Ingeniería del Software de Gestión 2
Grado en Ingeniería Informática en Sistemas de Información
Enseñanzas Prácticas y de Desarrollo

EPD 7: Arquitectura Software. El patrón Modelo-Vista-Controlador (MVC) en Java.Swing

Objetivos

- Afianzar el concepto de Arquitectura del Software
- Desarrollar aplicaciones en capas
- Introducir la noción de patrón arquitectónico.
- Conocer el patrón arquitectónico MVC y cómo es usado en java.Swing

Conceptos

1. Introducción.

Como en la mayoría de las ocasiones en el mundo del software, no hay una única descripción válida para definir el concepto de arquitectura del software. Actualmente existen multitud de definiciones, dentro de las cuales destacamos las realizadas por Eoin Woods, la enunciada por *Rational Unified Process*, o la más actual, recogida en el libro *Software Architecture in Practice (2nd edition)*.

1. *Eoin Woods*:

La arquitectura software es el conjunto de decisiones, que tomadas de forma incorrecta, pueden causar que tu proyecto sea cancelado.

2. Rational Unified Process, 1999:

Una arquitectura es el conjunto de decisiones significativas sobre la organización de un sistema software, la selección de elementos estructurales y sus interfaces para las que el sistema es compuesto, junto con su comportamiento especificado en la colaboración entre esos elementos, la composición de estas estructuras y los elementos que guían a los grandes subsistemas progresivamente, y el estilo de la arquitectura que guía esta organización.

3. Software Architecture in Practice (2nd edition), 2003:

La arquitectura software de un programa o sistema de computación es la estructura o estructuras del sistema, que constan de elementos software, las propiedades visibles externamente de esos elementos, y la relación entre ellos.

Las propiedades "externamente visibles" se refieren a aquellas suposiciones de otros elementos que puedan entender a un elemento, como sus servicios proporcionados, características de funcionamiento, control por defecto, uso de recurso compartido, y cosas así. Seguidamente se presentan en detalle las implicaciones de estas definiciones:

Primero, *la arquitectura define elementos*. La arquitectura expresa información sobre cómo se relacionan los elementos uno con otro. Esto significa que la arquitectura *omite* específicamente ciertas informaciones sobre los elementos que no pertenecen a su interacción. De este modo, una arquitectura es, ante todo, una *abstracción* de un sistema que suprime detalles de elementos que no afectan a cómo lo usan, son usados, relaciones con, o interaccionan con otros elementos. En casi todos los sistemas modernos, los elementos interactúan con cada uno de los otros por medio de interfaces que dividen los detalles de un elemento en partes públicas y privadas. La arquitectura es generada con la parte púbica de esta división; los detalles privados de los elementos – detalles que tiene que ver solamente con implementación interna- no son arquitecturales.

Segundo, la definición aclara que el sistema *puede y debe comprometer más de una estructura* y que ninguna de éstas puede ser *la* arquitectura. Por ejemplo, todos los proyectos no triviales están divididos en unidades de implementación; estas unidades son dadas por responsabilidades específicas, y son la base de la asignación de trabajos para grupos de programadores. Este tipo de elementos constarán de programas y datos tales que software en otras unidades de implementación puedan llamar o acceder, además de programas y datos que son privados. En grandes proyectos, los elementos, incluso, serán subdivididos y asignados a sub-equipos de trabajo. Esta es un tipo de estructura que a menudo se usaba para describir un sistema. Es una estructura muy estática, ya que se centra en la forma en la que es dividida la funcionalidad del sistema y es asignada a los grupos de

implementación. Existen otras estructuras que están mucho más centradas en la forma en la que los elementos interactúan entre ellos para llevar a cabo la funcionalidad del sistema.

Tercero, la definición implica que *cada sistema software tiene una arquitectura* debido a que cada sistema puede estar compuesto de elementos y relaciones sobre ellos. En el caso más trivial, un sistema es, en sí mismo, un elemento simple – una arquitectura probablemente inútil y poco interesante., pero una arquitectura en todo caso. Aun cuando cada sistema tenga una arquitectura, no necesariamente es conocida por todos. Desafortunadamente, una arquitectura puede existir independiente de su descripción o especificación, lo cual aumenta la importancia de la *documentación de la arquitectura y la reconstrucción de ésta*.

