**Propuesta de una arquitectura de aplicaciones Java utilizando JPA con la máxima independencia entre las capas**

Francisco Javier Gil Gala. [UO230948@uniovi.es](mailto:UO230948@uniovi.es). Universidad de Oviedo.

1. Introducción

La siguiente propuesta de arquitectura de aplicaciones esta basada en el uso de un modelo definido en 3 capas: lógica, persistencia y presentación. Este diseño intenta primar la organización de la aplicación en varios paquetes permitiendo, ente otras cosas, una gran escalabilidad en las aplicaciones construidas. Esta arquitectura no tiene solo un ámbito educativo o de investigación si no que pretende ser un completo framework para cualquier tiene de aplicación Java, desde aplicaciones de escritorio a aplicaciones móviles.

La arquitectura define una serie de clases e interfaces con objeto de estructurar toda la aplicación. Estas clases e interfaces no deben ser modificadas por el programador, ya que el comportamiento de dichas clases se encuetra parametrizado a través de un archivo de propiedades.

No estamos ante nada nuevo, simplemente es una arquitectura basada en el modelo de capas n-capas pero aplicando los principios SOLID y algunos de los patrones de diseño propuestos en [1] y [2]

1. Herramienta de gestión de proyecto propuesta

La herramienta para la gestión de proyectos utilizada ha sido Maven. Aunque no es un aspecto relevante en la arquitectura propuesta si que es una herramienta que permite no tener que implementar patrones destinados a mantener las librerias externas actualizadas.

Aunque la arquitectura este pensada para poder utilizar cualquier herramienta de gestión de protectos, Maven es una de las herramientas de gestión propicia para aplicaciones de escritorio o empresariales, como es el caso del ejemplo práctico.

Para aplicaciones Android se propone Grandle al ser la herramienta de construcción de proyectos que actualmente Android propone. Sobre la arquitectura propuesta para Android se tendrán que hacer modificaciones sobre la implementación de la capa de persistencia, ya que Android no tiene soporte directo para una implementación de JPA.

1. Diseño propuesto

El primer aspecto a comentar es el relativo a la ilustración 1, se puede observar la gestión de paquetes que propone Maven.



Ilustración 1

Observa como se separa las implementaciones concretas y el código de prueba que sirve de cobertura.



Ilustración 2

En la ilustracion 2 se observa como la distribución de los paquetes intenta eliminar la dependencia entre las distintas capas. De esta distribución de paquetes se consigue discernir una serie de capas que serán el esqueleto de cualquier aplicación diseñada con esta arquitectura.

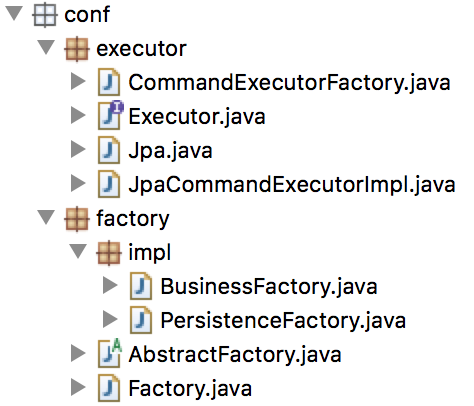


Ilustración 3

El corazón de esta arquitectura se articula en el paquete conf. Como podemos observar en la ilustración 3 principalmente tendremos dos bloques estructurados en dos paquetes: el paquete executor y el paquete factory.

* 1. Paquete executor

El paquete executor es la implementación concreta de JPA sin especificar aun que implementación concreta se esta utilizando. La idea de este conjunto de código es mantener la máxima independencia entre la implementación concreta de JPA y la capa business.

