**UNIVERSIDAD NACIONAL DEL SANTA**

**FACULTAD DE INGENIERÍA**

**ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS E INFORMÁTICA**



**"DISEÑO DE PAQUETES IMPLEMENTANDO EL PATRON DAO"**

**ASIGNATURA:**

ARQUITECTURA DE SOFTWARE EMPRESARIAL

**ESTUDIANTES**

AVALOS MUÑOZ JORGE

LOZANO DIAZ OMAR

TOMAS VARGAS ELISA

VALERA GARIBAY RENATO

VALERIO PACHECO JHENDER

**DOCENTE**

ING. DAYAN MACEDO

**CICLO**

VI

NUEVO CHIMBOTE, 01 DE JUNIO DE 2023

# **INTRODUCCIÓN**

En la actualidad, prácticamente todas las aplicaciones requieren acceso al menos a una fuente de datos, dichas fuentes son por lo general base de datos relacionales, por lo que muchas veces no tenemos problema en acceder a los datos, sin embargo, hay ocasiones en las que necesitamos tener más de una fuente de datos o la fuente de datos que tenemos puede variar, lo que nos obligaría a refactorizar gran parte del código. Para esto, tenemos el patrón Arquitectónico Data Access Object (DAO), el cual permite separar la lógica de acceso a datos de los Bussines Objects u Objetos de negocios, de tal forma que el DAO encapsula toda la lógica de acceso de datos al resto de la aplicación.

Implementar la lógica de acceso a datos en la capa de lógica de negocio puedes ser un gran problema, pues tendríamos que lidiar con la lógica de negocio en sí, más la implementación para acceder a los datos, adicional, si tenemos múltiples fuentes de datos o estas pueden variar, tendríamos que implementar las diferentes lógicas para acceder las diferentes fuentes de datos, como podrían ser: bases de datos relacionales, No SQL, XML, archivos planos, Webservices, etc).

El patrón DAO propone separar por completo la lógica de negocio de la lógica para acceder a los datos, de esta forma, el DAO proporcionará los métodos necesarios para insertar, actualizar, borrar y consultar la información; por otra parte, la capa de negocio solo se preocupa por lógica de negocio y utiliza el DAO para interactuar con la fuente de datos.

INDICE

[**INTRODUCCIÓN** 2](#_Toc142636343)

[**MARCO TEÓRICO** 4](#_Toc142636344)

[1. DISEÑO DE PAQUETES 4](#_Toc142636345)

[1.1. Principios de diseño de paquetes 4](#_Toc142636346)

[1.2. Cohesión de paquetes 4](#_Toc142636347)

[1.3. Acoplamiento de paquetes 7](#_Toc142636348)

[2. APLICANDO EL PATRÓN DAO: 13](#_Toc142636349)

[2.1. Beneficios de esta Aplicación: 13](#_Toc142636350)

[EJERCICIO PRÁCTICO 14](#_Toc142636351)

[CONCLUSIÓNES 16](#_Toc142636352)

# **MARCO TEÓRICO**

# DISEÑO DE PAQUETES

Un paquete es una pieza de software reusable y testeable que provee un conjunto de funcionalidades especificas dentro de un sistema y está formada por un grupo de clases.  Al hablar de paquetes nos referimos a un componente o una librería que genera un archivo .jar o .dll.

## Principios de diseño de paquetes

Estos principios fueron introducidos por Robert C. Martin, existen 6 principios que se aplican al diseño de paquetes, los tres primeros principios hablan acerca de la cohesión y los otros tres acerca del acoplamiento entre paquetes.

## Cohesión de paquetes

Los 3 primeros principios de diseño de paquetes señalan que debe contener cada paquete para obtener una alta cohesión.