Cuarto, *el comportamiento de cada elemento es parte de la arquitectura* en la medida en que ese comportamiento pueda ser observado o percibido desde el punto de vista de cualquier elemento. Este comportamiento es lo que aportan los elementos para interactuar entre ellos, lo que es claramente parte de la arquitectura. Esto no significa que el comportamiento exacto y el funcionamiento de cada elemento deba ser documentado en todas las circunstancias; sino en aquellas que describan las influencias de cómo otro elemento debe ser escrito para interactuar con o influenciar en la aceptabilidad del sistema como todo.

Finalmente, la definición es indiferente de si la *arquitectura para un sistema es buena o mala*.

2. Modelo de organización.

Dentro de los sistemas modernos podemos encontrar diferentes modelos de organización. Estos modelos, que definen la estrategia básica usada para estructurar el sistema en cuestión, son diferentes dependiendo, sobre todo, de los requisitos no funcionales de dicho sistema. De esta forma, estos requisitos guiarán la elección de un modelo en capas, repositorio o cliente-servidor.

En esta práctica nos centraremos en la descripción del modelo en capas, la cual será la elección obligatoria para desarrollar los trabajos de la asignatura. Esta imposición está fundamentada en las principales ventajas de este modelo:

1. Separación de funciones: todo lo relacionado con la interfaz del usuario va en una capa, las reglas de negocio en otra y el manejo de datos en una tercera capa. No se mezcla en una capa código correspondiente a otra.
2. Reutilización: el código correspondiente a una capa puede ser reutilizado desde varias partes de la capa inmediatamente superior.
3. Escalabilidad: sabiendo dónde está el código correspondiente a cada capa, pueden realizarse modificaciones dentro de una capa para mejorar o aumentar el tamaño del sistema de software, con un mínimo impacto en las capas restantes.
4. Facilidad de mantenimiento: mediante esta división, es mucho más sencillo localizar errores en el código o efectuar mejoras.

Seguidamente, se expondrá el modelo en capas tanto para aplicaciones stand-alone como para web.

2.1. Aplicaciones en capas

La estrategia tradicional de utilizar aplicaciones compactas causa gran cantidad de problemas de integración en sistemas software complejos como pueden ser los sistemas de gestión de una empresa o los sistemas de información integrados consistentes en más de una aplicación. Estas aplicaciones suelen encontrarse con importantes problemas de escalabilidad, disponibilidad, seguridad o integración entre otros.

Para solventar estos problemas se ha generalizado la división de las aplicaciones en capas que normalmente serán tres: una capa que servirá para guardar los datos (base de datos), una capa para centralizar la lógica de negocio (modelo) y por último una interfaz gráfica que facilite al usuario el uso del sistema.

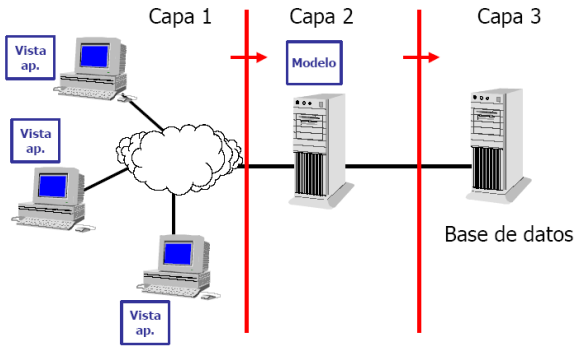


Figura 1. Arquitectura en tres capas

Si establecemos una separación entre la capa de interfaz gráfica (cliente), replicada en cada uno de los entornos de usuario, y la capa modelo, que quedaría centralizada en un servidor de aplicaciones, según el diagrama que podemos ver en la Figura 1, "Arquitectura en tres capas", obtenemos una potente arquitectura que nos otorga algunas ventajas:

1. Centralización de los aspectos de seguridad y transaccionalidad, que serían responsabilidad del modelo.
2. No replicación de lógica de negocio en los clientes: esto permite que las modificaciones y mejoras sean automáticamente aprovechadas por el conjunto de los usuarios, reduciendo los costes de mantenimiento.
3. Mayor sencillez de los clientes.

Si intentamos aplicar esto a las aplicaciones web, debido a la obligatoria sencillez del software cliente que será un navegador web, nos encontramos con una doble posibilidad:

- Crear un modelo de 4 capas, tal y como puede verse en la Figura 2, "Arquitectura web en cuatro capas", separando cliente, servidor web, modelo y almacén de datos. Lo que no permite una mayor extensibilidad en caso de que existan también clientes no web en el sistema, que trabajarían directamente contra el servidor del modelo.

