interfaz de usuario y administrar su estado, manejar eventos, validar entrada, definir un esquema de navegación de las páginas y dar soporte para internacionalización y accesibilidad.
- Un conjunto por defecto de componentes para la interfaz de usuario.
- Dos bibliotecas de etiquetas personalizadas para JavaServer Pages que permiten expresar una interfaz JavaServer Faces dentro de una página JSP.
- Un modelo de eventos en el lado del servidor.
- Administración de estados.
- Beans administrados.

Struts

- Es un framework de la capa de presentación que implementa el patrón de patrón MVC en Java. Evidentemente, como todo framework intenta simplificar notablemente la implementación de una arquitectura según el patrón MVC.

- El mismo separa muy bien lo que es la gestión del workflow de la aplicación, del modelo de objetos de negocio y de la generación de interfaz.

Hibernate

- Como la definen sus autores, es una **herramienta opensource** de mapeo objeto/relacional (ORM) para ambientes Java.

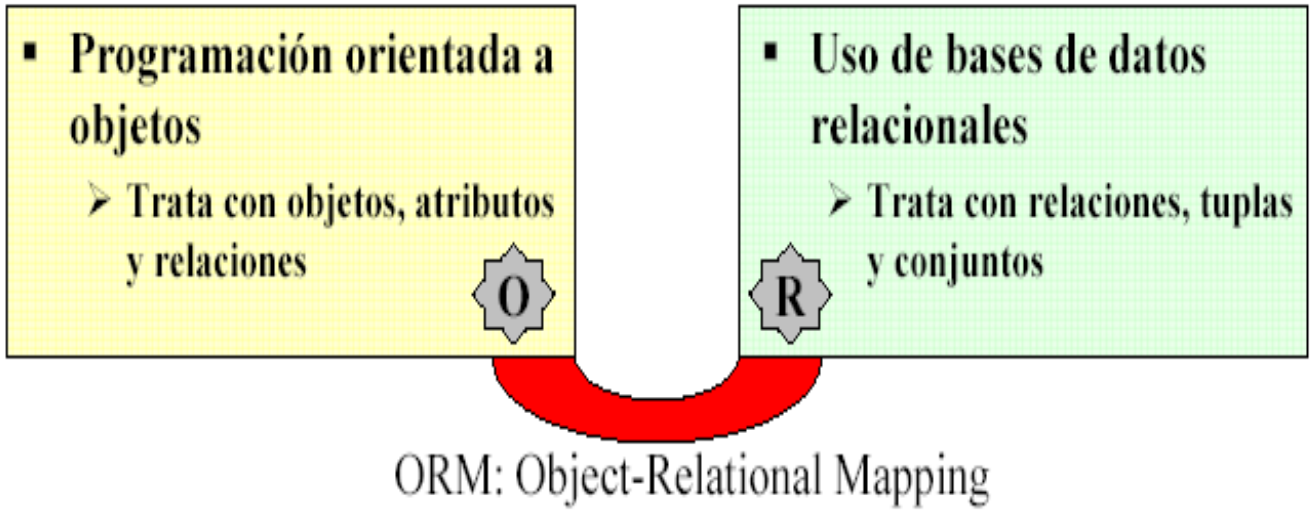
- Utiliza en vez de ello el mecanismo de **reflexión** de Java.

Características

Hibernate – Características

- No intrusivo (estilo POJO). Plain Old Java Object
- Muy buena documentación.
- Comunidad activa con muchos usuarios
- Transacciones, caché, asociaciones, polimorfismo, herencia, lazy loading, persistencia transitiva, estrategias de fetching.
- Potente lenguaje de consulta (HQL): subqueries, outer joins, ordering, proyeccion (report query), paginación.
- Fácil testeo.
- No es estandard.

¿Porqué usar Hibernate?



- **Problema**: un 35% del código de una aplicación para realizar la correspondencia O ↔ R
- **Solución**: utilizar una ORM, por ejemplo Hibernate

El modelo relacional trata con relaciones, tuplas y conjuntos y es muy matemático por naturaleza.

Sin embargo, el paradigma orientado a objetos trata con objetos, sus atributos y relaciones entre objetos.

Cuando se quiere hacer que los objetos sean persistentes utilizando para ello una base de datos relacional, uno se da cuenta de que hay una desavenencia entre estos dos paradigmas, la también llamada **diferencia objeto-relacional** (object – relational gap).

Un mapeador objeto-relacional **(ORM)** nos ayudará a evitar esta diferencia.

¿Cómo se manifiesta esta brecha entre ambos paradigmas?

Si estamos utilizando objetos en nuestra aplicación y en algún momento queremos que sean **persistentes**, normalmente abriremos una conexión JDBC, crearemos una sentencia SQL y copiaremos todos los valores de las propiedades sobre una **PreparedStatement** o en la cadena SQL que estemos construyendo.

¿Qué pasa con las asociaciones?

¿Y si el objeto contiene a su vez a otros objetos?