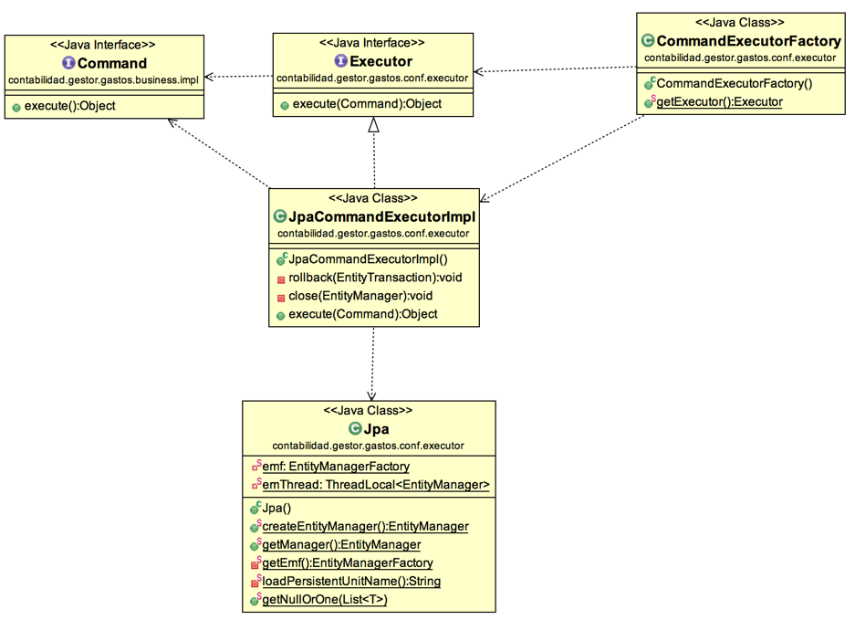


Ilustración 4

Como se puede observar en la ilustración 4 JPACommandExecutorImpl será una implementación de la interfaz Executor que recibirá una implementación concreta de la interfaz Command. Como se verá en los siguientes apartados cada una de las operaciones de negocio estará representada por una implementación de Command que será pasada por parámetro a la implementación concreta de Executor.

Ilustración 5

**private** Executor executor = CommandExecutorFactory.*getExecutor*();

Como se puede observar en la ilustración 5 en algun lugar de la capa business deberá existir una llamada a la factoria de Executor, en este caso CommandExecutorFactory, que devuelva la implementación concreta de Executor, en este caso JPACommandExecutorImpl.

Por otro lado comentar que, aunque aparezca en el diseño final, la clase JPA unicamente se encarga en gran medida de buscar la EntityManager y de crearla a partir de una la unidad de persistencia, por lo que su diseño no tiene gran interés en el diseño de esta arquitectura.

* 1. Paquete factory

Este paquete es el corazón en si de la arquitectura, se encarga tanto de instanciar las implementaciones concretas de las factorias como de los servicios que cada una de ellas proporcionará. Todo esto se encuentra parametrizado a partir de una archivo de propiedades denominado factories.properties que esta diseñado para que el propio desarrollador puede parametrizar su aplicación y seleccionar por programa que implementación utilizar.