* REP: The realease/reuse equivalency principle:

Los paquetes que son liberado pueden usarse a través de un sistema de Tracking pueden ser efectivamente reusados. Para poder reusar efectivamente un código este debe venir en una caja negra (un archivo .dll o .jar) el cual es usado, pero no cambiado por nosotros.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <?xml version="1.0" encoding="utf-8"?>  <packages>    <package id="AntiXSS" version="4.3.0" targetFramework="net40" />    <package id="Common.Logging" version="2.3.1" targetFramework="net40" />    <package id="elmah" version="1.2.2" targetFramework="net40" />    <package id="NLog" version="3.1.0.0" targetFramework="net40" />    <package id="Quartz" version="2.3" targetFramework="net40" />  </packages> |

Por el lado de .Net contamos con el administrador de paquetes [Nuget](http://en.wikipedia.org/wiki/NuGet" \t "_blank" \o "Nuget), cada vez que instalamos un paquete este registra la versión con la que estamos trabajando y ante una actualización del paquete (hecha por los responsables de su mantenimiento) somos nosotros los que decidimos si aplicamos los cambios o seguimos trabajando con la versión con la que empezamos el desarrollo.

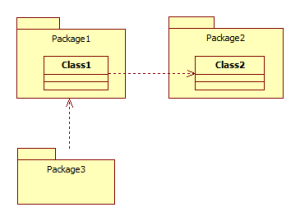
Además, para saber cuándo usar un paquete externo debemos asegurarnos que cumpla con lo siguiente:

* Documentación: Completa, acertada y actualizada.
* Mantenimiento: Bugs corregidos, las mejoras son consideradas, etc.
* Confiabilidad: No hay cambios repentinos, se mantiene las versiones anteriores.

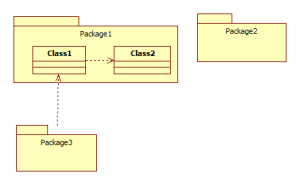
Otros administradores de paquetes:

* RubyGems / Bundler (Ruby)
* PIP / PyPI (Python)
* Packagist / Composer (PHP)
* NPM (Node.JS)
* Bower (JS, CSS, HTML)
* CocoaPods (Objective-C)
* Maven (Java)
* Lein (Clojure)
* CCP: The common closure principle

Las clases que cambian juntas son empaquetadas juntas, un cambio en un paquete afecta a todas las clases que se encuentren dentro. El objetivo de este principio es minimizar el impacto de un cambio para afectar la menor cantidad de paquetes posibles, si existen clases que tienen un alto acoplamiento estas se deben empaquetar juntas. Al seguir este principio aumentamos el mantenimiento ya que al producirse un cambio este afecte pocos paquetes que tendrán que ser recompilados.

[](https://code2read.files.wordpress.com/2015/04/common-closure-principle-antes.png)

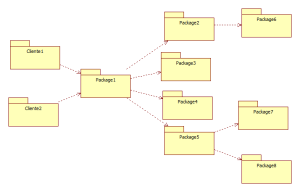
*Common closure principle – antes*

[](https://code2read.files.wordpress.com/2015/04/common-closure-principle-despues.png)

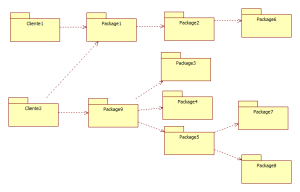
*Common closure principle – despues*

* CRP: The common reuse principle

Las clases que son usadas juntas deben empaquetarse juntas. Las clases en un paquete son reusadas juntas, si reúsas una de las clases, reúsas todas. Esto quiere decir que sin un desarrollador confía en una clase del paquete debe confiar en todas. Debemos tener en cuenta que al usar un paquete se traen todas sus referencias, lo cual origina un problema si solo estamos interesados en ciertas funcionalidades. Este principio es análogo al principio de diseño SRP, si un paquete tiene muchas responsabilidades debe partirse para no obligar a tener dependencias innecesarias. Al seguir este principio conseguiremos trabajar con paquetes específicos con responsabilidades únicas.