¿Los almacenaremos también en la Base de Datos?

¿Automáticamente?

¿Manualmente?

¿Qué haremos con las claves ajenas?

Básicamente, una ORM intenta hacer todas estas tareas pesadas por nosotros. Con una buena ORM, sólo tendremos que definir la forma en la que establecemos la correspondencia entre las clases y las tablas una sola vez (indicando que propiedad se corresponde con que columna, que clase con que tabla, etc.).

Después de esto, podremos hacer cosas como utilizar **POJO** de nuestra aplicación y decirle a nuestra ORM que los haga persistentes, con una instrucción similar a esta: **orm.save(myObject)**

Hibernate Query Language HQL

Hibernate proporciona además un lenguaje para el manejo de consultas a la base de datos.

Este lenguaje es similar a SQL y es utilizado para obtener objetos de la base de datos según las condiciones especificadas en el HQL.

El uso de HQL nos permite usar un lenguaje intermedio que según la base de datos que usemos y el dialecto que especifiquemos será traducido al SQL dependiente de cada base de datos de forma automática y transparente.

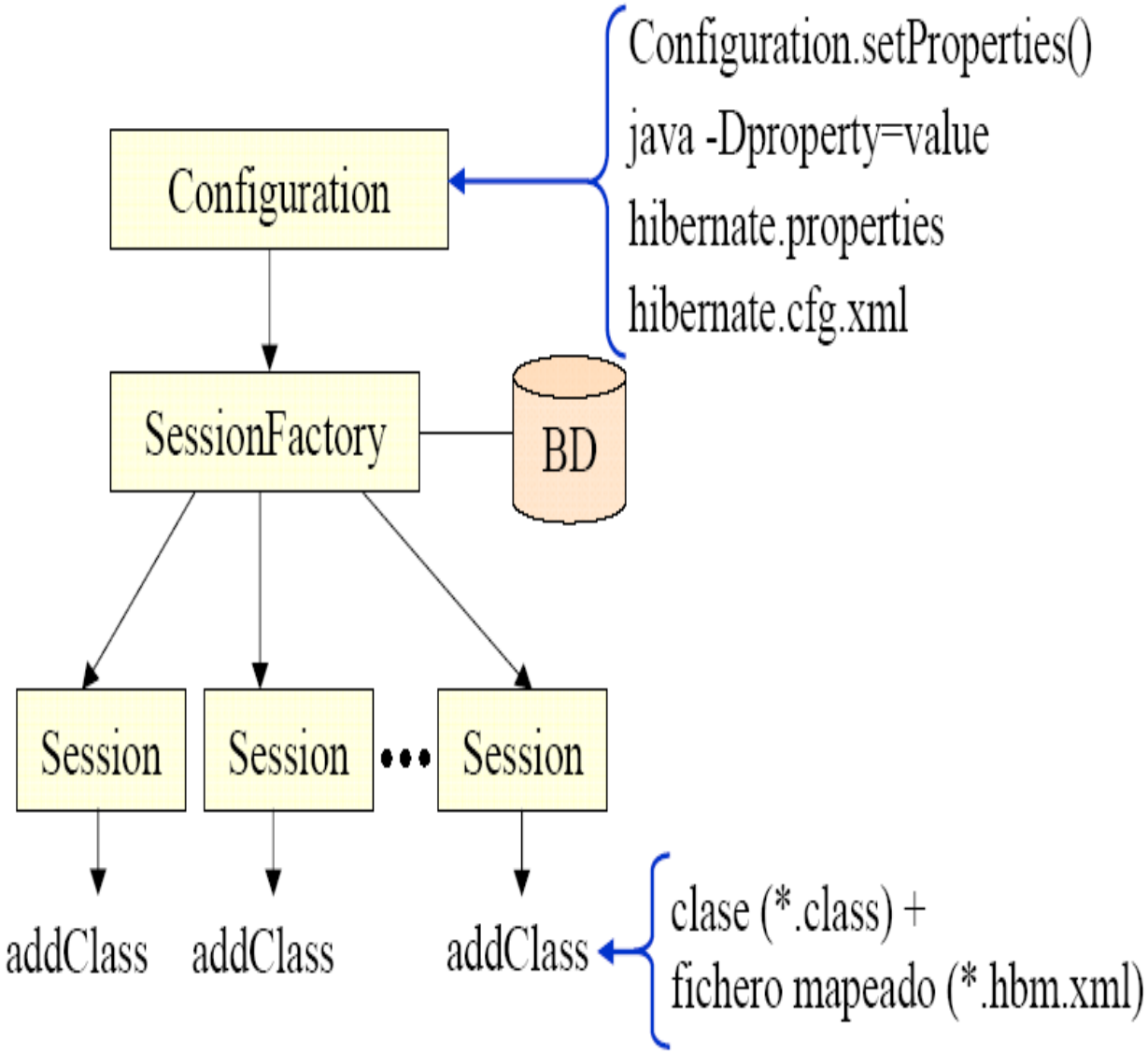
Hibernate mediante un objeto especial, quizás el concepto clave más importante dentro Hibernate, que es la **Sesión** (**clase Session**).

