MEDAC
Instituto Oficial de Formación Profesional

ACCESO A DATOS
TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Programación de componentes de acceso a datos

15

ÍNDICE

OBJETIVOS

Aprenderemos el concepto de componente Java.

Veremos sus propiedades, atributos y eventos.

Persistiremos distintos objetos.

Veremos herramientas visuales y empaquetado de componentes.

/ 1. Introducción y contextualización práctica

Comenzaremos esta última unidad del módulo, conociendo el significado de los componentes en java orientados a *beans*: EJB (*Enterprise Java Bean*). También estudiaremos sus propiedades, atributos y diferentes eventos, así como su propagación

Repasaremos algunos objetos de persistencia JPA que ya vimos en unidades anteriores pero que nos servirán de gran ayuda para el desarrollo de componentes.

Por último, nombraremos algunos de los IDEs de desarrollo que podremos utilizar para programar componentes en Java.

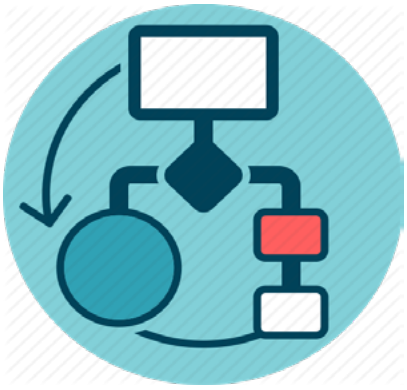

Fig. 1. Componentes.

Planteamiento del caso práctico inicial

A continuación, vamos a plantear un caso práctico a través del cual podremos aproximarnos de forma práctica a la teoría de este tema.

Escucha el siguiente audio donde planteamos la contextualización práctica de este tema, encontrarás su resolución al finalizar la unidad.

Audio Intro. "Usabilidad Entity Transaction"

https://bit.ly/33cux5l

/ 2. Concepto y características de un componente

Generalmente, un componente es una **unidad de *software* que encapsula un segmento de código con ciertas funciones**.

Los estilos de estos componentes pueden ser estilos proporcionados por el entorno de desarrollo (incluidos en la interfaz de usuario) o pueden ser no visuales, siendo sus funciones similares a las de las bibliotecas remotas. Los componentes del *software* tienen las siguientes características principales:

- **Un componente es una unidad ejecutable** que se puede instalar y utilizar de forma independiente.
- Puede **interactuar y operar** con otros componentes desarrollados por terceros, es decir, empresas o desarrolladores pueden utilizar componentes y agregarlos a las operaciones que se realizan, lo que significa que pueden estar compuestos por estos componentes.
- **No tienen estado**, al menos su estado no es visible desde 'fuera'.
- La programación orientada a componentes o un componente, **se puede asociar a los distintos engranajes electrónicos** que, en su conjunto, forman un sistema más grande.

Podemos afirmar que un componente **será formado por** los siguientes elementos:

1. **Atributos, operaciones y eventos**: Es parte de la interfaz del componente. Todo en su junto representaría el nivel sintáctico.
2. **Comportamiento**: Representa la parte semántica.
3. **Protocolos y escenarios**: Representa la compatibilidad de las secuencias de los mensajes con otros componentes, y la forma de actuar que tendrá el componente en diferentes escenarios donde llegará a ser ejecutado.
4. **Propiedades**: Las diferentes características que puede tener el componente.

Fig. 2. Concepto de componente.

/ 3. Propiedades y atributos de un componente: indexadas y simples

Las propiedades del componente determinan su estado y lo distinguen del resto. Se ha adelantado anteriormente que estos componentes carecen de estado, pero estos componentes tienen una serie de propiedades a las que se puede acceder y modificar de diferentes formas. **Las propiedades de los componentes se dividen en simples, indexadas, compartidas y restringidas**.

Audio 1. "Alcance de las propiedades de los beans de Java"

https://bit.ly/3kJ4M28

Se puede verificar y modificar las propiedades del componente accediendo al método del componente o la función de acceso. Hay dos tipos de estas funciones:

- **El método *get*** se utiliza para consultar o leer el valor de un atributo. La sintaxis general es la siguiente:

```
Tipo getNombrePropiedad()
```

- **El método *set*** se utiliza para asignar o cambiar el valor de un atributo. Su sintaxis general en Java es:

```
SetNombrePropiedad(Tipo valor)
```

Centrándonos en las propiedades o atributos indexados y simples, podemos decir, que los **atributos simples** son atributos que **representan un solo valor.** Por ejemplo, si hay un botón en la interfaz gráfica, los atributos simples serán atributos relacionados con su tamaño, color de fondo, etiqueta, etc.

