Programación orientada a eventos con AS 3.0

Web

El manejo de eventos puede significar la comunicación entre clases, capas y entidades como el usuario y el sistema; el sistema consigo mismo y el sistema con otros sistemas. El manejo de eventos es algo tan particular e importante que existe el paradigma de Programación Orientada a Eventos.

En Programación Orientada a eventos, en la mayoría de los casos, las clases no se comunican directamente. Esto da por resultado, módulos y entidades completamente independientes. Una de las maneras de lograr el bajo acoplamiento es justamente comunicar las clases por eventos. Esto implica ir un paso más allá en la forma en la que planteamos el desarrollo de un sistema o aplicación, y se transforma incluso en una metodología de trabajo.

Esta metodología exige agregar una dimensión más a la Programación orientada a objetos e implica otra manera de pensar, desarrollar, probar y mantener nuestro sistema. Este concepto se basa en lo que se denomina llamados asincrónicos: se
efectúa un llamado y se puede recibir o no respuesta en otro momento de manera diferida.

En algunos casos, podremos aplicar por completo los conceptos de Programación Orientada a Eventos, y en el caso particular de videojuegos o juegos casuales, en circunstancias concretas, no se podrá aplicar en forma completa este paradigma para lograr una mayor *performance*. Por supuesto, que antes de tomar este tipo de decisiones se debe analizar y estudiar la situación en profundidad para determinar con precisión hasta qué punto su utilización puede mermar o no el desempeño general.

Las características de este paradigma de programación son:

- Llamados asincrónicos.

- El código no se verá concentrado en un solo método o método que se ejecuta secuencialmente.
- Los llamados a los métodos que manejan los eventos, son efectuados por otras entidades que no necesitan conocen al objeto.
- Tendremos un nivel de indirección más al comunicar las clases.
- Obtendremos un bajo acoplamiento entre clases.
- La aplicación se tornará más flexible y fácil de extender.

Patrón Observer y ActionScript 3.0

Uno de los temas que despiertan más interés, hasta devoción, son los patrones de diseño. Los patrones de diseño fueron definidos en un mítico libro llamado: *Design Patterns: Elements of Reusable Object-Oriented Software*. Sus autores Gamma, Helm, Johnson y Vlissides. Son conocidos con el afectuoso mote de: *Gang of four* (La banda de los cuatro).

El propósito de este patrón de diseño, según escribieron estos autores: "Define la dependencia entre objetos, de uno a muchos, de tal forma que cuando uno cambia su estado, todos sus objetos dependientes son notificados y actualizados automáticamente".

Existen algunas diferencias entre la forma en que se implementa el patrón *Observer* y como se lo implementa en ActionScript 3.0. Recordemos que los diagramas e implementaciones que se describen en el libro *Gang of four*, no debe ser tomado al pie de la letra, sino solamente como orientación y deben ser adaptados a cada aplicación, según las necesidades de diseño. Viendo el esquema UML (*Unified Modeling*
Language, Lenguaje unificado de modelado).

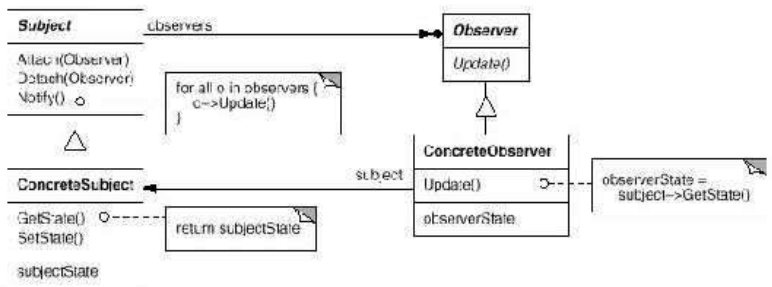

Fig. 1 Diagrama del patrón Observer.

Podemos hasta cierto punto hacer un paralelo con lo que hemos visto:

Subject -> Target

Observer -> eventListener: aquí tenemos la primera diferencia, el "observador" o escucha en el ejemplo es un objeto; en ActionScript 3.0, es un método.

Veamos los métodos de cada uno y sus paralelos:

```
Subject
  Attach (Observer) -> addEventListener(type:String, event-
Listener:Function, useCapture:Boolean)

  Detach(Observer) -> removeEventListener(type:String,
eventListener:Function, useCapture:Boolean)

  Notify() -> dipatchEvent(event:Event)

Observer
  Update() -> eventHandler(event:Event) :
```

Aquí tenemos otra diferencia. Hay un único punto de entrada para la notificación, mientras que en ActionScript 3.0, cualquier método que acepte como parámetro un objeto del tipo `Event` puede funcionar como escucha.

El ejemplo que nos presenta *Gang of four* se basa en notificar a los `Observer` en cuanto el `Subject` cambia su estado. El `Subject` tiene dos métodos: uno, para cambiar el estado; y otro, para tomarlo al modo *getter* y *setter* (que ya hemos visto). El método `SetState` provocará que se dispare el método `Notify`.

Como siempre se remarca, no es buena práctica aplicar los patrones a raja tabla independientemente de la necesidad que se tenga. Tampoco, pensar en patrones antes que entender el problema y su solución. A la mala utilización del patrón se le llama antipatrón.

Patrón Mediator y ActionScript 3.0

Si bien el patrón que se relaciona con la Programación Orientada a Eventos es el `Observer`, existe otro patrón muy utilizado y potente, pero que no tiene la misma popularidad en los desarrollos que el `Observer`. Esto se debe a la tendencia a establecer relaciones de alto acoplamiento entre clases, ya sea llamando a métodos

de un objeto a otro directamente o bien pasando referencia de objetos dentro de la aplicación.

Esta situación con el tiempo lleva a un gran laberinto de llamados, y a lo que comúnmente se llama *spaghetti code*. Muchas veces a la metodología de tipear código directamente sin tomar el recaudo de realizar una fase de diseño, un modelado de clase o como mínimo algún esquema, se dice que se codea a lo *cowboy*, esto seguramente se debe a la relación con el *spaghetti western*. Desarrollar con una preparación y arquitectura previa podríamos denominarlo: codear a lo *Jedi*.

El propósito del patrón `Mediator`, según *Gang of four* es: "Encapsular en un objeto cómo interactúan un conjunto de objetos. `Mediador` promueve un bajo acoplamiento, evitando que los objetos se referencien entre sí explícitamente, y permite que varíen independientemente".

Si bien `Mediator` tiene cierta similitud con el `Facade` (fachada), difiere fundamentalmente en que mientras el `Facade` simplifica y abstrae el funcionamiento de otras clases existentes, el `Mediador` abstrae la comunicación entre las clases y, además, agrega funcionalidad en la forma en que se comunican entre sí. Veamos el UML del `Mediator`.

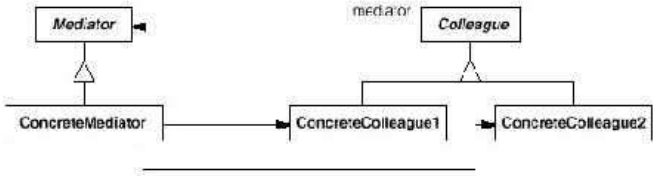

Fig. 2

Si analizamos el diagrama UML, veremos que tiene una gran similitud con el patrón `Observer`. La gran diferencia es el propósito de este patrón, que además de implementar un mecanismo de aviso a los "interesados" que en el libro *Gang of four* se denomina "colegas", también implementa distintos tipos de lógicas para determinar qué y a quién notificar.

El ejemplo más claro para entender el patrón `Mediator` es el de la torre de control y los aviones. Los aviones para despegar y aterrizar no se hablan entre sí, sino que se comunican únicamente con la torre de control. Y la torre de control es la que
orquesta las comunicaciones entre los aviones, y cada uno de los ellos solo necesitan conocer a la torre de control.

Existen distintas maneras de implementar un `Mediator`, y orquestar las comunicaciones entre los objetos. En algunos casos, se pasa como referencia a la clase
`Mediator` a cada uno de los objetos (a los que se llama `Colega`. Otras veces,

cada uno de los `Colega` pasa una referencia al `Mediator`, y éste de acuerdo con el objeto del que se trate encausa las comunicaciones.

En cualquiera de dichos dos casos existiría un acoplamiento entre clases, pero el inconveniente mayor que encontramos es que al pasar como referencia un objeto, no sabemos como lo manejará la clase que lo recibe, y esta práctica hace propenso al nacimiento de los consabidos *zombies*. Esto exige que al implementarlo seamos muy cautos y cuidadosos en el uso de estas referencias.

La práctica más segura es la que utiliza una combinación con el patrón `Observer`.

El `Mediator`, comunica al resto de los `Colega` el cambio a través de eventos. Lo cual permite:

- A los `Colega` mantener desacoplados los objetos entre sí.
- El `Mediator` no necesita conocer a los objetos que está vinculando.
- Evitar la posibilidad de generar *zombies*.

Debemos tener en cuenta que si bien este patrón es totalmente conveniente para encapsular comunicaciones complejas, manteniendo desacoplados los objetos intervinientes, esta clase puede tornarse monolítica y difícil de mantener.
El utilizar el `Mediator` en combinación con el patrón `Observer` permite que a través de eventos, cada uno de los objetos reaccione, en consecuencia, reduciendo la cantidad de lógica que debe tener el `Mediator`.

Esto nos lleva al siguiente patrón de patrones el: `Event Aggregator`.

Event Aggregator

Una aplicación que resulta muy difícil de mantener es cuando tenemos muchos objetos que deben comunicarse entre sí, y más cuando algunos objetos cumplen la función de despachador y de suscriptores a la vez, detectando un evento y redistribuyéndolo al mismo tiempo,. Esto sucede, porque obliga a los objetos a conocerse entre sí, llevando a aplicaciones con flujos muy complicados de seguir, y eventualmente de corregir.

El `Event Aggregator` funciona como un único despachador de eventos al cual se suscriben el resto de los objetos. De esta manera, los objetos solo deben conocer al `Event Aggregator` para suscribirse. A su vez, éste conoce a todos los demás objetos y se subscribe a sus eventos. Así, el `Event Aggregator` recibe y distribuye los eventos, en otras palabras, tiene escuchas para los eventos de los objetos y también es escuchado. Esto simplifica enormemente los mecanismos de suscripción, despacho y el flujo de comunicación entre objetos.

Veamos un esquema que es lo que simplifica el `Event Aggregator`.

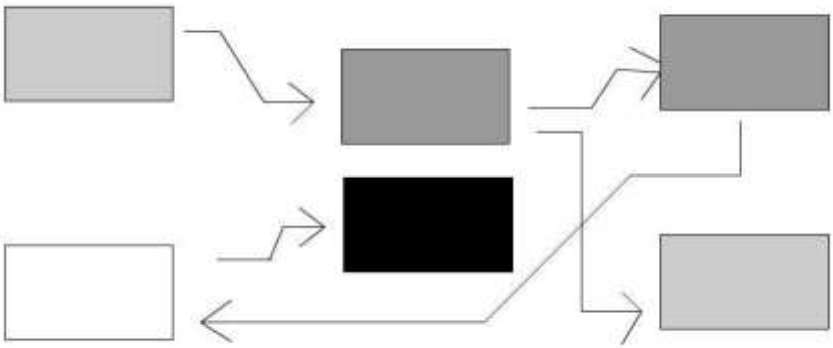

Fig. 3 Esquema de relaciones sin el Event Aggregator.