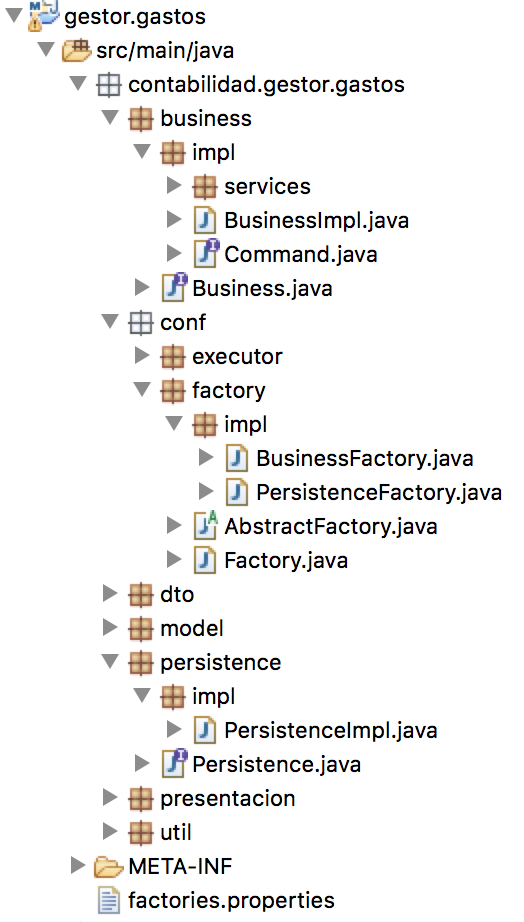


Ilustración 6

En la ilustración 6 podemos ver los paquetes y sus clases e interfaces que intervienen en la organización de la arquitectura.

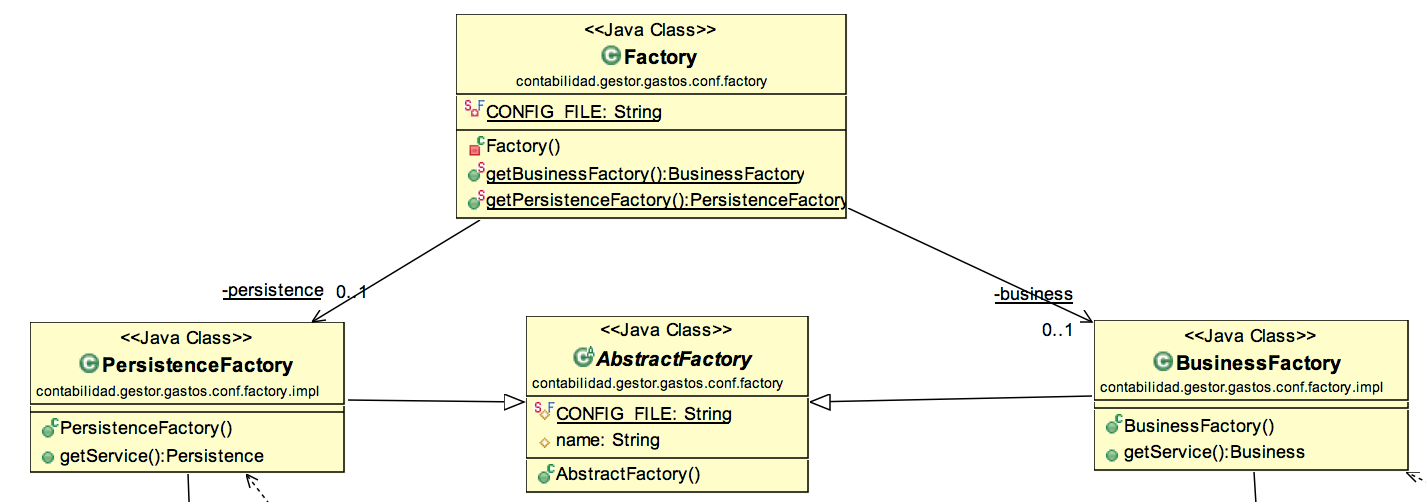


Ilustración 7

En la ilustración 7 se puede ver como será la clase Factory la encargada de instanciar las factorias de persistencia y de negocio (business) Dichas factorias extienden de la clase AbstractFactory unicamente para eliminar código duplicado.

Como se puede observar en la ilustración 8, que describe la implementación de la clase, tendremos un Singleton que negará instaciar esta clase. La idea es que unicamente instaciemos una vez las factorias y que sean comunes a toda la aplicación.