Fig. 3. Propiedades indexadas en Java.

Además de las propiedades de los valores simples y únicos, existe otro tipo de propiedades más complejas que son muy similares a un conjunto de valores: **los índices.** Todos los elementos de este tipo de atributo comparten el mismo tipo y se puede acceder a ellos por índice.

Concretamente, se puede acceder mediante el método de acceso que se mencionó anteriormente *(get / set)*, aunque la llamada de estos métodos variará, ya que solo se puede acceder a cada propiedad a través de su índice.

La sintaxis que encontramos en Java sería:

```
setPropertyName (int index, PropertyType value)
```

/ 4. Propiedades y atributos de un componente: compartidas y restringidas

Además de los atributos simples y los atributos indexados, los componentes también tienen otros dos tipos de atributos:

- **Atributos compartidos.**
- **Atributos restringidos.**

Los atributos **compartidos** se refieren a los datos mediante los que informa a todos los interesados sobre el atributo, cuando cambian, por lo que solo se les notifica cuando cambia el atributo. El mecanismo de notificación se basa en **eventos**, es decir, hay un componente de origen que notifica al componente receptor cuando un atributo compartido cambia a través de un evento. Debe quedar claro que estas propiedades no son bidireccionales, por lo que el componente receptor no puede responder.

Para que un componente permita propiedades compartidas, debe admitir dos métodos para registrar componentes que estén interesados en cambios de propiedad (uno para agregar y otro para eliminar). Su sintaxis general en Java sería:

```
addPropertyChangeListener (PropertyChangeListener x) y removePropertyChangeListener
(PropertyChange Listener x).
```

Por otro lado, las propiedades **restringidas** buscarán la aprobación de otros componentes antes de cambiar su valor. Al igual que con los atributos compartidos, se deben proporcionar dos métodos de registro para el receptor.

Su sintaxis general en Java sería:

```
addVetoableChangeListener (VetoableChangeListener x) y
removeVetoableChangeListener (Vetoabl eChange Listener x).
```

Las propiedades son uno de los aspectos relevantes de la interfaz del componente porque son elementos que forman parte de la vista externa del componente (representada como una vista pública). Desde la perspectiva del control y uso de componentes, estos elementos observables son particularmente importantes y son la base para caracterizar otros aspectos de los componentes.

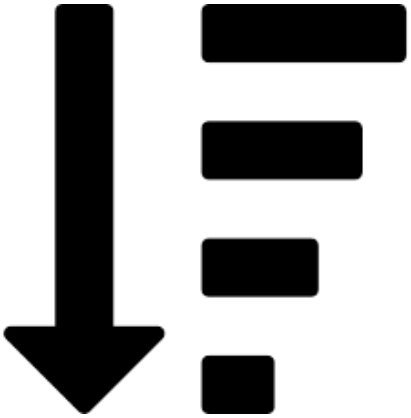

Fig. 4. Atributos.

Video 1. "POJOS y beans"
https://bit.ly/3nMTqwj

/ 5. Caso práctico 1: "Implementación de un componente y propagación"

Planteamiento: Se requiere la realización de un componente que a través de la clase *PropertyChangeSupport* ejecute los métodos propagados de:

- *addPropertyChangeListener*
- *removePropertyChangeListener*

Nudo: Tal y como hemos visto en los puntos anteriores de la unidad, nos dispondremos a usar la lógica de componentes eEJB para así poder propagar los diferentes cambios.

Desenlace:

1. En primer lugar deberemos de importar la paquetería **java.beans**

2. Luego crearíamos un objeto de la clase mencionada *PropertyChangeSupport*:

```
aprivate PropertyChangeSupport objetoCambios = new
PropertyChangeSupport(this);
```

Código 1. PropertyChange objeto.

Como podemos observar, el objeto del código anterior mantiene una lista de oyentes y por consiguiente, lanzará los diferentes eventos de cambio de propiedad.

3. A continuación, implementaremos los métodos que nos ofrece *PropertyChangeSupport*, simplemente envolveremos las diferentes llamadas de los métodos del objeto soportado:

```
public void addPropertyChangeListener(PropertyChangeListener listener) {
    changes.addPropertyChangeListener(listener);
  }
  public void removePropertyChangeListener(PropertyChangeListener listener) {
    changes.removePropertyChangeListener(listener);
  }
```

Código 2. Métodos.

/ 6. Eventos

La iteración entre componentes se denomina **control activo**, es decir, el funcionamiento de un componente se activa mediante la llamada de otro componente.

Además del control activo (la forma más común de invocar operaciones), existe otro método llamado **control reactivo**, que se refiere a **eventos de componentes**, como los permitidos por el modelo de componentes EJB.