El `Event Aggregator` podría manejar un único tipo de evento.

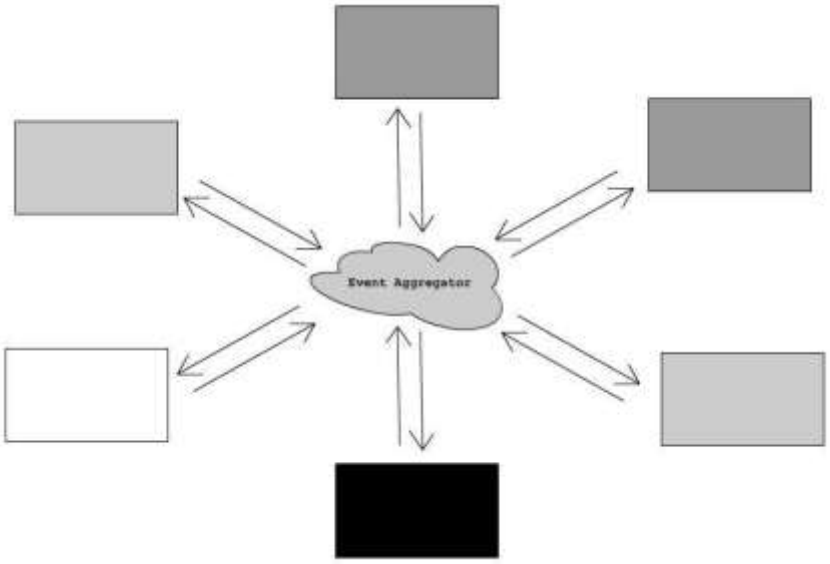

Fig. 4 Esquema de relaciones utilizando el Event Aggregator.

Se trata de abstraer y simplificar este conjunto de eventos con los cuales se intercomunicaran las clases, encapsulándolos en un paquete, utilizando distintos

tipos de eventos, y haciendo que el `Event Aggregator` tome la información y los "convierta" en los eventos que esperan los objetos de nuestra aplicación. Este comportamiento está asociado a los patrones: `Facade` y `Adapter`. Esta práctica además mantiene bajo control el manejo de eventos de la aplicación así como facilita el uso y control de la memoria.

Implementando el Event Aggregator

La cuestión fundamental por la cual existen las buenas prácticas, los distintos paradigmas de programación y los patrones de diseño es para manejar el CAMBIO, ya que es lo único que no varía en cualquier proyecto de software. Por eso, existen todas estas técnicas para lograr aplicaciones de calidad. Y entendemos por calidad a las aplicaciones que cumplan con lo que espera el cliente, sean flexible al cambio y ajuste, y además fáciles de extender y escalar. Haremos un ejercicio para aplicar las técnicas que estamos viendo. Por supuesto, que la aplicación que estamos planteando puede ser resuelta utilizando una sola clase. Y por lo que representa la aplicación puede resultar exagerada la cantidad de clases que vamos a utilizar. Y es que la intención es mostrar a través de un ejemplo sencillo cómo pensamos e implementamos distintas técnicas de desarrollo, si es que tenemos la inquietud de ver un ejemplo completo aplicado. De lo contrario, podemos pasar directamente a la próxima sección.

Consideraciones generales de la aplicación

Supongamos que recibimos un requerimiento como el que sigue: una aplicación para desplegar contactos y recorrerlos. De momento, no tendríamos la opción de editar, agregar o remover.

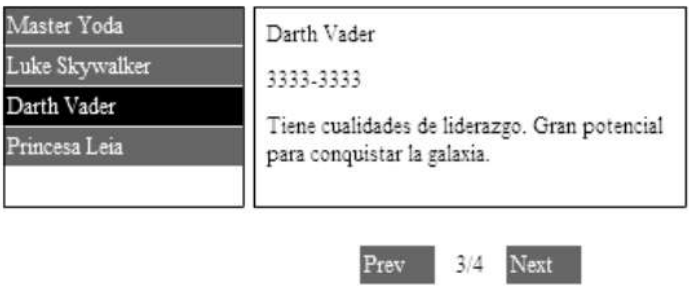

Fig. 5

La forma de abordar cualquier proyecto es: categorizar y reconocer entidades, realizar abstracciones, y luego representar esta metáfora en clases. Para el caso que se nos plantea, podemos dejar de lado la metáfora, ya que gran parte de las clases y entidades se encuentran implícitos en el bosquejo. Y tampoco hace falta comenzar desde cero. Sabemos a grandes rasgos algunas estrategias que emplearemos y que nos ayudarán a tomar decisiones: cómo distribuir el código en nuestra aplicación, y cómo haremos las implementaciones.

Cómo funcionará nuestra aplicación

Para plasmar como funcionará nuestra aplicación listaremos los siguientes *user stories* (historias de usuario) que nos guiarán en el desarrollo de nuestra aplicación.

- La aplicación cargará los datos de los contactos de una fuente determinada.
- Al cargarse los datos:
  - El sistema desplegará en el "Panel lista de contactos", botones con el nombre y apellido de contacto.
  - Se seleccionará el primer contacto en forma automática.
- Se selecciona el contacto desde "Panel lista de Contactos": al hacer clic sobre algunos de ellos, se seleccionará el contacto respectivo.
- Se selecciona el contacto desde "Panel de navegación": al hacer clic sobre algunos de los botones: "anterior" o "siguiente" seleccionará el contacto respectivo.
- Al seleccionarse un contacto:
  - En el "Panel lista de contactos", se marcará el contacto seleccionado.
  - En el "Panel de Contacto", se mostrará el nombre y apellido, número de teléfono y comentario correspondiente al contacto seleccionado.
  - En el "Panel de navegación", estando en el primer contacto seleccionado, se deshabilitará el botón con el texto "anterior".
  - En el "Panel de navegación", estando el último contacto seleccionado, se deshabilitará el botón con el texto "siguiente".
  - En el "Panel de navegación", se mostrará el índice actual y la cantidad total de contactos.

Relaciones entre los paneles

Las relaciones que existen entre los paneles son las que se muestran en la siguiente figura:

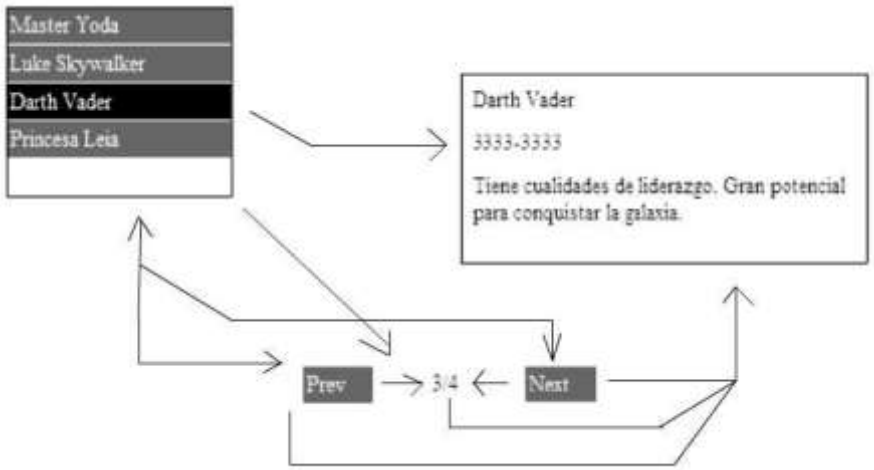

Fig. 6

Si bien se trata de un ejemplo simple podemos ver como las interacciones pueden revestir cierto grado de complejidad, pero no al momento de hacerlo, sino de cuando se efectúan los agregados o cambios. Así, es como en pequeñas brechas se filtran y acumulan *bug*s, que en la medida en la que se empieza a avanzar y a lidiar con los cambios de requerimiento, lleva a que la aplicación se torne difícil de mantener, extender, y también de hacer arreglos.

El fenómeno que sucede luego en grandes aplicaciones como un juego de consola triple A es que un cambio muy pequeño que no reviste ninguna complejidad se torna una odisea solucionarlo, por el tipo de arquitectura. Cuando se logra dar con la fracción de código para hacer el arreglo y se lo efectúa, pueden propagarse errores en otras secciones de la aplicación.

El hecho de trabajar en capas permite mantener a raya a los errores. Vamos estableciendo barricadas que bloquean la propagación de errores. Otra cuestión a tener en cuenta son los grandes y pequeños proyectos previos a la entrega o en la etapa de *bugfixing* (arreglo de errores). Los equipos se encuentran con una presión psicológica mayor, muchas veces se encuentran con acumulación de cansancio. En un escenario de este tipo el tener modularizada la aplicación permite al desarrolla-

dor concentrarse en puntos específicos de la aplicación y modificarlos con alto grado de seguridad, debido a que es limitado hasta donde puede propagarse un error.

Como hemos dicho utilizaremos el patrón de patrones MVC (Modelo, Vista, Controlador). Repasaremos de qué se trata.

MVC: Modelo – Vista - Controlador

Aplicaremos la estructura de MVC (*Model, View Controller*, Modelo, Vista Controlador). Esta arquitectura se basa en separar en tres capas nuestra aplicación:

- **Modelo**: todo lo referido a datos de la aplicación.
- **Vista**: esta capa se ocupa de la representación visual de algunos o todos los datos del Modelo.
- **Controlador**: es el encargado de traducir al sistema las acciones del usuario que pueden resultar en una actualización del Modelo, lo cual desencadena a su vez que la Vista se actualice para mantenerse consistente con los datos. El Controlador determina cómo el usuario interactuará con el sistema. Traduce las entradas de teclas, botones, o también puede ser que se encargue de procesar lo que capture con una cámara Web para realidad aumentada, un sonido que se detecta con el micrófono, movimientos con acelerómetro, etcétera.

Esta arquitectura dota de flexibilidad a aplicaciones con representación visual y permite que varíen independientemente el manejo de datos, su representación e interacción.

Hay muchas maneras como estas tres capas pueden implementarse e interrelacionar. Un mismo conjunto de datos puede tener distintas formas de ser representado y distintas Vista. Podría suceder que tengamos un juego que se realiza con teclado y mouse. Si se quisiera hacer el mismo juego, pero que se juegue por reconocimiento de imágenes, solamente se debería cambiar la capa relacionada con el Controlador sin alterar las otras capas.

Con el MVC abstraemos en capas responsabilidades similares, con el `Event Aggregator` abstraemos la comunicación entre las capas.

Existen muchas maneras de interrelacionar estas tres entidades. Una manera muy usual, y además conveniente de implementar el MVC, es entendiéndolo como tres capas: una para el `Model`, otra para el `View`, y otra para el `Controller`. Cada una de estas capas puede tener una clase que funciona como el patrón Me-

diador, abstrayendo la comunicación con los subcomponentes de cada una de las capas. El ejemplo más representativo es el de la Vista, el cual puede tener distintos paneles visuales. Podemos tener una clase que se encargue de manejar y orquestar todos estos paneles. Entonces, la aplicación no necesitaría conectarse con cada uno de los paneles, sino únicamente con una clase que maneje estos paneles. La forma de implementarlo es aplicar el patrón `Composite` (Composición), en donde una clase puede contener a otras. O bien utilizar clases `Mediator`, que se comunican entre sí aplicando la técnica del `Event Aggregator`.