Ilustración 8

**public** **class** Factory {

**private** **static** **final** String ***CONFIG\_FILE*** = "/factories.properties";

**private** **static** BusinessFactory *business*;

**private** **static** PersistenceFactory *persistence*;

**private** Factory() { }

**public** **static** BusinessFactory getBusinessFactory(){

**if** (*business* == **null**)

**try** {

*business* = (BusinessFactory) Lector.*createClass*(***CONFIG\_FILE***, "BUSINESS\_FACTORY");

} **catch** (...) {...}

**return** *business*;

}

**public** **static** PersistenceFactory getPersistenceFactory(){

// código similar

}

Ilustración 9

**public** **class** BusinessFactory **extends** AbstractFactory {

**private** Business business;

**public** Business getService(){

**if** (business == **null**)

**try** {

business = (Business) Lector.*createClass*(***CONFIG\_FILE***, "BUSINESS\_IMPL");

} **catch (...**) {...}

**return** business;

}

}

La ilustración 9 muestra la implmentación de la factoria de negocio, de forma similar será la de la factoria de persistencia. Como se observa al igual que pasa con Factory estas clases utilizan la clase Lector para instanciar las implementaciones concretas.

Ilustración 10

**public** **class** Lector {

**public** **static** Object createClass(String file, String factoryType) {

String className = *loadProperty*(file, factoryType);

Class<?> clazz = Class.*forName*(className);

**return** clazz.newInstance();

}

**public** **static** String loadProperty(String file, String property) {

Properties p = *loadPropertiesFile*(file);

String value = p.getProperty(property);

**return** value;

}

**public** **static** Properties loadPropertiesFile(String file) {

Properties p = **new** Properties();

InputStream is = Factory.**class**.getResourceAsStream(file);

p.load(is);

is.close();

**return** p;

}

}

En la ilustración 10 se puede ver, a modo de ejemplo, el código de la clase lector, como haciendo uso de la reflexión se generan las clases a partir del archivo de propiedades factories.properties.

Este código, al igual que todo el código anterior, no deberia modificarse nunca para la aplicación, el usuario únicamente tendrá que diseñar las implementaciones concretas, pero no esta clases. Dichas implementaciones las tendrá que definir dentro de los paquetes correspondientes. En este sentido es importate definir siempre (o redefinir) las interfaces Business y Persistence con la idea de programas siempre para una interfaz y no para implementación.

Por tanto, será una buena práctica hacer que los accesos a las implementaciones concretas sean lo más transparentes posibles. Será una buena práctica definir clases Services encargadas de devolver la implementación.

Ilustración 11

**class** \_PersistenceService {

**public** **static** Persistence get(){

**return** Factory.*getPersistenceFactory*().getService();

}

}

En la ilustración 11 se puede observar el código de para acceder a los servicios de persistencia desde la capa de negocio. Como es lógico esta clase unicamente deberá ser accesible desde la capa de negocio por lo que seria una buena praxis dejarla con protección de paquete y situarla dentro del paquete business.impl.services para que unicamente los servicios concretos puedan llamarla. De una forma similar se puede hacer en la capa de presentación.

Ilustración 13

**public** **class** App {

**public** **static** **void** main(String[] args) {

\_Services.*get*().isUser("Fran", "TelecoEspartano");

}

}

Ilustración 12

**public** **class** \_Services {

**public** **static** Business get(){

**return** Factory.*getBusinessFactory*().getService();

}

}

En las ilustraciones 12 y 13 se puede observar la clase encargada de llamar a los servicios de la capa de negocio y como se realizaría la llamada desde la capa de presentación.