[](https://code2read.files.wordpress.com/2015/04/common-reuse-principle-antes.png)

*Common reuse principle – antes*

[](https://code2read.files.wordpress.com/2015/04/common-reuse-principle-despues.png)

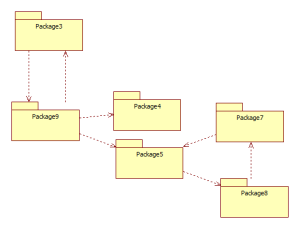
*Common reuse principle – despues*

## Acoplamiento de paquetes

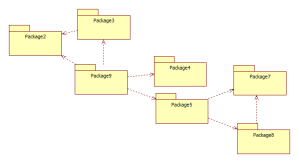
Estos 3 últimos principios de diseño de paquetes hablan acerca de las métricas que evalúan la estructura de un sistema con la finalidad de conseguir un bajo acoplamiento.

* ADP: The acyclic dependencies principle

Un paquete no debe tener dependencias cíclicas con otros paquetes. La estructura de dependencias de un paquete debe ser un Direct Acyclic Graphic esto quiere decir que no debe haber dependencias cíclicas.

[](https://code2read.files.wordpress.com/2015/04/acyclic-dependencies-principle-antes.png)

*Acyclic dependencies principle – antes*

[](https://code2read.files.wordpress.com/2015/04/acyclic-dependencies-principle-despues1.png)

*Acyclic dependencies principle – despues*

* SDP: The stable dependencies principle

La dependencia de paquetes en un diseño debe estar en dirección de la estabilidad de los paquetes. Un paquete solo debe depender de paquetes que son más estables que el para que nuestro paquete sea fácil de cambiar.

* Métrica de inestabilidad:

Esta métrica es usada para conocer la resistencia al cambio que tiene un paquete, su fórmula es la siguiente: I = Ce / (Ce+Ca). Donde:

Ca: El número de clases que se encuentran fuera del paquete y que dependen de las clases que se encuentran dentro del paquete, Afferent = entrante.

(Ce): El número de clases que se encuentran fuera del paquete y que las clases que se encuentran dentro del paquete dependen de ellas, Efferent = saliente.

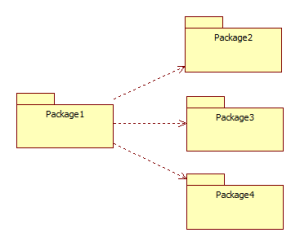
El rango para esta métrica es de 0 a 1, con I=0 indica que un paquete es completamente estable y con I>0 indica que es paquete completamente inestable y que cualquier cambio será difícil de implementar.

Métrica de inestabilidad Caso 1: Paquete 1

Ce = 3

Ca = 0

Resultado = 1, Es un paquete inestable.

[](https://code2read.files.wordpress.com/2015/04/mc3a9trica-de-inestabilidad-caso1.png)

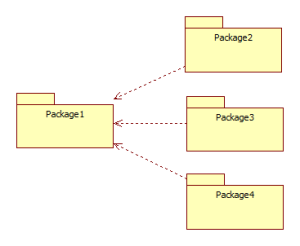
*Métrica de inestabilidad – caso1*

Métrica de inestabilidad Caso 2: Paquete 1

Ce = 3

Ca = 0

Resultado = 0, Es un paquete estable.

[](https://code2read.files.wordpress.com/2015/04/mc3a9trica-de-inestabilidad-caso2.png)

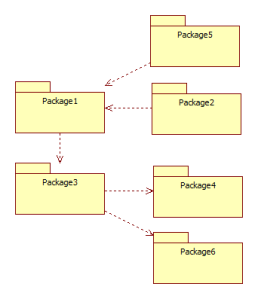
*Métrica de inestabilidad – caso2*

Métrica de inestabilidad Caso 3: Paquete 1

Ce = 1

Ca = 2

Resultado = 0.33, Es un paquete inestable.