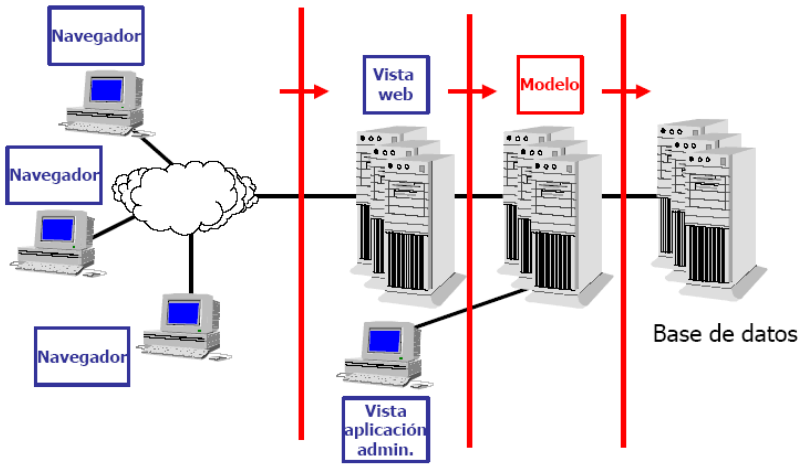


Figura 2. Arquitectura web en cuatro capas

- Mantener el número de capas en 3, como se ve en la Figura 3, "Arquitectura web en 3 capas", integrando interfaz web y modelo en un mismo servidor aunque conservando su independencia funcional. Ésta es la distribución en capas más común en las aplicaciones web.

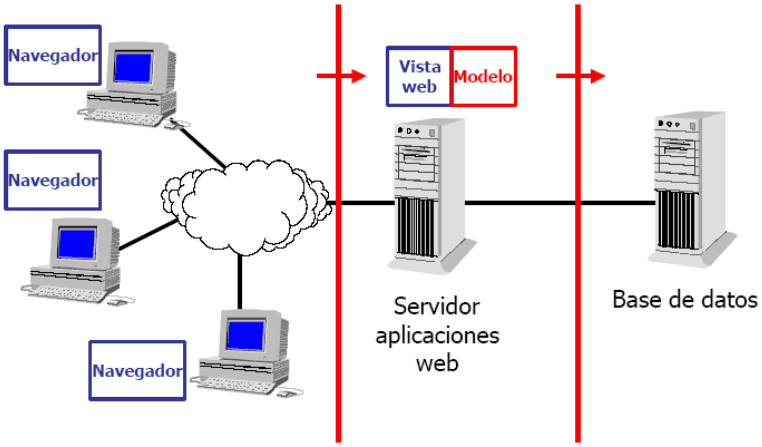


Figura 3. Arquitectura web en 3 capas

3. Patrón arquitectónico Modelo-Vista-Controlador (MVC)

La arquitectura Model-View-Controller surgió como patrón arquitectónico para el desarrollo de interfaces gráficos de usuario en entornos Smalltalk. Su concepto se basaba en separar el modelo de datos de la aplicación de su representación de cara al usuario y de la interacción de éste con la aplicación, mediante la división de la aplicación en tres partes fundamentales (ver figura 4):