Lo primero que se destaca es que el `Controller` trabajará como un `EventAggregator`, suscribiéndose a los eventos del `Model` y el `View`. Tanto el `View` como el `Model` actuarán como fachadas en la intercomunicación de sus respectivas clases.

Vemos que las relaciones entre los ítems, utilizando la técnica del `Event Aggregator`, se ven ampliamente simplificadas. Esto no significa que la complejidad desaparezca, pero sí que quedará centralizada, encapsulada, y en algunos casos hasta oculta. Como desventaja, el `Event Aggregator` puede llegar a volverse complejo.

Veamos como se ve la figura de relaciones aplicando las técnicas del MVC y el `EventAggregator`.

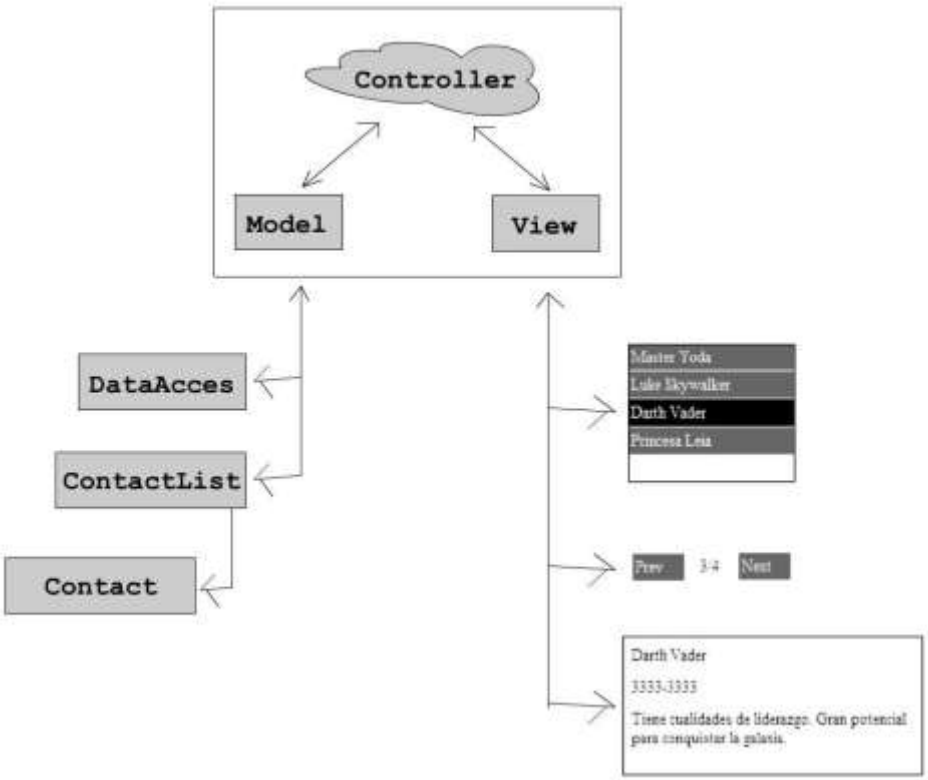

Fig. 7

A grandes rasgos tendremos lo siguiente:

`Model`: manejará la obtención y creación de la lista de contactos.
`View`: manejará la creación y despliegue de los objetos visuales que representarán los visualmente los datos.
`Controller`: manejará la lógica de interacción entre el `View`, el `Model` y el usuario. Contendrá la lógica necesaria para mantener consistentes al `Model` con el `View`.

Premisas de nuestro desarrollo

Vamos a listar los principios con los cuales nos manejaremos para implementar este ejemplo:

- Separaremos la aplicación en tres capas: Modelo, Vista y Controlador.
- Cada una de estas capas tendrán a su vez, las subdivisiones que sean necesarias.
- Las capas podrán funcionar independientes de las demás.
- Internamente, las capas podrán comunicarse por eventos o llamados directos.
- Mantendremos las interfaces de las clases con la menor cantidad posible de métodos. El reducir la cantidad de conectores y formas de acceder a la clase representará mayor simpleza de la aplicación en general.
- Intercomunicaremos a las capas por eventos.
- La clase del documento será la responsable de manejar a las tres capas. Esto implica: crearlas, inicializarlas si fuera necesario, interconectarlas y destruirlas.
- Cada capa podrá funcionar independiente de las otras.
- Priorizaremos la modularidad.

Veamos las entidades que entrarán en juego para realizar la aplicación de nuestro ejemplo:

Tendremos tres clases: `Model`, `View` y `Controller`, que actuarán como los respectivos `Facades` de estas tres capas. Y además, cumplirán un rol muy cercano al `Abstract Factory` (Factoría abstracta) ya que en el caso de `Model` y `View`, crearán un conjunto de instancias determinadas.

En la raíz de `src` se encontrará la clase del documento. Esa clase se encargará de manejar las tres capas, y de ser necesario, inicializarlas y destruirlas.

Ahora, lo que haremos es identificar las clases principales en cada capa con su responsabilidad. Veamos capa por capa.

Documento de la clase

La clase que será el punto de entrada de nuestra aplicación estará en la raíz de la carpeta `src` y se llamará `Main`. Durante el desarrollo utilizaremos otra clase como la principal para ir probando cada una de las clases que vayamos incorporando, su nombre será `MainTest`. Eventualmente, podemos hacer distintas clases para luego realizar pruebas unitarias de nuestra aplicación, pero eso está fuera del alcance de este ejercicio.

Por lo tanto, comenzaremos por hacer la clase `MainTest`, como sigue a continuación:

```
package
{
  /**
   * Punto de entrada de la aplicación.
   * Su responsabilidad es manejar los objetos del tipo:
   * Model, View y Controller.
   * Esto implica, crearlos, eventualmente inicializarlos y destruirlos.
   */
  import flash.display.Sprite;

  public class MainTest extends Sprite
  {
    /**
     * Constructor de la clase
     */
    public function MainTest()
    {
      initialize();
    }

    /**
     * Inicializa la clase.
     */
    private function initialize():void
    {
      trace("MainText ::> initialize");
    }
  }
}
```

Configurando esta `MainTest` como clase del documento, al probar deberíamos ver en la ventana de salida:

```
MainText ::> initialize
```

Implementando la capa del Modelo

Todas las clases referidas a esta clase irán en el paquete:

com.alfaomegaeditor.model

Las clases que implementaremos son:

- **Model:** es el `Facade` de la capa correspondiente al Modelo. Crea y destruye los objetos. Es el punto de comunicación entre la capa del Modelo y otras entidades.
- **ContactList:** maneja una lista de contactos del tipo `Contact`. Esto incluye crear y destruir la lista y permitir manejar el índice actual.
- **Contact:** la responsabilidad de esta clase es contener los datos de un contacto.
- **DataAccess:** accede y contiene los datos que se hayan cargado de una fuente determinada, puede ser un servicio Web, un XML o los datos que se encuentren en alguna otra clase.

La clase Model

Comenzaremos por el punto de conexión con esta capa. Se trata de la clase `Model`. Como esta clase tiene la función de manejar a las clases del modelo a través de eventos, deberemos darle el conocimiento para agregar y remover escuchas. Por lo tanto, haremos que extienda el `EventDispatcher` de ActionScript, y la clase `Model` quedaría de la siguiente manera:

```
package com.alfaomegaeditor.model
{
  /**
   * La responsabilidad de esta clase es: ser el Facade de la capa: Modelo.
   * Crea y destruye los objetos. Es el punto de comunicación entre esta capa
```

```
   * y otras entidades.
   */

  import flash.events.EventDispatcher;

  public class Model extends EventDispatcher
  {
    /**
     * Constructor de la clase
     */
    public function Model()
    {
      initialize();
    }

    /**
     * Inicializa el objeto.
     */
    private function initialize():void
    {
      trace("Model ::> initialize");
    }
  }
}
```

Por lo cual, desde `MainTest` importaremos a `Model` y lo instanciaremos en la inicialización de `MainTest`. Para no sobrecargar el método `inicitialize`, repartiremos entre distintos métodos los pasos de inicialización de cada clase.

```
package
{
  /**
   * Punto de entrada de la aplicación.
   * Su responsabilidad es manejar los objetos del tipo:
```

```
   * Model, View y Controller.
   * Esto implica, crearlos, eventualmente inicializarlos y des-
truirlos.
   */

  import com.alfaomegaeditor.model.Model;

  import flash.display.Sprite;

  public class MainTest extends Sprite
  {
    private var _model:Model;

    /**
     * Constructor de la clase
     */
    public function MainTest()
    {
      initialize();
    }

    /**
     * Inicializa la clase.
     */
    private function initialize():void
    {
      trace("MainText ::> initialize");
      createObjects();
    }

    /**
     * Crea todos los objetos necesarios.
     */
    private function createObjects():void
    {
      _model = new Model();
    }
```

```
    }
}
```

Por supuesto que para ser consistentes con el nombre del paquete, tendremos la siguiente estructura de carpetas:

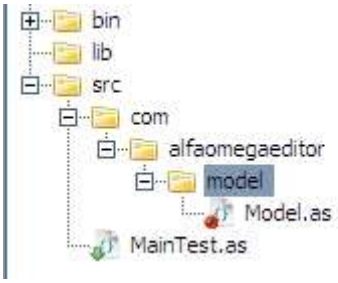

Fig. 8

Probamos y tendremos la salida:

```
MainText ::> initialize
Model ::> initialize
```

Implementando las clases de Model

La clase `Model` será el centro neurálgico de la capa `Model`. Como vimos en el diagrama hay otras dos clases que debemos crear: `Contact`, que representa los datos de cada contacto; y `ContactList`, que será la clase encargada de manejar justamente la lista de contactos. Por lo tanto, ahora nos focalizaremos en estas dos clases.

Primeros, abordaremos `Contact`, para que cuando estemos con `ContactList`, tengamos una clase real sobre la cual trabajar.

Contact

Esta clase tendrá la responsabilidad de tener los datos de cada contacto. Sus propiedades serán: `name` (nombre), `lastName` (apellido), `phoneNumber` (número

de teléfono), `commentary` (comentario). Es una clase que no tiene métodos. Solo tendrá propiedades privadas con sus correspondientes *setters* y *getters*.

```
package com.alfaomegaeditor.model
{
  /**
   * La responsabilidad de esta clase es contener los datos de un
contacto.
   */

  public class Contact
  {
    private var _name:String;
    private var _lastName:String;
    private var _phoneNumber:String;
    private var _commentary:String;

    public function Contact(name:String="",
                            lastName:String="",
                            phoneNumber:String="",
                            commentary:String="")
    {
      this.name = name;
      this.lastName = lastName;
      this.phoneNumber = phoneNumber;
      this.commentary = commentary;
    }

    public function get name():String
    {
      return _name;
    }

    public function set name(value:String):void
    {
      _name = value;
```

```
    }

    public function get lastName():String
    {
      return _lastName;
    }

    public function set lastName(value:String):void
    {
      _lastName = value;
    }

    public function get phoneNumber():String
    {
      return _phoneNumber;
    }

    public function set phoneNumber(value:String):void
    {
        _phoneNumber = value;
    }

    public function get commentary():String
    {
      return _commentary;
    }

    public function set commentary(value:String):void
    {
        _commentary = value;
    }

    public function toString():String
    {
      var newLine:String = "\n";
      var tab:String = "\t";
      var dataString:String = "";
```

```
      dataString += tab + "name: " + name + newLine;
      dataString += tab + "lastName: " + lastName + newLine;
      dataString += tab + "phoneNumber: " + phoneNumber + newLine;
      dataString += tab + "commentary: " + commentary + newLine;
      dataString += tab + "-----------" + newLine;
      return dataString;
    }
  }
}
```

ContactList

Esta clase comenzó primero con el pseudocódigo, y luego se le agregó el código correspondiente. El pseudocódigo pasó a convertirse en los comentarios y documentación de la clase. Tendrá una responsabilidad muy parecida al patrón `Iterator` (iterador). Se encargará de tener un vector de `Contact` y de manejar el índice de la lista de contactos.