[](https://code2read.files.wordpress.com/2015/04/mc3a9trica-de-inestabilidad-caso3.png)

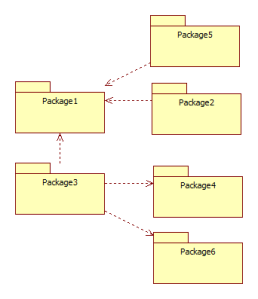
*Métrica de inestabilidad – caso3*

Métrica de inestabilidad Caso 4: Paquete 1

Ce = 0

Ca = 3

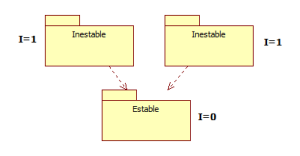
Resultado = 0, Es un paquete estable.

[](https://code2read.files.wordpress.com/2015/04/mc3a9trica-de-inestabilidad-caso4.png)

*Métrica de inestabilidad – caso4*

No todos los paquetes deben ser estables

Si todos los paquetes es un sistema fueran estables no se podrían hacer cambios. En la estructura de paquetes que diseñemos algunos paquetes deben ser estables y otros inestables. Los paquetes estables deben mantener partes que no cambian, como el diseño de alto nivel del sistema (core), y los paquetes inestables deben mantener las partes cambiantes, como las implementaciones (infraestructura).

[](https://code2read.files.wordpress.com/2015/04/configuracic3b3n-ideal-de-paquetes.png)

*Configuración ideal de paquetes*

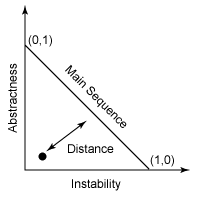
En el caso se desee cambiar o extender un paquete estable se debe aplicar el principio OCP para crear clases que son flexibles a ser extendidas sin necesidad de modificarlas, para esto debemos hacer uso de clases abstractas.

* SPA: The stable abstraction principle

Los paquetes que son estables deben ser abstractos y los paquetes inestables deben ser concretos. La abstracción de un paquete debe en proporción a su estabilidad. Un paquete debe ser abstracto de modo que su estabilidad no impida que se extienda. Además, los paquetes estables deben ser fáciles de depender de ellos y los paquetes inestables deben ser fáciles de cambiar.

* Métrica Distance from the Main Sequence:

Esta métrica indica que la abstracción y la estabilidad de los paquetes están estrechamente relacionados. También es conocido como «The Perpendicular Distance of a Package from the Idealized Line A + I = 1» el valor ideal es D = 0, es decir, a más abstracto un paquete es, más estable debería ser ya que debe tener muchos clientes que dependan de sus abstracciones. Los paquetes que son más estables (I=0) y al mismo tiempo más abstractos (A=1) son considerados buenos. Pasa lo mismo cuando un paquete es inestable (I=1) y al mismo tiempo es concreto (A=0) también son considerados buenos paquetes.

[](https://code2read.files.wordpress.com/2015/04/distance-from-the-main-sequence.png)

Distance from the Main Sequence

# APLICANDO EL PATRÓN DAO:

Cada entidad de la base de datos tiene su propio DAO en el paquete com.empresa.inventario.dataaccess.dao.

Los DAOs proporcionan métodos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos específica de cada entidad.

## Beneficios de esta Aplicación:

* Organización Clara: La estructura en capas y el diseño de paquetes facilitan la navegación y comprensión del sistema, ya que los componentes relacionados están agrupados juntos.
* Mantenibilidad: Los cambios en una capa o módulo específico tienen un impacto mínimo en otras partes del sistema.
* Reutilización: Los componentes pueden ser reutilizados en diferentes partes del sistema debido al modularidad.
* Seguridad: La capa de acceso a datos se encarga de la comunicación con la base de datos, lo que mejora la seguridad y evita la exposición directa de la base de datos en otras capas.
* Escalabilidad: La arquitectura permite agregar nuevos módulos o funcionalidades sin afectar el funcionamiento de otras partes del sistema.