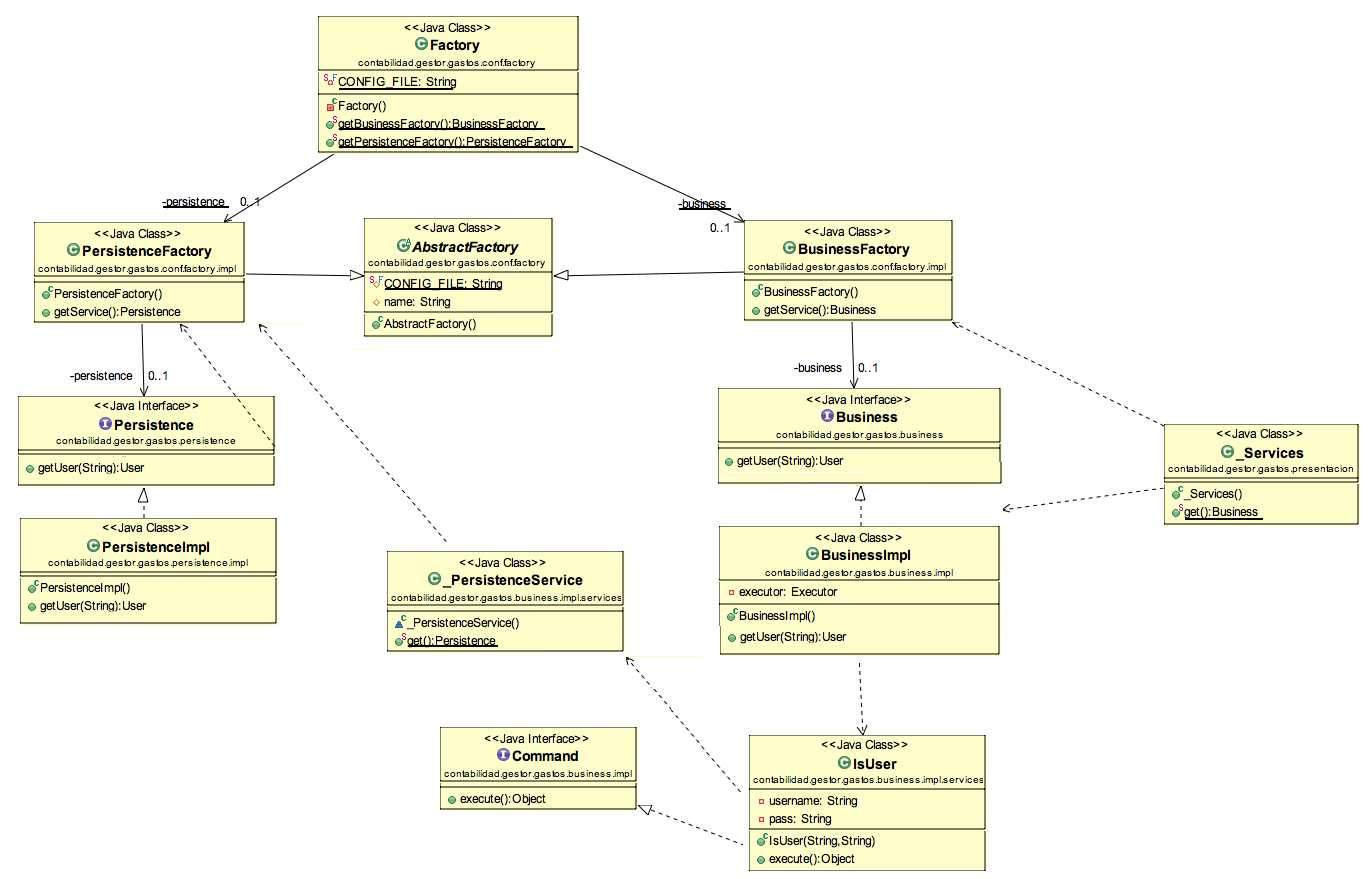


Ilustración 14

Finalmente en la ilustración 14 se muestra como quedarían las relaciones entre las distintas clases e interfaces.

* 1. Otros aspectos relevantes

Es posible que de la ilustración 14 no quede completamente claro como se deben construir las aplicaciones. En la ilustración 6 podiamos observar como era la estructuración de paquetes de la arquitectura. Se puede observar como tanto la capa de persistencia como la de negocio tienen un paquete services, es en dicho paquete donde se deben añadir las implementaciones diseñadas por el desarrollador.

* + 1. Capa de negocio

En la ilustración 15 podemos observar, a modo de ejemplo, una implementación concreta de la capa de negocio.