En negritas veremos lo que sería la interfaz de la clase, es decir, los métodos públicos.

El procedimiento es hacer cambios pequeños cada vez y probar que todo siga funcionando bien. Nosotros nos saltearemos esos pasos, pero queda como ejercicio para el lector, teniendo en mente que en teoría los hemos hecho.

```
package com.alfaomegaeditor.model
{
  /**
   * La responsabilidad de esta clase es manejar una lista de con-
tactos.
   * Esto incluye crear y destruir el contactList y permitir mane-
jar el
   * índice actual.
   */

  import flash.events.EventDispatcher;

  public class ContactList extends EventDispatcher
```

```
  {
    private var _contactList:Vector.<Contact>;
    private var _currentIndex:uint;
    private var _indexHasChanged:Boolean;

    /**
     * Constructor de la clase.
     */
    public function ContactList()
    {
      //Inicializamos el objeto.
      initialize();
    }

    /**
     * Dado un índice, devuelve el Contact correspondiente.
     * @param index: índice del Contact a obtener. Internamente el
índice es
     * transformado en un índice válido.
     * @return: devuelve el Contact correspondiente.
     */
    public function getContactAt(index:uint):Contact
    {
      //Convertimos el index a un dato válido.
      var index:uint = convertToValidIndex(index);
      //Obtenemos el contacto.
      var contact:Contact = _contactList[index];
      //Lo devolvemos
      return contact;
    }

    public function nextIndex():uint
    {
      //Sumamos uno a currentIndex (ver el setter de currentIndex).
      currentIndex ++;
      //Devolvemos el currentIndex.
      return currentIndex;
```

```
    }

    public function prevIndex():uint
    {
      //Si _currentIndex es 0
      //  Devolvemos el _currentIndex y detenemos el método.
      if (_currentIndex == 0)
      {
        return _currentIndex;
      }

      //De lo contrario, restamos uno.
      currentIndex --;
      //Y lo devolvemos.
      return currentIndex;
    }

    /**
     * Agrega Contact desde un array de objetos.
     * @param arrayData: array de objetos con las propiedades de
Contact.
     * @return: la cantidad total de Contact, luego de ser inserta-
dos los nuevos
     * contactos.
     */
    public function addContactsFromArray(arrayData:Array):uint
    {
      var property:String;
      var dataObject:Object;
      _contactList.length = 0;
      var index:uint = 0;
      var arrayDataLenght:uint = arrayData.length;
      //Recorremos el array creando por cada object el correspon-
diente Contact.
      while (index < arrayDataLenght)
      {
        //Obtenemos el object.
        dataObject = arrayData[index];
```

```
        //Hacemos un nuevo Contact.
        var contact:Contact = new Contact();
        for (property in dataObject)
        {
          //Volcamos cada propiedad del objeto.
          contact[property] = dataObject[property];
        }
        //agregamos el contacto.
        _contactList.push(contact);
        index++
      }

      //Devolvemos la cantidad total de contactos.
      return length;
    }

    public function set length(length:uint):void
    {
      _contactList.length = length;
    }

    public function get length():uint
    {
      return _contactList.length;
    }

    public function get currentIndex():uint
    {
      return _currentIndex;
    }

    public function set currentIndex(value:uint):void
    {
      //Si la cantidad de contactos es 0, detenemos el método.
      if (0 == length)
      {
        return;
```

```
    }

    var index:uint = convertToValidIndex(value);

    //Guardamos en la variable si hubo un cambio de índice.
    _indexHasChanged = index != _currentIndex;

    //Asignamos el nuevo valor del índice.
    _currentIndex = index;
  }

  public function get indexHasChanged():Boolean
  {
    return _indexHasChanged;
  }

  /**
   * Devuelve una cadena con los pares: propiedad/valor.
   * @return: cadena con los pares: propiedad/valor.
   */
  override public function toString():String
  {
    var dataString:String = "";
    var newLine:String = "\n";
    var tab:String = "\t";
    var index:uint;
    var contactListLength:uint = _contactList.length;
    while (index < contactListLength)
    {
      dataString += tab + index + ")" + newLine;
      dataString += _contactList[index].toString();
      index++
    }
    return dataString;
  }
```

```
    /**
     * Destruye todos los objetos complejos.
     */
    public function destroyObjects():void
    {

      _contactList = null
    }

    /**
     * Inicializa el objeto.
     */
    private function initialize():void
    {
      //Creamos el _contactList
      _contactList = new Vector.<Contact>;
    }

    /**
     * Dado un índice lo convierte en un índice válido.
     * @param value: el índice que se desea convertir.
     * @return: devuelve el índice válido.
     */
    private function convertToValidIndex(value:uint):uint
    {
      var index:uint;
      //Verificamos que no esté fuera de la cantidad total de con-
tactos
      var isOutOfBound:Boolean = value >= length;

      if (isOutOfBound)
      {
        //Si hay contactos devolvemos el mayor índice, de lo con-
trario 0.
        index = length > 0 ? length - 1 : 0;
      }
      else
```

```
      {
        //Si llegamos hasta aquí es porque es un valor válido.
        index = value;
      }

      //Devolvemos el index.
      return index;
    }
  }
}
```

Para tener un mapa de la situación en el proyecto: nuestro propósito es lograr la primera historia de usuario: la aplicación cargará los datos de los contactos de una fuente determinada. Recordemos que la historia de usuario no debe contener información técnica. Hasta lograr este resultado, nuestra aplicación deberá realizar varios pasos.

Integrando las clases que maneja el Modelo

Por lo tanto, ahora probaremos las clases `ContactList` y `Contact`. Podemos olvidarnos por un momento de `MainTest` y focalizarnos en `Model`. Desde la clase `Model`, probaremos las dos clases antes mencionadas, de la siguiente manera:

```
package com.alfaomegaeditor.model
{
  /**
   * La responsabilidad de esta clase es: ser el Facade de la capa:
Modelo.
   * Crea y destruye los objetos. Es el punto de comunicación entre
esta capa
   * y otras entidades.
   */

  import flash.events.EventDispatcher;

  public class Model extends EventDispatcher
```

```
{
  private var _contactList:ContactList;

  /**
   * Constructor de la clase
   */
  public function Model()
  {
    initialize();
    loadData();
    trace("_contactList:\n" + _contactList.toString());
  }

  /**
   * Carga los datos
   */
  public function loadData():void
  {
    //Desde un Array en donde se encuentran los datos volcados.
    var arrayData:Array = [
                          {
                            name: "name 0",
                            lastName: "lastName 0",
                            phoneNumber: "phoneNumber 0",
                            commentary: "commentary 0"
                          },
                          {
                            name: "name 1",
                            lastName: "lastName 1",
                            phoneNumber: "phoneNumber 1",
                            commentary: "commentary 1"
                          },
                          {
                            name: "name 2",
                            lastName: "lastName 2",
                            phoneNumber: "phoneNumber 2",
                            commentary: "commentary 2"
```

```
                                }
                              ];
      _contactList.addContactsFromArray(arrayData);
    }

    /**
     * Destruye todos los objetos complejos.
     */
    public function destroyObjects():void
    {
      _contactList.destroyObjects();
      _contactList = null
    }

    /**
     * Inicializa el objeto.
     */
    private function initialize():void
    {
      trace("Model ::> initialize");
      createObjects();
    }

    /**
     * Crea los objetos correspondientes.
     */
    private function createObjects():void
    {
      _contactList = new ContactList();
    }
  }
}
```

Y tendremos por salida:

```
MainText ::> initialize
Model ::> initialize
_contactList:
      0)
      name: name 0
      lastName: lastName 0
      phoneNumber: phoneNumber 0
      commentary: commentary 0
      -----------
      1)
      name: name 1
      lastName: lastName 1
      phoneNumber: phoneNumber 1
      commentary: commentary 1
      -----------
      2)
      name: name 2
      lastName: lastName 2
      phoneNumber: phoneNumber 2
      commentary: commentary 2
      ----------
```

Con respecto a `loadData`, no vamos a entrar en detalles sobre la carga de datos externos y su parseo. En su lugar, abstraeremos la obtención de datos. Para este ejercicio nos alcanza con saber que habrá un objeto del tipo: `DataAcces` y que contendrá la propiedad `arrayData`. Entonces, creamos esa clase dentro del paquete `model`, en donde estamos trabajando.

```
package com.alfaomegaeditor.model
{
  /**
   * Esta clase tiene la responsabilidad de contener los datos que
se hayan
   * cargado de una fuente determinada, puede ser un servicio web,
un XML o los
   * datos que se encuentren en alguna otra clase.
   */

  public class DataAccess
  {
    protected var _arrayData:Array;
```

```
    public function DataAccess()
    {
    }

    public function get arrayData():Array
    {
      return _arrayData;
    }
  }
}
```

Los próximos pasos serán crear una clase `Dummy` con datos de prueba, mover el código de prueba de la clase `Model` a `Dummy`. Esta clase estará en el paquete:

```
package com.alfaomegaeditor.testing
```

```
package com.alfaomegaeditor.testing
{
  /**
   * Esta clase es para hacer pruebas durante el desarrollo.
   */

  import com.alfaomegaeditor.model.DataAccess;
  public class Dummy extends DataAccess
  {
    protected var _dummyData:Array = [
                                        {
                                          name:"Master",
                                          lastName:"Yoda",
                                          phoneNumber:"1111-1111",
                                          commentary:"El ser con ma-
yor conocimiento Jedi. Está haciendo un curso de gramática."
                                        },
                                        {
```

```
                                        name:"Luke",
                                        lastName:"Skywalker",
                                        phoneNumber:"2222-2222",
                                        commentary:"Es su tiempo
libre juega al Farmville, dice que le hace recordar sus orígenes."
                                      },
                                      {
                                        name:"Darth",
                                        lastName:"Vader",
                                        phoneNumber:"3333-3333",
                                        commentary:"Tiene cualida-
des de liderazgo. Gran potencial para conquistar la galaxia."
                                      },
                                      {
                                         name:"Princesa",
                                         lastName:"Leia",
                                         phoneNumber:"4444-4444",
                                         commentary:"Luego de ven-
cer al Imperio, se transformó en líder de la liberación galáctica
femenina."
                                      }
                                    ];


    public function Dummy()
    {
      super();
      _arrayData = _dummyData;
    }
  }
}
```