Este ejemplo ilustra cómo el diseño de paquetes, junto con la organización en capas y la implementación del patrón DAO, puede ayudar a crear un sistema empresarial bien estructurado, modular y eficiente.

# EJERCICIO PRÁCTICO

Implementar una aplicación para gestionar una empresa de alquiler de vehículos. Los vehículos

se alquilan por horas debiendo tener las siguientes características:

* Carros:
* Identificador, único para cada vehículo
* Descripción, una cadena que describe al vehículo
* Categoría, existen 3 categorías (A: 18 Soles/hora, B: 14 Soles/hora, C: 10
* Soles/hora)
* Precio/hora: precio dependiendo de la categoría
* Motos:
* Identificador, único para cada vehículo
* Descripción, una cadena que describe al vehículo
* Precio/hora: 8 Soles/hora
* Bicicletas:
* Identificador, único para cada vehículo
* Descripción, una cadena que describe al vehículo
* Precio/hora: 2 Soles/hora

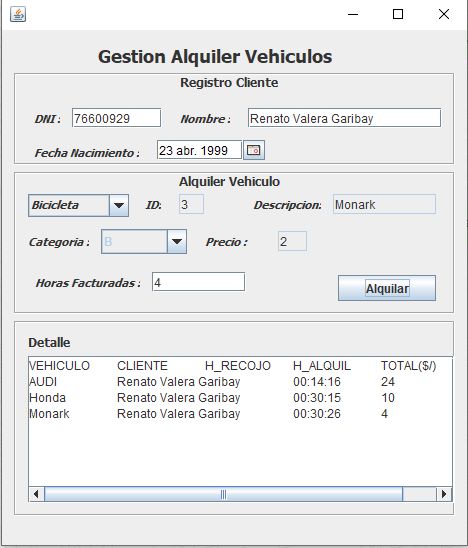
❖ Los datos que se almacenan de los clientes son: nombre, DNI y fecha de nacimiento

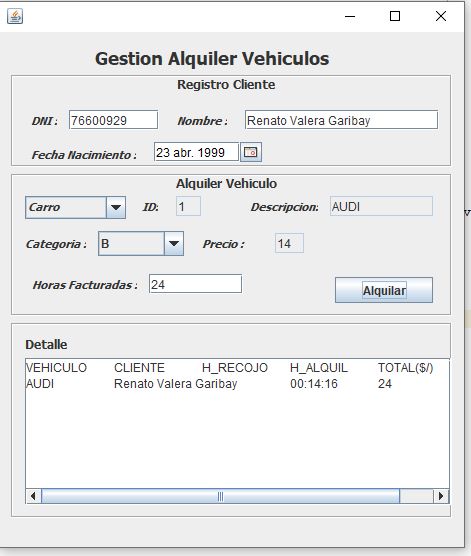
❖ Cuando un cliente alquila un vehículo, se debe llevar un registro con los siguientes datos:

vehículo alquilado, cliente, la hora de recogida y número total de horas facturadas. Sobre

el registro se podrá calcular el volumen total de horas alquiladas y el volumen total de

dinero facturado.





# CONCLUSIÓNES

Son 6 principios que se dividen en dos grupos: cohesión y acoplamiento. En cohesión tenemos: REP: The ralease/reuse equivalency principle, se enfoca en el reuso; CCP: The common closure principle, se enfoca en el matenimiento; y CRP: The common reuse principle, se enfoca en la división de dependencias. Para la parte de acoplamiento tenemos: ADP: The acyclic dependencies principle, indica que no deben existir dependencias cíclicas entre los paquetes; SDP: The stable dependencies principle, indica que se debe depender de paquetes estables; y SPA: The stable abstraction principle, indica que mientras más estable es una clase más abstracta debe ser.