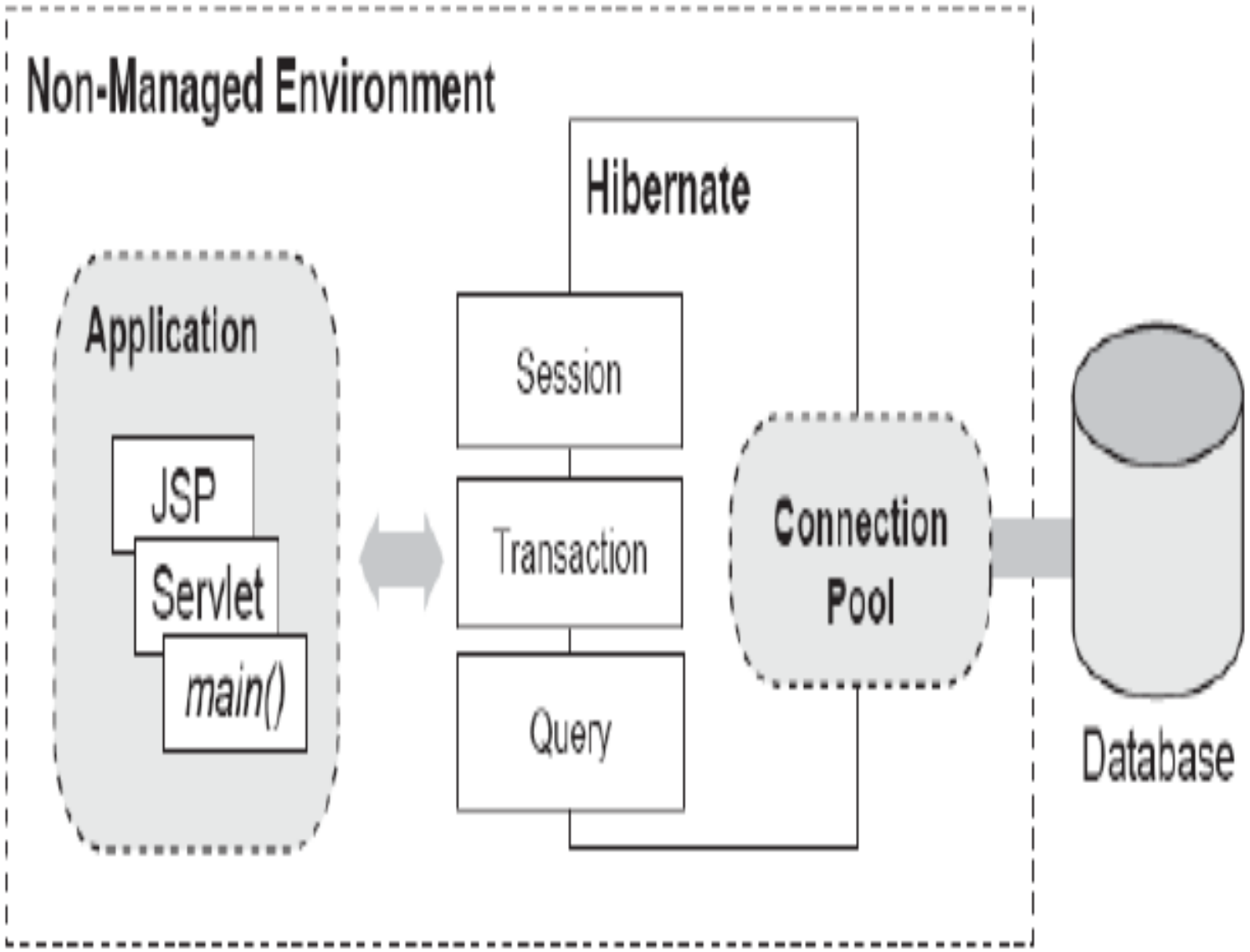
Se puede equiparar a grandes rasgos al concepto de conexión de JDBC y cumple un papel muy parecido, es decir, sirve para delimitar una o varias operaciones relacionadas dentro de un proceso de negocio, demarcar una transacción y aporta algunos servicios adicionales como una caché de objetos para evitar interacciones innecesarias contra la BD.

En este sentido, la clase Session ofrece métodos como **save(Object object), createQuery(String queryString), beginTransaction(), close()** para interactuar con la BD tal como estamos acostumbrados a hacerlo con una conexión JDBC (de hecho "envuelve" una conexión JDBC), pero con una diferencia:

mayor simplicidad, es decir, guardar un objeto, por ejemplo, consiste en algo así como **session.save(miObjecto)**, sin necesidad de especificar una sentencia SQL.

Configuración




Non-Managed Environment
Hibernate
Application
JSP
Servlet
main()
Session
Transaction
Query
Connection Pool
Database