En la clase `Model`, haremos un cambio en el método: `loadData`. Creará un objeto del tipo: `DataAccess` y ya no tendremos código de prueba directamente en esta clase. Crearemos entonces la clase `DataAccess` dentro de `Model`.

```
package com.alfaomegaeditor.model
{
```

```
  /**
   * La responsabilidad de esta clase es: ser el Facade de la capa:
Modelo.
   * Crea y destruye los objetos. Es el punto de comunicación entre
esta capa
   * y otras entidades.
   */

  import com.alfaomegaeditor.testing.Dummy;

  import flash.events.EventDispatcher;

  public class Model extends EventDispatcher
  {
    private var _contactList:ContactList;

    /**
     * Constructor de la clase
     */
    public function Model()
    {
      initialize();
      loadData();
      trace("_contactList:\n" + _contactList.toString());
    }

    /**
     * Carga los datos
     */
    public function loadData():void
    {
      //Creamos el objeto DataAcces.
      var dataAccess:DataAccess = new Dummy();
      //Tomamos los datos de un DataAcces.
      var arrayData:Array = dataAccess.arrayData;

      _contactList.addContactsFromArray(arrayData);
```

```
    }

    /**
     * Destruye todos los objetos complejos.
     */
    public function destroyObjects():void
    {
      _contactList.destroyObjects();
      _contactList = null
    }

    /**
     * Inicializa el objeto.
     */
    private function initialize():void
    {
      trace("Model ::> initialize");
      createObjects();
    }

    /**
     * Crea los objetos correspondientes.
     */
    private function createObjects():void
    {
      _contactList = new ContactList();
    }
  }
}
```

La estructura de clases y paquetes quedaría como muestra la figura:

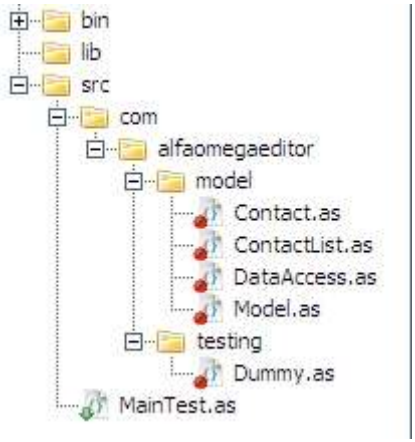

Fig. 9

Con esto damos por terminado la historia de usuario:

1. La aplicación cargará los datos de los contactos de una fuente determinada.

La salida que tenemos ahora es la siguiente:

```
MainText ::> initialize
Model ::> initialize
_contactList:
      0)
      name: Master
      lastName: Yoda
      phoneNumber: 1111-1111
      commentary: El ser con mayor conocimiento Jedi. Está
haciendo un curso de gramática.
      -----------
      1)
      name: Luke
      lastName: Skywalker
      phoneNumber: 2222-2222
      commentary: Es su tiempo libre juega al Farmville,
dice que le hace recordar sus orígenes.
      -----------
      2)
      name: Darth
      lastName: Vader
      phoneNumber: 3333-3333
```

```
      commentary: Tiene cualidades de liderazgo. Gran po-
tencial para conquistar la galaxia.
      -----------
      3)
      name: Princesa
      lastName: Leia
      phoneNumber: 4444-4444
      commentary: Luego de vencer al Imperio, se transformó
en líder de la liberación femenina galáctica.
      -----------
```

Ahora, debemos implementar las comunicaciones que recibirá el `Model`. Una de ellas ya está implementada, y es el método: `loadData`, que se llama directamente.

Entonces, estableceremos los métodos que manejarán los eventos de recepción de pedido de cambio de índice. Estos pedidos de cambio de índices serán determinados en el objeto `ControllerEvent`. Esto nos permitirá cumplir con las historia de usuario:

2. Como usuario deseo seleccionar el contacto desde "Panel lista de Contactos": al hacer clic sobre algunos de los contactos, se seleccionará el contacto respectivo.
3. Como usuario deseo seleccionar el contacto desde "Panel de navegación": al hacer clic sobre algunos de los botones: "anterior" o "siguiente", se seleccionará el contacto respectivo.

Implementando el ControllerEvent

Los datos para el pedido de cambio de índice por cualquiera de estos botones quedarán encapsulados en el objeto:

`com.alfaomegaeditor.events.ControllerEvent`.

Veamos cómo queda implementada esta clase:

```
package com.alfaomegaeditor.events
{
  import com.alfaomegaeditor.model.Contact;
  import com.alfaomegaeditor.utils.cloneObject;
```

```
import flash.events.Event;

public class ControllerEvent extends Event
{
  public static const ASK_INDEX:String = "askIndex";
  public static const ASK_NEXT_INDEX:String = "askNextIndex";
  public static const ASK_PREV_INDEX:String = "askPrevIndex";

  public static const UPDATE_INDEX:String = "updateIndex";

  private var _contactIndex:uint;
  private var _totalContacts:uint;

  private var _fullName:String;
  private var _phoneNumber:String;
  private var _commentary:String;

  public function ControllerEvent(type:String,
                                  contactIndex:uint=0,
                                  totalContacts:uint=0,
                                  fullName:String="",
                                  phoneNumber:String="",
                                  commentary:String="")
  {
    super(type);

    _contactIndex = contactIndex;
    _totalContacts = totalContacts;

    _fullName = fullName;
    _phoneNumber = phoneNumber;
    _commentary = commentary;
  }

  public function get contactIndex():uint
  {
```

```
      return _contactIndex;
    }

    public function get totalContacts():uint
    {
      return _totalContacts;
    }

    public function get fullName():String
    {
      return _fullName;
    }

    public function get phoneNumber():String
    {
      return _phoneNumber;
    }

    public function get commentary():String
    {
      return _commentary;
    }

    override public function clone():Event
    {
      var event:ControllerEvent = Control-
lerEvent(cloneObject(this));
      return event;
    }
  }
}
```

Algo a destacar es que en este objeto, se guardarán los datos de `Contact` en distintas propiedades. Esto permite que objetos que no lo conocen puedan manejar un evento de este tipo. Recordemos que nuestro `Controller`, actuará como un `Event Aggregator`. Algo ventajoso es que utiliza propiedades primitivas que

cualquier objeto en cualquier paquete puede usar sin necesidad de importar otras clases. Una desventaja puede ser la cantidad de propiedades que son necesarias pasar a través del constructor.

Veamos otro detalle. Las propiedades solo tienen un *getter*, es decir, son de solo lectura. Lo que permite definir la propiedad una sola vez (cuando es creado el objeto en este caso). Y luego no puede ser modificada.

El que las propiedades de un evento sean de solo lectura es una característica habitual para evitar que cualquier objeto modifique el estado del evento cuando es despachado.

Por último, hemos incorporado una función que se encarga de hacer un clon de un objeto y de todos los objetos que éste pueda contener y lo utilizamos en el método `clone`.

Esta función se encontrará en:

`com.alfaomegaeditor.utils.cloneObject`

El código se ve a continuación:

```
package com.alfaomegaeditor.utils
{
  import flash.utils.ByteArray;

  /**
   * Crea una copia profunda de un objeto y los objetos que pudiera
   * tener dentro.
   * @param source: objeto a copiar.
   * @return: un clon del objeto y de los objetos contenidos dentro
de este.
   */
  public function cloneObject(source:Object):*
  {
    var byteArray:ByteArray = new ByteArray();
    byteArray.writeObject(source);
    byteArray.position = 0;
    return(byteArray.readObject());
  }
}
```

Model: recepción de eventos

Dando al `Model` la capacidad de manejar eventos del tipo `ControllerEvent`, estaremos estableciendo las bases de una comunicación desacoplada entre el Modelo y el resto de las entidades. Agregaremos los manejadores de evento para estos tres pedidos de cambio de índice: índice anterior, índice siguiente e índice determinado.

Por ahora, pondremos un comentario indicando que deberá despacharse un evento en caso que el índice haya cambiado. Volveremos a este tema muy pronto cuando veamos envío de eventos.

En el `Model` importamos la clase:

`com.alfaomegaeditor.events.ControllerEvent`

Agregamos los tres manejadores de evento, ahora `Model`, nos quedaría así:

```
package com.alfaomegaeditor.model
{
  /**
   * La responsabilidad de esta clase es: ser el Facade de la capa:
Modelo.
   * Crea y destruye los objetos. Es el punto de comunicación entre
esta capa
   * y otras entidades.
   */

  import com.alfaomegaeditor.events.ControllerEvent;

  import com.alfaomegaeditor.testing.Dummy;

  import flash.events.EventDispatcher;

  public class Model extends EventDispatcher
  {
    private var _contactList:ContactList;
```

```
/**
 * Constructor de la clase
 */
public function Model()
{
  initialize();
  loadData();
  trace("_contactList:\n" + _contactList.toString());
}

/**
 * Carga los datos
 */
public function loadData():void
{
  //Creamos el objeto DataAcces.
  var dataAccess:DataAccess = new Dummy();
  //Tomamos los datos de un DataAcces.
  var arrayData:Array = dataAccess.arrayData;

  _contactList.addContactsFromArray(arrayData);
}

/**
 * Manejadores de los eventos de cambio de índice.
 */
public function onAskChangeIndex(event:ControllerEvent):void
{
  //Obtenemos el índice a cambiar.
  var index:uint = event.contactIndex;
  //Cambiamos el índice en _contactList.
  _contactList.currentIndex = index;

  //Si se produjo un cambio de índice despachar evento.
}
public function onAskNextIndex(event:ControllerEvent):void
```

```
    {
      _contactList.nextIndex();

      //Si se produjo un cambio de índice despachar evento.
    }
    public function onAskPrevtIndex(event:ControllerEvent):void
    {
      _contactList.prevIndex();

      //Si se produjo un cambio de índice despachar evento.
    }

    /**
     * Destruye todos los objetos complejos.
     */
    public function destroyObjects():void
    {
      _contactList.destroyObjects();
      _contactList = null
    }

    /**
     * Inicializa el objeto.
     */
    private function initialize():void
    {
      trace("Model ::> initialize");
      createObjects();
    }

    /**
     * Crea los objetos correspondientes.
     */
    private function createObjects():void
    {
      _contactList = new ContactList();
    }
```

```
  }
}
```

En este punto, tendríamos que probar que cada uno de los eventos funcione, haciendo pruebas suscribiendo al `Model` a estos eventos, y luego despacharlos, comprobando que todo funcione bien. Para no irnos de foco, estas pruebas quedan a discreción del lector.

Model: envío de eventos.

Por su lado, el `Model` enviará eventos del tipo `ModelEvent`. Estos serán a los que se suscribirá el `Controller` (léase `EventAggregator`). El `Controller` recibirá estos eventos y los "traducirá" a `ControllerEvent`, luego lo despachará. Los eventos que despachará el `Model` serán:

`com.alfaomegaeditor.events.DataEvent.DATA_READY:`

Cuando tenga disponible los datos de los contactos, este evento, en nuestra prueba, será disparado desde el método: `loadData`. Es la manera que tendrá el `Model` de anunciar que los datos están listos para ser utilizados.