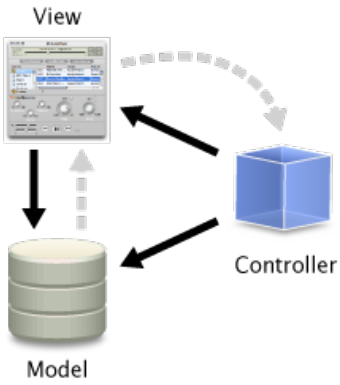


Figura 4. Arquitectura MVC. Un ejemplo de arquitectura en 3 capas

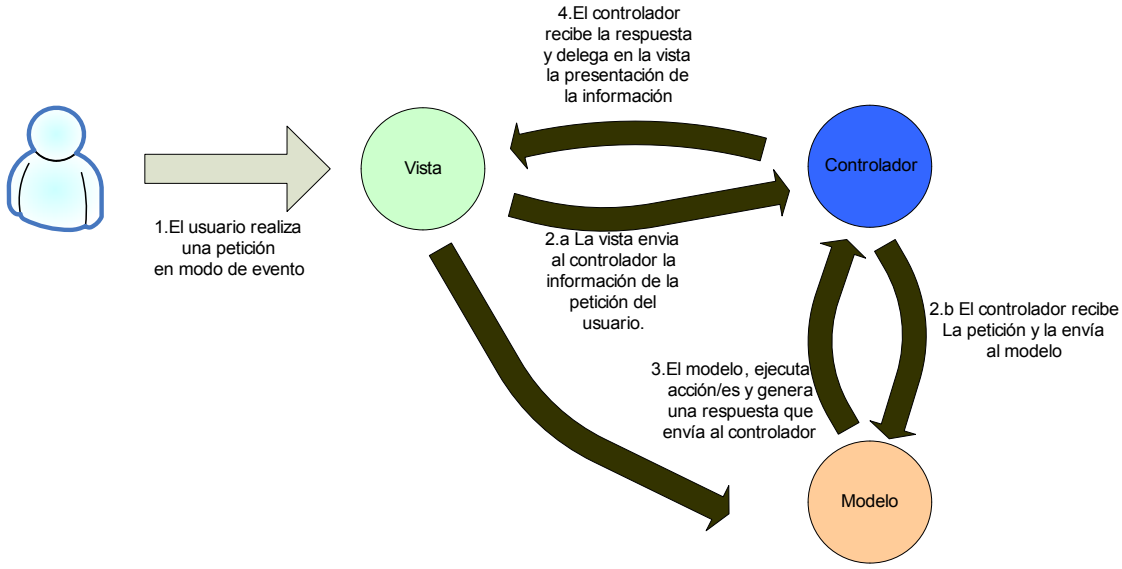
- **El modelo:** es la representación específica de la información con la cual el sistema opera, encapsula los datos y las funcionalidades (lógica de negocio). Esta capa es independiente de cualquier representación de salida y/o comportamiento de entrada. También es la encargada de organizar y ejecutar las posibles operaciones con la base de datos. Dentro de la plataforma J2EE, la capa que nos ocupa sería modelada por un conjunto de datos Java, existiendo dos claras alternativas de implementación: utilizando objetos java tradicionales llamados POJOs (*Plain Old Java Objects*) o bien utilizando EJB (*Enterprise JavaBeans*) en sistemas con unas mayores necesidades de concurrencia o distribución.

- **La vista:** es la parte gráfica de la aplicación, muestra la información al usuario de manera que éste pueda interactuar con la misma. Pueden existir múltiples vistas dependiendo de las necesidades concretas del sistema, por ejemplo una aplicación, con un mismo modelo, podría estar diseñada para un entorno tipo iPhone y un Explorador convencional; en este caso el controlador será quien decida qué vista usar. Existen diversas tecnologías que se usan para implementar una vista, entre las que podemos destacar: XSLT(*Extensible Stylesheet Language Transformations*), JSP(*JavaServer Pages*), Swing o Applets de Java.

- **El controlador:** recibe e interpreta la interacción del usuario, actuando sobre el modelo y la vista de manera adecuada para provocar cambios de estado en la representación interna de los datos, así como en su visualización. Esta capa recoge las entradas, usualmente como eventos que codifican los movimientos o pulsación de botones del ratón, pulsaciones de teclas, etc. Los eventos son traducidos a solicitudes de servicio ("**service requests**") para el modelo o la vista. Si en el sistema existen múltiples vistas, es el encargado de seleccionar cual es la indicada para la petición concreta.

3.1. Ciclo de Vida

El ciclo de vida de MVC es representado normalmente en función de las 3 capas presentadas anteriormente y el usuario del sistema. El siguiente diagrama representa el ciclo de vida de una manera sencilla:



Seguidamente se explica cada una de las etapas del ciclo de vida anterior:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.)
2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega.
3. El controlador accede al modelo, actualizándolo, posiblemente solicitando de forma adecuada la ejecución de acciones solicitadas por el usuario, por ejemplo una búsqueda de un elemento a través de un formulario.
4. Una vez procesada la información solicitada el controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). el controlador no pasa objetos de dominio (el modelo) a la vista aunque puede dar la orden a la vista para que se actualice. *Nota: En algunas implementaciones la vista no tiene acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista.*
5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

3.2. Ventajas del uso de MVC

Si bien es cierto que las ventajas o inconvenientes del uso de este patrón suelen ser algo subjetivas, generalmente esta arquitectura es usada en grandes sistemas a nivel empresarial, esto es así porque el **correcto uso del patrón MVC** aporta una gran cantidad de ventajas para el desarrollo de una aplicación.