En el control reactivo, los componentes pueden generar eventos correspondientes a **solicitudes de invocación de operaciones**. Posteriormente, otros componentes del sistema recolectarían estas solicitudes, y activarían llamadas a operaciones para sus propósitos de procesamiento.

Por ejemplo, cuando el puntero del mouse pasa sobre el ícono del escritorio, si la forma del ícono del escritorio cambia, el componente de icono emitiría eventos relacionados con la operación de cambiar la forma del icono.

Fig. 5. Eventos.

/ 7. Introspección

Las herramientas de desarrollo descubren las características de los componentes (es decir, propiedades de los componentes, métodos y eventos) a través de un proceso llamado **introspección**. La introspección, por tanto, se puede definir como un **mecanismo a través del cual se pueden descubrir las propiedades, métodos y eventos contenidos en el componente**.

Los componentes admiten la introspección de dos formas:

- Mediante el uso de **convenciones de nomenclatura específicas**, que se conocen al nombrar las características de los componentes. Para EJB, la clase *Introspector* comprueba el EJB para encontrar esos patrones de diseño y descubrir sus características.

- Proporcionando información clara sobre **atributos, métodos o eventos** de clases relacionadas.

En particular, EJB admite múltiples niveles de introspección. En un nivel bajo, esta introspección se puede lograr mediante la posibilidad de reflexión. Estas características permiten a los objetos Java descubrir información sobre métodos públicos, campos y constructores de clases cargados durante la ejecución del programa. La reflexión permite la introspección de todos los componentes del *software*, y todo lo que se tiene que hacer es declarar el método o variable como público para que pueda ser descubierto a través de la reflexión.

Fig. 6. Introspección.

/ 8. Persistencia del componente

Para EJB, la persistencia se proporciona con la **biblioteca JPA de Java**. Hoy en día podemos encontrar tal **especificación en Oracle**. Para usar JPA, se requiere el uso al menos del JDK 1.5 (conocido como Java 5) en adelante, ya que amplía las nuevas especificaciones del lenguaje Java, como son las anotaciones. Para utilizar la biblioteca JPA, es esencial tener un conocimiento sólido de POO (programación orientada a objetos), Java, y lenguaje de consulta estructurado.

Las clases que componen JPA son las siguientes:

- **Persistence**. La clase *javax.persistence.Persistence* contiene un método auxiliar estático, el cual usamos para tener un objeto *EntityManagerFactory* de forma independiente del que se obtiene a través de JPA.

- **EntityManagerFactory**. La clase *javax.persistence.EntityManagerFactory* extraemos objetos de tipo *EntityManager* usando el conocido patrón de Factory.

- **EntityManager**. La clase *javax.persistence.EntityManager* es la interfaz JPA principal para la persistencia de aplicaciones. Cada *EntityManager* puede crear, leer, modificar y eliminar un grupo de objetos persistentes.

- **Entity**. La clase *Entity* es una anotación Java. La encontramos al mismo nivel que las clases Java serializables. Cada una de estas entidades las representamos como registros diferentes en base de datos.

- **EntityTransaction**. Permite operaciones conjuntas con datos persistentes, de tal forma, que pueden crear grupos para persistir distintas operaciones. Si todo el grupo realiza las operaciones sin ningún problema se persistirá, de lo contrario, todos los intentos fallarán. En caso de error, la base de datos realizará *rollback* y volverá al estado anterior.

- **Query**. Cada proveedor de JPA ha implementado la interfaz *javax.persistence.Query* para encontrar objetos persistentes procesando ciertas condiciones de búsqueda. JPA utiliza JPQL y SQL para estandarizar el soporte de consultas. Podremos obtener un objeto Query a través del *EntityManager*.

Finalmente, aquellas anotaciones JPA (o anotaciones EJB 3.0) las encontraremos en *javax.Persistence*. Muchos IDE que admiten JDK5 (como Netbeans, Eclipse, IntelliJ IDEA) poseen utilidades y complementos para exportar clases de entidad usando las diferentes anotaciones JPA partiendo de la arquitectura de la base de datos.

Video 2. "Java EE tutorial"
https://bit.ly/32WddRF

/ 9. Caso práctico 2: "Persistencia de 2 objetos"

Planteamiento: a modo recordatorio de persistencia se requiere realizar un ejemplo de persistencia de 2 objetos de clase persona cuyos atributos sean según constructor:

- Nombre
- Edad
- Dirección

Se requiere usar la clase *EntityManagerFactory*.

Nudo: crearemos distintos objetos de una clase persona y los persistiremos.