Esto contribuye a las historias de usuario:

2. A. El sistema desplegará en la "Panel lista de contactos", botones con el nombre y apellido de contacto.

2. B. Se seleccionará el primer contacto en forma automática, cada vez que cambie el índice de la lista de contactos, lo cual contribuye a las historia de usuario desde la 2 a la 5:

`com.alfaomegaeditor.events.ModelEvent.UPDATE_INDEX`

Y aquí la clase: `com.alfaomegaeditor.events.DataEvent`

```
package com.alfaomegaeditor.events
{
  import com.alfaomegaeditor.utils.cloneObject;
```

```
  import flash.events.Event;

  public class DataEvent extends Event
  {
    public static const DATA_READY:String = "dataReady";

    private var _arrayData:Array;

    public function DataEvent(type:String, arrayData:Array)
    {
      super(type);
      _arrayData = arrayData;
    }

    public function get arrayData():Array
    {
      return _arrayData;
    }

    override public function clone():Event
    {
     var event:DataEvent= DataEvent(cloneObject(this));
      return event;
    }
  }
}
```

La clase: `com.alfaomegaeditor.events.ModelEvent`

```
package com.alfaomegaeditor.events
{
  import com.alfaomegaeditor.model.Contact;
  import com.alfaomegaeditor.utils.cloneObject;
```

```
import flash.events.Event;

public class ModelEvent extends Event
{
  public static const ASK_INDEX_CHANGE:String = "askIndexChange";
  public static const ASK_NEXT_INDEX:String = "askNextIndex";
  public static const ASK_PREV_INDEX:String = "askPrevIndex";
  public static const UPDATE_INDEX:String = "updateIndex";

  private var _contact:Contact;
  private var _contactIndex:uint;
  private var _totalContacts:uint;

  public function ModelEvent(type:String,
                             contact:Contact,
                             contactIndex:uint,
                             totalContacts:uint)
  {
     super(type);
    _contact = contact;
    _contactIndex = contactIndex;
    _totalContacts = totalContacts;
  }

  public function get contact():Contact
  {
    return _contact;
  }

  public function get contactIndex():uint
  {
    return _contactIndex;
  }

  public function get totalContacts():uint
  {
```

```
      return _totalContacts;
    }

    override public function clone():Event
    {
      var event:ModelEvent = ModelEvent(cloneObject(this));
      return event;
    }
  }
}
```

Refinando la clase Model

Ahora bien, importamos las clases necesarias, luego, hacemos los llamados correspondientes desde los manejadores de evento, e implementamos los métodos necesarios para despachar los eventos. Finalmente, sacamos los `trace`.

```
package com.alfaomegaeditor.model
{
  /**
   * La responsabilidad de esta clase es: ser el Facade de la capa:
Modelo.
   * Crea y destruye los objetos. Es el punto de comunicación entre
esta capa
   * y otras entidades.
   */

  import com.alfaomegaeditor.events.DataEvent;
  import com.alfaomegaeditor.events.ModelEvent;
  import com.alfaomegaeditor.events.ControllerEvent;

  import com.alfaomegaeditor.testing.Dummy;

  import flash.events.EventDispatcher;

  public class Model extends EventDispatcher
```

```
  {
    private var _contactList:ContactList;

    /**
     * Constructor de la clase
     */
    public function Model()
    {
      initialize();
    }

    /**
     * Carga los datos
     */
    public function loadData():void
    {
      //Creamos el objeto DataAcces.
      var dataAccess:DataAccess = new Dummy();
      //Tomamos los datos de un DataAcces.
      var arrayData:Array = dataAccess.arrayData;

      _contactList.addContactsFromArray(arrayData);

      //Despachamos evento de que los datos están listos.
      var event:DataEvent = new DataEvent(DataEvent.DATA READY, ar-
rayData);
      dispatchEvent(event);

      //Si existe algún contacto, despachar evento ModelE-
vent.UPDATE_INDEX del
      //índice: 0.
      if (_contactList.length > 0)
      {
        createAndDispatchEvent(ModelEvent.UPDATE_INDEX, 0);
      }
    }
```

```
/**
 * Manejadres de los eventos de cambio de índice.
 */
public function onAskChangeIndex(event:ControllerEvent):void
{
  //Obtenemos el índice a cambiar.
  var index:uint = event.contactIndex;
  //Cambiamos el índice en _contactList.
  _contactList.currentIndex = index;

  //Si se produjo un cambio de índice despachar evento.
  tryToDispatchEvent();
}
public function onAskNextIndex(event:ControllerEvent):void
{
  _contactList.nextIndex();

  //Si se produjo un cambio de índice despachar evento.
  tryToDispatchEvent();
}
public function onAskPrevtIndex(event:ControllerEvent):void
{
  _contactList.prevIndex();

  //Si se produjo un cambio de índice despachar evento.
  tryToDispatchEvent();
}

/**
 * Destruye todos los objetos complejos.
 */
public function destroyObjects():void
{
  _contactList.destroyObjects();
  _contactList = null
}
```

```
    /**
     * Inicializa el objeto.
     */
    private function initialize():void
    {
      createObjects();
    }

    /**
     * Crea los objetos correspondientes.
     */
    private function createObjects():void
    {
      _contactList = new ContactList();
    }

    /**
     * Si el índice cambió, se despacha el evento.
     */
    private function tryToDispatchEvent():void
    {
      //Si el índice cambió, despachamos el evento.
      if (_contactList.indexHasChanged)
      {
        createAndDispatchEvent(ModelEvent.UPDATE_INDEX,
            _contactList.currentIndex);
      }
    }

    /**
     * Despacha evento del Model.
     */
    private function createAndDispatchEvent(type:String, in-
dex:uint):void
    {
      var contact:Contact = _contactList.getContactAt(index);
      var lenght:uint = _contactList.length;
```

```
      var event:ModelEvent = new ModelEvent(type, contact, index,
lenght);
      dispatchEvent(event);
    }
  }
}
```

Ahora la estructura de clases y paquetes sería la siguiente:

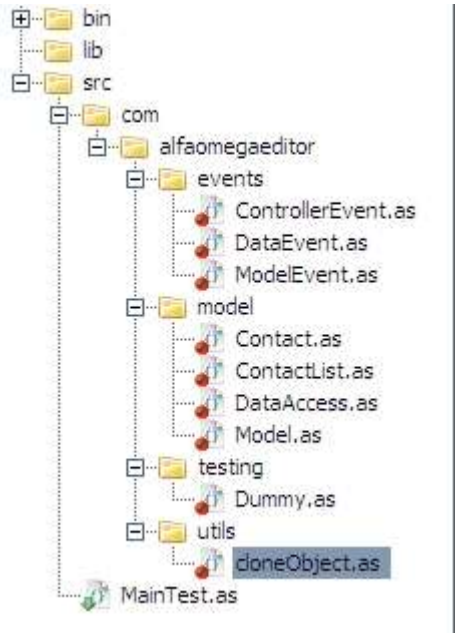

Fig. 10

Implementando la capa de la Vista

Una de las prácticas más habituales al trabajar con MVC es la de no agregar ningún tipo de lógica para el manejo de datos, sino simplemente representarlos. En esta capa es donde también estarán las representaciones de los botones. Para el caso de nuestro ejercicio serán el nexo con el usuario. La Vista estará representada por la clase: `View`. Esta clase estará suscripta a los eventos del tipo `ControllerEvent` que despachará el `Controller`. El `Controller` a su vez se suscribirá a los eventos del `View`

relacionados con el clic de los distintos botones. El `Controller` traducirá estos eventos y los despachará. Lo cual permitirá que tanto el `Model` como el `View` se mantengan comunicados y consistentes. Al trabajar en esta capa estaremos aportando a las historias de usuario desde la 2 hasta la 5.

En el manejo de la Vista, utilizaremos las siguientes clases:

`View:` es el `Facade` de la capa correspondiente a la Vista. Crea y destruye los objetos. Es el punto de comunicación entre la capa de la Vista y otras entidades.
`ContactListPanel:` despliega un botón por cada contacto y a través de ellos se selecciona el contacto correspondiente.
`Button:` esta clase representa un botón con un cuadrado de fondo y un texto. Es utilizado por las clases: `ContactListPanel` y por `NavegationPanel`.
`ContactPanel:` muestra los datos del contacto seleccionado.
`NavegationPanel:` tiene dos botones que permiten seleccionar el contacto: anterior ó siguiente, con respecto al contacto seleccionado. Tiene un campo de texto en donde se muestra el índice del contacto actual y cuántos contactos existen en total.

La clase View

Esta clase será la encargada de manejar los paneles y ser el `Facade` entre las clases visuales y el resto de las entidades de la aplicación. Esta clase creará y configurará los paneles. Y cada panel será responsable de construirse a sí mismo. Creamos la carpeta `view` al mismo nivel que `model` y dentro agregamos la siguiente clase `View:`

```
package com.alfaomegaeditor.view
{
  /**
   * La responsabilidad de esta clase es: ser el Facade de la capa:
Vista.
   * Crea y destruye los objetos. Es el punto de comunicación entre
esta capa
   * y otras entidades.
   */

  import flash.display.Sprite

  public class View extends Sprite
```

```
  {
    /**
     * Constructor de la clase.
     */
    public function View()
    {
      initialize();
    }

    /**
     * Destruye todos los objetos complejos.
     */
    public function destroyObjects():void
    {
    }

    /**
     * Inicializa el objeto.
     */
    private function initialize():void
    {
      trace("View ::> initialize");
    }
  }
}
```

Y para probar esta clase haremos ajustes a la clase: `MainTest`.

```
package
{
  /**
   * Punto de entrada de la aplicación.
   * Su responsabilidad es manejar los objetos del tipo:
   * Model, View y Controller.
```

```
   * Esto implica, crearlos, eventualmente inicializarlos y des-
truirlos.
   */

  import com.alfaomegaeditor.model.Model;
  import com.alfaomegaeditor.view.View;

  import flash.display.Sprite;

  public class MainTest extends Sprite
  {
    private var _model:Model;
    private var _view:View;

    /**
     * Constructor de la clase
     */
    public function MainTest()
    {
      initialize();
    }

    /**
     * Inicializa la clase.
     */
    private function initialize():void
    {
      trace("MainText ::> initialize");
      createObjects();
      addChild(_view);
    }

    /**
     * Crea todos los objetos necesarios.
     */
    private function createObjects():void
    {
```

```
        _model = new Model();
        _view = new View();
      }
    }
}
```

Compilamos, probamos y vemos la salida:

```
MainText ::> initialize
View ::> initialize
```

Ahora, nos concentraremos en las clases que instanciarán la clase `View`.

ContactListPanel

Continuamos por la clase: `ContactListPanel`.
Esta clase se encargará de desplegar el panel de los botones de contacto.
Tendremos dos objetos gráficos: `_rectOutline:Shape`, que será el recuadro de la botonera; y `_buttonsContainer:Sprite`, en donde se desplegarán los botones.

Para que aparezca el recuadro de la botonera, el constructor recibirá los parámetros: `width` y `height`, que definirán el alto y ancho del recuadro.

Como hemos estado viendo en otros ejemplos, el `initialize` de `View` realizará los siguientes pasos:

- Guardar el ancho y el alto que se le pase al constructor.
- Crear el contenedor de botones y el marco de la botonera.
- Se configuran las partes del panel.
- Y se hacen los `addChild` correspondientes.

Para que visualmente la clase quede más clara, utilizaremos un único método para hacer todo en el `buildPanels`. Queda como tarea para el lector pulir este aspecto de fraccionar este método.