Una de las razones principales es que simplifica mucho la tarea de mantenimiento del sistema. Si por ejemplo, una misma aplicación debe ejecutarse tanto en un navegador estándar como un navegador de un dispositivo móvil, solamente es necesario crear una vista nueva para cada dispositivo; manteniendo el controlador y el modelo original, ya que será el mismo controlador el que se encargue de separar y organizar las peticiones de los usuarios (http, WebService, etc...).

Por otro lado el uso de este patrón suele aumentar la escalabilidad del sistema desarrollado, ya que la separación en capas consigue que la implementación de nuevos requisitos funcionales pueda ser independiente en la mayoría de los casos; si por ejemplo tenemos un reproductor de música MP3 y queremos añadirle la funcionalidad de ordenar la lista de reproducción por orden alfabético, bastaría con crear un botón nuevo en la interfaz y una función en el modelo, dejando intacto el resto del código de la aplicación.

Otra ventaja es la facilidad para la realización de pruebas unitarias de los componentes del sistema; si por ejemplo queremos realizar pruebas de que nuestra aplicación ejecuta correctamente las consultas a la base de datos, bastaría con implementarlas sobre las clases del modelo que interactúan con la base de datos, aislando al resto de componentes del sistema y simplificando mucho el desarrollo de las mismas.

Por último, cabe señalar la reutilización de componentes. Supongamos que tenemos el reproductor MP3 que y nos solicitan que desarrollemos un reproductor de video tipo Windows Media Player capaz de reproducir tanto MP3 como videos; en este caso, lo normal sería reutilizar tanto el modelo como el controlador del reproductor MP3 y ampliarlo para que sea capaz de reproducir videos, con esto hemos simplificado el desarrollo del nuevo sistema enormemente.

3.3. Inconvenientes del patrón

En el caso de tener que tomar una decisión a la hora de llevar a cabo una desarrollo software con este patrón, se deberá tener en cuenta las siguientes desventajas del mismo:

1. La división en capas del sistema hace que muchas veces tengamos que crear y mantener un mayor número de ficheros que con el desarrollo tradicional.
2. A veces la estructura en capas añade complejidad al sistema, sobre todo en sistemas pequeños.
3. El aprendizaje de este modelo por parte de los desarrolladores suele ser mas costoso que usando otros modelos mas sencillos.

4. MVC && Java.swing

La mayoría de componentes Swing han sido diseñados usando el patrón MVC. Esto supone una gran ventaja al poderse variar el modelo de datos para cada componente swing.

De una forma general, podría entenderse que el la Vista haría referencia al aspecto gráfico del componente, el Modelo al conjunto de datos que maneja el componente y el Controlador, la gestión de eventos que relaciona la Vista con el Modelo.

Para todo componente, Swing propone un modelo general que conviene adaptar para las necesidades concretas que tengamos en cada momento. Los Modelos proporcionados por Swing usan la clase java.util.Vector para el almacenamiento de datos, lo que podría provocar un uso desmesurado de recursos. Igualmente, la adaptación de este Modelo nos ofrece la posibilidad de trabajar con componentes más eficientes.

Bibliografía

Len Bass, Paul Clements, Rick Kazman. Software Architecture in Practice. ISBN-10: 0321154959, 2003

David Garlan and Mary Shaw. An Introduction to Software Architecture. Advances in Software Engineering and Knowledge Engineering, Volume I, 1993.

Ingeniería del Software (Edición 7 traducida). Ian Somerville. Editorial Pearson, Addison Wesley.

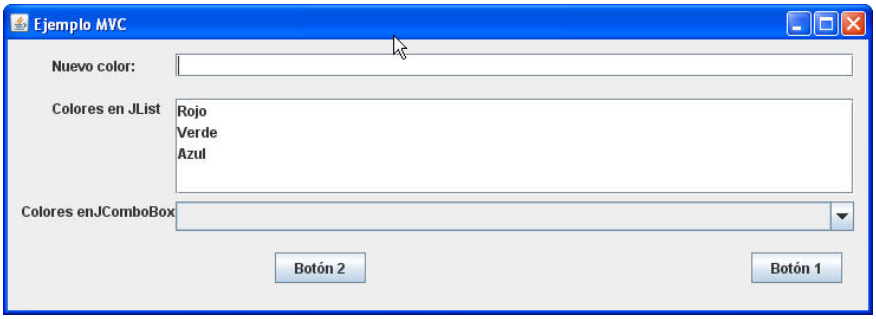
Ingeniería del Software. Un enfoque práctico (Edicion 6 Traducido). Roger S. Pressman. Editorial Mac Graw Hill. ISBN 970-10-573-3

Head First. Design Patterns. Eric Freeman & Elisabeth Freeman. Editorial O´REILLY. ISBN 0-596-00712-4

Experimentos

Usar la estructura MVC usada en la implementación javax.swing para implementar una pequeña aplicación.