Desenlace: A continuación, mostraremos el código por medio del cual persistiremos 2 objetos de la clase *persona*. Primero nos crearemos los diferentes objetos mediante constructor:

```
Persona persona1 = new Persona("pedro",25, "calle 1");
Persona persona2 = new Persona("pedro",25, "calle 7");
```

Código 3. Objetos persona.

Y después, procedemos a crear *EntityManagerFactory*, y a persistir los diferentes objetos:

```
EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("FactoryPersonas");
EntityManager em = emf.createEntityManager();
try {
   em.getTransaction().begin();
   em.persist(persona1);
   em.persist(persona2);
   em.getTransaction().commit();
} catch (Exception e) {
   e.printStackTrace();
}finally {
   em.close();
}
```

Código. 4. Persistiendo.

/ 10. Empaquetado de componentes

Un **módulo J2EE** consta de uno o más componentes J2EE del mismo tipo de contenedor, y descriptores de implementación de componentes de este tipo. El descriptor de despliegue del **módulo EJB** especifica los distintos atributos de nuestra transacción y la autorización de seguridad del EJB. Los **módulos JavaEE** sin descriptores de implementación de aplicaciones se pueden implementar como módulos separados.

Antes de EJB 3.1, todos los EJB debían estar empaquetados en estos archivos.

Como buena parte de todas las aplicaciones J2EE, contienen un ***front-end* web y un *back-end* EJB**, lo que significa, que se debe crear un .ear que contenga una aplicación con dos módulos: .war y *ejb-jar*.

Ésta es una buena práctica en el sentido de establecer una clara separación estructural entre la parte 'delantera' y la 'trasera.' No obstante, esta diferenciación será complicado delimitarla en aplicaciones simples.

Hay **cuatro tipos de módulos J2EE para aplicaciones web EJB**:

- **Módulos EJB**, que contienen archivos con clases EJB y descriptores de despliegue EJB. El módulo EJB está empaquetado como un archivo JAR con extensión .jar.
- **El módulo Web**, que contiene archivos con Servlet, archivos JSP, archivos de soporte de clases, archivos GIF y HTML y descriptores de implementación de aplicaciones web.

  El módulo web está empaquetado como un archivo JAR con una extensión *.war* (archivo web).
- **El módulo de la aplicación cliente**, que contiene archivos con clases y descriptores de la aplicación cliente.

  El módulo de la aplicación cliente está empaquetado como un archivo JAR con extensión *.jar*.
- **Módulo adaptador de recursos**, que contiene todas las interfaces, clases, bibliotecas nativas y otros documentos de Java y sus descriptores de implementación del adaptador de recursos.

Fig. 7. Empaquetado componente.

Audio 2. "Herramientas para desarrollo de componentes no visuales"

https://bit.ly/3llynKy

/ 11. Resumen y resolución del caso práctico de la unidad

En esta unidad hemos estudiado la **definición de componente en Java y sus características**.

También hemos visto los diferentes **métodos** que tendríamos disponibles a la hora de poblar las propiedades, atributos y eventos de los mismos

Hemos aprendido conceptos diferentes como el de **introspección**.

Y hemos dado un amplio repaso de todos los objetos que rodean a la **persistencia JPA** de objetos *beans* en Java

Resolución del caso práctico inicial

En el audio de introducción se nos propone trabajar sobre el objeto *EntityTransaction*, para ello veremos un ejemplo.

En este ejemplo, haremos uso de dicho objeto para realizar la persistencia en la capa de acceso a datos:

```
private static void agregarEntidad
{EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("PERSISTENCIA");
   EntityManager entityManager =
entityManagerFactory.createEntityManager();
   EntityTransaction entityTransaction = entityManager.getTransaction();
   entityTransaction.begin();
   Student estudiante = new Student("Juan", "Lopez", "micorreo@codigos.com");
   entityManager.persist(estudiante);
   entityTransaction commit();
   entityManager.close();
   entityManagerFactory.close();
}
```

Código 5. Uso Entity Transaction.

Como podemos observar en el código expuesto, englobamos en un método privado el método que nos hará dichas funciones. En él, en primer lugar, creamos una variable de tipo *EntityManagerFactory* donde damos nombre a nuestra factoría. A continuación creamos un *entityManager* del cual extraemos la transacción con *getTransaction()* para poder empezar a realizar operaciones. Creamos un estudiante 'normal', como ejemplo, lo persistimos, y con nuestra transacción hacemos *commit()* para que sea efectiva. De este modo usaremos nuestro objeto para persistir los diferentes componentes y objetos Java en general.

/ 12. Webgrafía

https://docs.oracle.com/javaee/5/tutorial