Tiene dos métodos públicos:

- **`displayData(arrayData:Array):void ->`** desde un `array` de objetos que deben tener las propiedades: `name` y `lastName`, despliega un botón por cada ítem del `Array`. Este método utiliza la clase `Button`.
- **`selectByIndex(indexSelected:uint):void ->`** mantiene seleccionado un solo botón de la botonera correspondiente, al índice que se le pasa como parámetro.

Y la clase `ContactListPanel` quedaría así.

```
package com.alfaomegaeditor.view
{
  import com.alfaomegaeditor.utils.createRectangle;

  import flash.display.Shape;
  import flash.display.Sprite;

  public class ContactListPanel extends Sprite
  {
    private var _width:Number;
    private var _height:Number;
    private var _buttonHeight:Number = 20;
    private var _rectOutline:Sprite;
    private var _buttonsContainer:Sprite;

    public function ContactListPanel(width:Number,
height:Number):void
    {
      initialize(width, height);
    }

    /**
     * Dado un array despliega los botones correspondientes.
     * @param arrayData: array con los datos a desplegar.
     */
    public function displayData(arrayData:Array):void
```

```
    {
      var buttonText:String
      var contact:Object;
      var index:uint = 0;
      var arrayDataLenght:uint = arrayData.length;
      while (index < arrayDataLenght)
      {
        contact = arrayData[index];
        //Armamos el texto del botón.
        buttonText = contact.name + " " + contact.lastName;
        //Creamos el botón.
        var button:Button = new Button(buttonText,  width,
_buttonHeight);
        //Lo ubicamos
        button.y = _buttonHeight * _buttonsContainer.numChildren;
        //Lo agregamos al container.
         _buttonsContainer.addChild(button);
        index++
      }
    }

    /**
     * Dado un índice, selecciona el botón correspondiente.
     * @param indexSelected: botón a seleccionar en función del ín-
dice.
     */
    public function selectByIndex(indexSelected:uint):void
    {
      //Deseleccionamos todos los botones.
      var button:Button;
      var index:int = _buttonsContainer.numChildren;
      while (--index > -1)
      {
        button = Button(_buttonsContainer.getChildAt(index));
        button.deselect();
      }
```

```
      //Cantidad de hijos de un _buttonsContainer.
      var topIndex:uint = _buttonsContainer.numChildren;
      //Si _buttonsContainer está vacío, detener el método.
      var isEmpty:Boolean = (topIndex == 0);
      if (isEmpty)
      {
        return
      }

      //Si el índice seleccionado es mayor a la cantidad de hijos
      //de  buttonsContainer, se le asigna el máximo índice dispo-
nible.
      if (indexSelected > topIndex)
      {
          indexSelected = (topIndex-1);
      }

      //Seleccionamos el botón correspondiente al índice.
      button = Button(_buttonsContainer.getChildAt(indexSelected));
      button.select();
    }

    private function initialize(width:Number, height:Number):void
    {
      buildPanel(width, height);
    }

    /**
     * Crea el panel con todos los items gráficos.
     */
    private function buildPanel(width:Number, height:Number):void
    {
      //Configuramos el ancho y alto.
      _width = width;
      _height = height;
      //Creamos los items gráficos.
      _buttonsContainer = new Sprite();
```

```
      _rectOutline = createRectangle(0xFFFFFF, _width, _height, 1);
      //Agregamos los items gráficos.
      addChild(_rectOutline);
      addChild(_buttonsContainer);
    }

    /**
     * Destruye todos los objetos complejos.
     */
    public function destroyObjects():void
    {
      while (_buttonsContainer.numChildren > 0)
      {
        var button:Button = But-
ton(_buttonsContainer.removeChildAt(0));
      }
      removeChild(_buttonsContainer);
      _buttonsContainer = null;

      removeChild(_rectOutline);
      _rectOutline = null;
    }
  }
}
```

createRectangle

La clase `Button` utilizará la función `createRectangle` que se encontrará en `util`, y que devolverá un `Sprite` con un rectángulo dentro, con el color alto y ancho que se le pase como parámetros.

Esta es la función implementada:

```
package com.alfaomegaeditor.utils
{
  import flash.display.Sprite;

  /**
   * Función que crear un Sprite y dibujar dentro un rectángulo.
   * @param color: color de fondo.
   * @param width: ancho del rectángulo.
   * @param height: alto del rectángulo.
   * @param thickness: grosor de la línea.
   * @param lineColor: color del borde.
   * @return: devuelve un Sprite con un rectángulo dentro.
   */
  public function createRectangle(color:Number,
                                  width:Number,
                                  height:Number,
                                  thickness:Number=0,
                                  lineColor:Number=0):Sprite
    {
      var rectangle:Sprite = new Sprite();
      if (thickness > 0)
      {
        rectangle.graphics.lineStyle(thickness, lineColor);
      }
      rectangle.graphics.beginFill(color);
      rectangle.graphics.drawRect(0, 0, width, height);
      return rectangle;
    }
}
```

Button

Ahora, implementaremos la clase `Button`. Le pasaremos como parámetros el texto del botón, ancho y alto. Los eventos de `MouseEvent`: `ROLL_OVER` y `ROLL_OUT`,

los manejará el botón. El seleccionar y el deshacer, la activación y la desactivación se manejarán desde el `ContactListPanel`.

Y ésta es la clase `Button` implementada.

```
package com.alfaomegaeditor.view
{
  /**
   * Clase encargada de representar y dar funcionalidad a un botón.
   */

  import flash.text.TextField;
  import flash.display.Sprite;
  import flash.events.MouseEvent;

  import com.alfaomegaeditor.utils.createRectangle;
  import com.alfaomegaeditor.view.View

  public class Button extends Sprite
  {
    protected const ALPHA_DEACTIVATED:Number = .15;
    protected const ALPHA_ACTIVATED:Number = 1;
    protected var _isSelected:Boolean;
    protected var _overBackground:Sprite;
    protected var _selectionBackground:Sprite;
    protected var _buttonBackground:Sprite;
    protected var _textField:TextField;
    protected var _textButton:String;
    protected var _width:Number;
    protected var _height:Number;

    /**
    * Constructor.
    * @param textButton: texto del botón.
    * @param width: ancho del botón.
    * @param height: alto del botón
    */
```

```
    public function Button(textButton:String, width:Number,
height:Number)
    {
      initialize(textButton, width, height);
    }

    /**
     * Representa la selección del botón.
     */
    public function select():void
    {
      //Hacemos visible el fondo de selección.
      _selectionBackground.visible = true;
    }

    /**
     * Representa la deselección del botón.
     */
    public function deselect():void
    {
      //Hacemos invisible el fondo de selección.
      _selectionBackground.visible = false;
    }

    /**
     * Activa el botón.
     */
    public function activate():void
    {
      //Activamos el botón.
      mouseEnabled = true;
      _overBackground.alpha = ALPHA_ACTIVATED;
      _selectionBackground.alpha = ALPHA_ACTIVATED;
      _buttonBackground.alpha = ALPHA_ACTIVATED;
    }

    /**
```

```
     * Desactiva el botón.
     */
    public function desactivate():void
    {
      //Desactivamos el botón.
      mouseEnabled = false
      _overBackground.alpha = ALPHA_ACTIVATED;
      _selectionBackground.alpha = ALPHA_DEACTIVATED;
      _buttonBackground.alpha = ALPHA_DEACTIVATED;
    }

    /**
     * Manejador del evento MouseEvent.ROLL_OVER
     * @param event: evento que envía ActionScript.
     */
    public function onRollOver(event:MouseEvent):void
    {
      //Resaltar el botón.
      _overBackground.visible = true;
    }

    /**
     * Manejador del evento MouseEvent.ROLL_OUT
     * @param event: evento que envía ActionScript.
     */
    public function onRollOut(event:MouseEvent):void
    {
      //Mostrar el botón en reposo.
      _overBackground.visible = false;
    }

    /**
     * Inicializa el objeto.
     */
    private function initialize(textButton:String,
                                width:Number,
                                height:Number):void
```

```
    {
      buildButton(textButton, width, height);
      setEventListener();
    }

    /**
     * Construye el botón.
     */
    protected function  buildButton(textButton:String,
                                    width:Number,
                                    height:Number):void
    {
      //Nombramos al botón.
      name = textButton;
      //Asignamos el texto, alto y ancho.
      _textButton = textButton;
      _width = width;
      _height = height;
      //Creamos los items del botón.
      _selectionBackground = createRectangle(View.COLOR_BLACK,
                                             _width, _height);
      _overBackground = createRectangle(View.COLOR_LIGHT_GRAY,
                                             _width, _height);
      _buttonBackground = createRectangle(View.COLOR_DARK_GRAY,
                                             _width, _height);
      _textField = new TextField();
      //Configuramos sus propiedades iniciales.
      _selectionBackground.visible = false;
      _overBackground.visible = false;
      _textField.width = _width;
      _textField.height = _height;
      _textField.text = _textButton;
      _textField.textColor = View.COLOR_WITHE;
      this.buttonMode = true;
      this.mouseChildren = false;
      //Se agregan los items gráficos.
```

```
      _buttonBackground.name = "_buttonBackground";
      _overBackground.name = "_overBackground";
      _selectionBackground.name = "_selectionBackground";
      _textField.name = "_textField";
      addChild(_buttonBackground);
      addChild(_overBackground);
      addChild(_selectionBackground);
      addChild(_textField);
    }
    /**
     * Configura los eventos de ROLL_OVER y ROLL_OUT.
     */
    protected function setEventListener():void
    {
      //Debemos configurar los parámetros useCapture y priority
      //para poder usar referencia débil.
      var useCapture:Boolean = false;
      var priority:uint = 0;
      var useWeakReference:Boolean = true;
      this.addEventListener(MouseEvent.ROLL_OVER, onRollOver,
                            useCapture, priority, useWeakRefer-
ence);
      this.addEventListener(MouseEvent.ROLL_OUT, onRollOut,
                            useCapture, priority, useWeakReferen-
ce);
    }
  }
}
```