E1 (15 mins). En primer lugar implemente una interfaz gráfica que tenga el aspecto mostrado en la figura siguiente



E2 (10 mins). Añada una funcionalidad a cada uno de los botones. Cuando el usuario pulse en el "Botón 1" inserte el valor escrito en el campo "Nuevo color" en el JComboBox. Para ello, no haga uso directamente del modelo por defecto que tiene asociado tal componente.

E3 (15 mins). Realice la operativa anterior pero al pulsar el Botón 2 y añadiéndole al JList. ¿Sería posible realizarlo sin usar el modelo directamente?

E4 (45 mins). Genere un modelo propio para que pueda ser usado en JList y JComboBox. Este modelo deberá ser capaz de albergar colores, que son definidos por String.

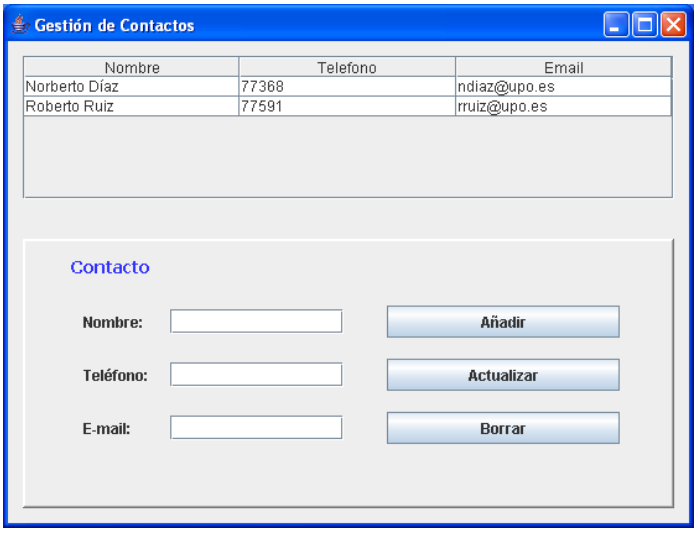
E5 (20 mins). Modifique el modelo por defecto usado por la instancia de Jlist y JComboBox por el creado anteriormente, de manera que ambos usen el mismo. ¿Es necesario que el modelo usado sea Singleton? Haga los cambios apropiados.

E6 (15 mins). Asocie la acción de pulsar el botón del ratón sobre el "Botón 2" a la misma acción del "Botón 1".

Problema

Se tiene una aplicación que gestiona los contactos que se encuentran en una agenda. Concretamente, da servicio tanto al alta como a la actualización, borrado o consulta de los datos que se encuentran en una agenda.

- Una interfaz gráfica con el siguiente aspecto:



- El negocio de la aplicación ha sido desarrollado mediante la implementación de la clase **ContactoImpl**, que implementa la interfaz **Contacto** con los siguientes métodos:

```
String getEmail();

String getNombre();

String getTelefono();

void setEmail(String email);

void setNombre(String nombre);

void setTelefono(String tlf);
```

- Nótese que en la versión actual no existe capa de persistencia, sino que se debería hacer de manera volátil.

Se pide:

a) Actualice la implementación de la capa de Modelo de Contacto usando la memoria del sistema. Nótese que debería hacer uso del patrón DAO y para ello debería definir la interfaz ContactoDAO con, al menos, las operaciones CRUD y la de listar.

b) Implemente la interfaz gráfica a partir del proyecto facilitado en WebCT.

c) Modifique el aspecto visual de la capa gráfica pero manteniendo la misma funcionalidad. Como ejemplo, se podría usar un JList para mostrar el nombre del contacto y que su selección mostrara su información en el JPanelContacto creado previamente.

Datos de la Práctica

Autor del documento: Norberto Díaz Díaz (Abril 2012).

Revisiones:

- Norberto Díaz Díaz (Mayo 2013): Revisión de formato, estimación temporal y modificación de ejercicios.

Estimación temporal:

- Parte presencial: 120 minutos.
    - Experimentos: 120 minutos.
- Parte no presencial: 270 minutos.
    - Lectura y estudio del guión y Bibliografía: 90 minutos
    - Problema: 180 minutos