Se observa como cada uno de los servicios que se suministre en la capa de negocio tendrá asociado una clase que implemente Command que será ejecutado desde la implementación concreta de Business, en este caso BusinessImpl, aunque esto es solo un ejemplo la arquitectura esta diseñada para seguir este esquema de ejecución.

Ilustración 15

**public** **interface** Business {

**public** **void** newUser(DTOUser user) **throws** BusinessException;

}

**public** **class** BusinessImpl **implements** Business {

**private** Executor executor = CommandExecutorFactory.*getExecutor*();

@Override

**public** User getUser(String user) **throws** BusinessException {

**return** (User) executor.execute(**new** GetUser(user));

}

}

**public** **class** NewUser **implements** Command {

**private** DTOUser user;

**public** NewUser(DTOUser user) {

**this**.user = user;

}

@Override

**public** Object execute() **throws** BusinessException {

User u = \_PersistenceService.*get*().getUser(user.getUsername());

**if** (u != **null**)

**throw** **new** BusinessException("El usuario ya existe");

**else**

\_PersistenceService.*get*().newUser(**new** User(user.getUsername(),

user.getName(), user.getPassword()));

**return** **null**;

}

}

Tambien se observa como se utilizan objetos DTO para transferir la información desde las capas de presentación a la capa de negocio, esto es opcional pero si que es una buena práctica y la arquitectura ya viene definida para tal fin. La idea de transferencia de objetos se puede observar en la ilustración 16.

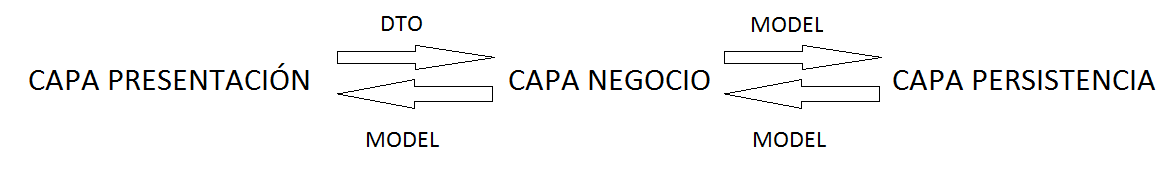


Ilustración 16

Tambien es una buena práctica tener una clase encargada de transformar objectos DTO a objectos del modelo y vicersesa, sobre todo cuando se trate de objetos con bastantes propiedades.

* + 1. Capa de persistencia

La implementación de la capa de persistencia puede hacerse de multiples formas, por ejemplo:

* Implementando Finders para las operaciones de consulta para cada una de las entidades del domino y dejar las operaciones de escritura, modificado y eliminación en las implementaciones de los Command.
* Implementando una clase que recoja todos los accesos a la base de datos.
* Implementando varias clases (por ejemplo una por entidad de dominio) que recoja todos los accesos a la base de datos.

Esta decisión será propia del programador, aunque una buena práctica sea el uso de Finders, la experiencia puede demostrar que no siempre es la mejor alternativa, sobre todo cuando el dominio del modelo es relativamente pequeño. De todas formas la arquitectura si que obliga a tener una interfaz que recoja todas las operaciones de persistencia (si se usan Finders unicamente las de consulta)

1. Conclusiones

Se puede concluir que la arquitectura propuesta es una vuelta de tuerca adicional sobre el modelo de capas tradicional. No se reinventa nada con esta propuesta, unicamente se muestra una arquitectura que desarrollada podría llegar a convertirse en un framework ligero y robusto para el diseño de aplicaciones Java.

Aunque la arquietctura vista no sea unicamente exclusiva de la tecnología Java si que en gran medida esta pensada para ser usada con lenguajes de ambito general y con soporte para el paradigma orientado a objetos como Java o C#.

Referencias bibliográficas

[1] Design Patterns, Gang of Four

[2] Publicaciones de Martin Fowler