NavegationPanel

Este panel emitirá los eventos relacionados con pedir el índice anterior o siguiente. Y en él se mostrará cuál es el índice actual y la cantidad total de contactos. Estos datos se representan a través del método: `displayData`. La implementación de esta clase es la siguiente:

```
package com.alfaomegaeditor.view
{
```

```
  import flash.display.Sprite;
  import flash.text.TextField;

  import com.alfaomegaeditor.events.ControllerEvent;
  import com.alfaomegaeditor.view.Button;

  public class NavegationPanel extends Sprite
  {
    public static const PREV:String = "Anterior";
    public static const NEXT:String = "Siguiente";
    private const GAP:Number = 10;
    private var _height:Number = 20;
    private var _width:Number = 50;
    private var _leftButton:Button;
    private var _rightButton:Button;
    private var _indexTotalContacts:TextField;
    private var _nextPrevContainer:Sprite;

    public function NavegationPanel()
    {
      buildPanel();
    }

    /**
     * Muestra el índice del contacto actual y el total de contac-
tos.
     * @param contactIndex:índice del contacto actual.
     * @param totalContacts: total de contactos.
     */
    public function displayData(contactIndex:uint, totalCon-
tacts:uint):void
    {
      var indexToDisplay:uint = (contactIndex + 1);
       indexTotalContacts.text = indexToDisplay + "/" + totalCon-
tacts;


      _rightButton.activate();
      _leftButton.activate();
```

```
      if (indexToDisplay == 1)
      {
        _leftButton.deactivate();
      }
      if (indexToDisplay == totalContacts)
      {
        _rightButton.deactivate();
      }
    }
    /**
     * Arma el panel.
     */
    private function buildPanel():void
    {
      mouseEnabled = false;
      //Creamos los items gráficos.
      var prev:String = NavegationPanel.PREV;
      _leftButton = new Button(prev, _width, _height);
      //Nombramos al botón.
      _leftButton.name = prev;
      var next:String = NavegationPanel.NEXT;
      _rightButton = new Button(next, _width, _height);
      //Nombramos al otro botón.
      _rightButton.name = next;
      _indexTotalContacts = new TextField();
      _indexTotalContacts.mouseEnabled = false;
      _indexTotalContacts.name = "_indexTotalContacts"
      _nextPrevContainer = new Sprite();
      _nextPrevContainer.mouseEnabled = false;
      _nextPrevContainer.name = "_nextPrevContainer"
      //Configuramos sus propieddades.
      var textPosition:Number = GAP +  leftButton.x +
_leftButton.width;
      _indexTotalContacts.x += textPosition;
      _indexTotalContacts.height = _height;
      _indexTotalContacts.width = _height;
```

```
        rightButton.x = textPosition +  indexTotalContacts.width +
GAP;
       //Agregamos los items gráficos.
       _nextPrevContainer.addChild(_leftButton);
       _nextPrevContainer.addChild(_rightButton);
       _nextPrevContainer.addChild(_indexTotalContacts);
       addChild(_nextPrevContainer);
     }
     /**
      * Destruye todos los objetos complejos.
      */
     public function destroyObjects():void
     {
       _nextPrevContainer.removeChild(_leftButton);
       _nextPrevContainer.removeChild(_rightButton);
       _nextPrevContainer.removeChild(_indexTotalContacts);
       removeChild(_nextPrevContainer);
       _leftButton = null;
       _rightButton = null;
       _indexTotalContacts = null;
       _nextPrevContainer = null;
     }
   }
}
```

ContactPanel

Esta clase no emitirá eventos, tiene un método público: `displayData`, que será llamado por el `View`, y le pasará los parámetros a desplegar en el panel.

Aquí tenemos la clase implementada:

```
package com.alfaomegaeditor.view
{
  import com.alfaomegaeditor.utils.createRectangle;

  import flash.display.Sprite;
  import flash.text.TextField;
```

```
  public class ContactPanel extends Sprite
  {
    private var _width:Number;
    private var _height:Number;
    private var _gapY:Number = 25;
    private var _initialX:Number = 5;
    private var _initialY:Number = 5;
    private var _commentaryHeight:Number = 60;
    private var _rectOutline:Sprite;
    private var _nameLastName:TextField;
    private var _phoneNumber:TextField;
    private var _commentary:TextField;

    public function ContactPanel(width:Number, height:Number):void
    {
      buildPanel(width, height);
    }

    /**
     * Despliega los parámetros que se le pasa:
     * @param fullName: nombre y apellido.
     * @param phoneNumber: número de teléfono.
     * @param commentary: comentario.
     */
    public function displayData(fullName:String,
                                phoneNumber:String,
                                commentary:String):void
    {
      //Al cambiar el índice, asignamos los textos correspondien-
tes.
      _nameLastName.text = fullName;
      _phoneNumber.text = phoneNumber;
      _commentary.text = commentary;
    }

    private function buildPanel(width:Number, height:Number):void
```

```
    {
      _width = width;
      _height = height;
      var textWidth:Number = _width - 10;
      var xPosition:Number = _initialX;
      var yPosition:Number = _initialY;
      //Creamos y ubicamos los campos de texto.
      _nameLastName = new TextField();
      _nameLastName.width = textWidth;
      _nameLastName.x = xPosition;
      _nameLastName.y = yPosition;
      yPosition += _gapY;
      _phoneNumber = new TextField();
      _phoneNumber.width = textWidth;
      _phoneNumber.x = xPosition;
      _phoneNumber.y = yPosition;
      yPosition += _gapY;
      _commentary = new TextField();
      _commentary.width = textWidth;
      _commentary.height = _commentaryHeight;
      _commentary.multiline = true;
      _commentary.wordWrap = true;
      _commentary.x = xPosition;
      _commentary.y = yPosition;
      //Creamos el rectángulo de marco.
      _rectOutline = createRectangle(0xFFFFFF, _width, _height, 1);
      //Agregamos los DisplayObject.
      addChild(_rectOutline);
      addChild(_nameLastName);
      addChild(_phoneNumber);
      addChild(_commentary);
    }

    /**
     * Destruye todos los objetos complejos.
     */
```

```
    public function destroyObjects():void
    {
      removeChild(_rectOutline);
      removeChild(_nameLastName);
      removeChild(_phoneNumber);
      removeChild(_commentary);
      _rectOutline = null;
      _nameLastName = null;
      _phoneNumber = null;
      _commentary = null;
    }
  }
}
```

Ahora, modificaremos el `View` para probar que se representen correctamente los objetos visuales y que podamos reflejar en ellos datos. El método con el cual haremos eso será: `displayData`.

La clase `View` quedaría como sigue:

```
package com.alfaomegaeditor.view
{
  /**
   * La responsabilidad de esta clase es: ser el Facade de la capa:
Vista.
   * Crea y destruye los objetos. Es el punto de comunicación entre
esta capa
   * y otras entidades.
   */

  import flash.display.Sprite

  public class View extends Sprite
  {
    internal static const GAP:Number = 5;
    internal static const COLOR_BLACK:Number = 0;
    internal static const COLOR_WITHE:Number = 0xFFFFFF;
    internal static const COLOR_LIGHT_GRAY:Number = 0x999999;
    internal static const COLOR_DARK_GRAY:Number = 0x666666;

    private var _contactListPanel:ContactListPanel;
    private var _contactPanel:ContactPanel;
    private var _navegationPanel:NavegationPanel;
    private var _contactListPanelWidth:Number = 130;
    private var _contactListPanelHeight:Number = 108;
    private var _contactPanelWidth:Number = 235;
    private var _contactPanelHeight:Number = 108;

    /**
     * Constructor de la clase.
     */
    public function View()
    {
      initialize();
    }
```

```
    /**
     * Inicializa el objeto.
     */
    private function initialize():void
    {
      buildPanels();
      displayData();
    }

    /**
     * Arma los paneles.
     */
    private function buildPanels():void
    {
      //Creamos los paneles.
       contactListPanel = new ContactList-
Panel(_contactListPanelWidth,

_contactListPanelHeight);
      _contactPanel = new ContactPanel(_contactPanelWidth,
                                       _contactPanelHeight);
      _navegationPanel = new NavegationPanel();
      //Configuramos algunas de sus propiedades.
      _contactPanel.x = _contactListPanel.width + GAP;
      _navegationPanel.y = _contactPanel.y + _contactPanel.height;
      _navegationPanel.x = _contactPanel.x +
          (_contactPanel.width - _navegationPanel.width) / 2;
      //Los agregamos.
      addChild(_contactListPanel);
      addChild(_contactPanel);
      addChild(_navegationPanel);
    }

    /**
     * Método para mostrar los datos.
     */
```

```
    private function displayData():void
    {
      //Obtenidos los datos de prueba
      var dummy:Dummy = new Dummy();
      var arrayData:Array = dummy.arrayData;
      var index:uint = 0;
      var contact:Object = arrayData[index];

      //Probamos:
      //ContactListPanel
      _contactListPanel.displayData(arrayData);
      _contactListPanel.selectByIndex(index);
      //ContactPanel
       contactPanel.displayData(contact.name + " " + con-
tact.lastName,
                                contact.phoneNumber,
                                contact.commentary);
      //NavegationPanel
      _navegationPanel.displayData(index, arrayData.length);
    }

    /**
     * Destruye todos los objetos complejos.
     */
    public function destroyObjects():void
    {
      _contactListPanel.destroyObjects();
      removeChild(_contactListPanel);
      _contactListPanel = null;

      _contactPanel.destroyObjects();
      removeChild(_contactPanel);
      _contactPanel = null;

      _navegationPanel.destroyObjects();
      removeChild(_navegationPanel);
      _navegationPanel = null;
```

```
        }
    }
}
```

Las clases y paquetes quedarían como se muestra a continuación:

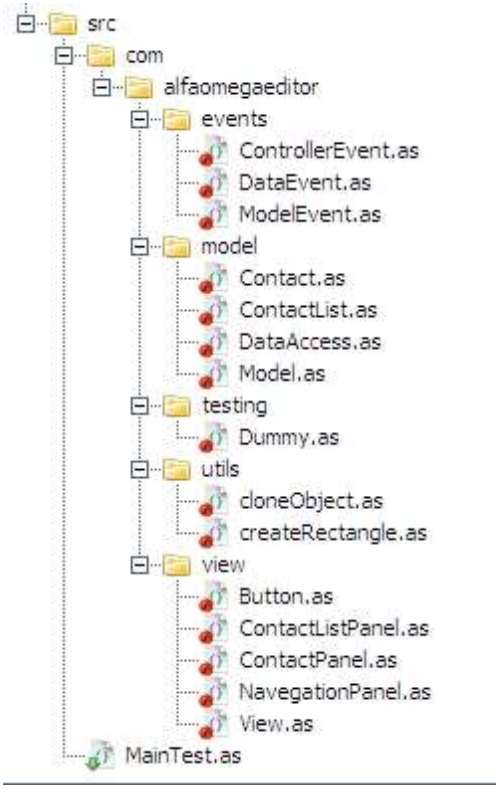

Fig. 11

Y el gráfico que obtendríamos sería algo muy similar a lo que se muestra a la siguiente figura:

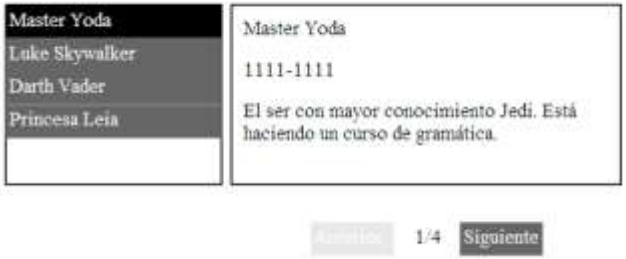

Fig. 12

View: envío de eventos

Ya tenemos la representación resuelta, ahora nos queda tomar los eventos de los paneles. Entonces, suscribiremos el `View` a los eventos de botón de `ConctactListPanel` y de `NavegationPanel`.

Los eventos de estos dos paneles implican un pedido de índice en el Modelo. Hay que destacar que el clic de un botón en `View` no implica su cambio inmediato, sino que este cambio está sujeto a la actualización del Modelo. Un ejemplo general del circuito de un evento iniciado en alguno de los botones del panel es el siguiente:

1) Se hace clic en un botón y el `View` recibe un `MouseEvent`, que lo traduce en un pedido de cambio de índice.
2) El `View` despacha el evento en el cual están los datos del cambio que se está requiriendo en un evento del tipo `ViewEvent`.
3) El `Controller` recibe el `ViewEvent`, lo traduce a un `ControllerEvent` y lo despacha.
4) El `Model` recibe el `ControllerEvent` y en función del evento, hace el llamado directo a los métodos de los objetos correspondientes.

Veamos un diagrama de este circuito en la siguiente figura:

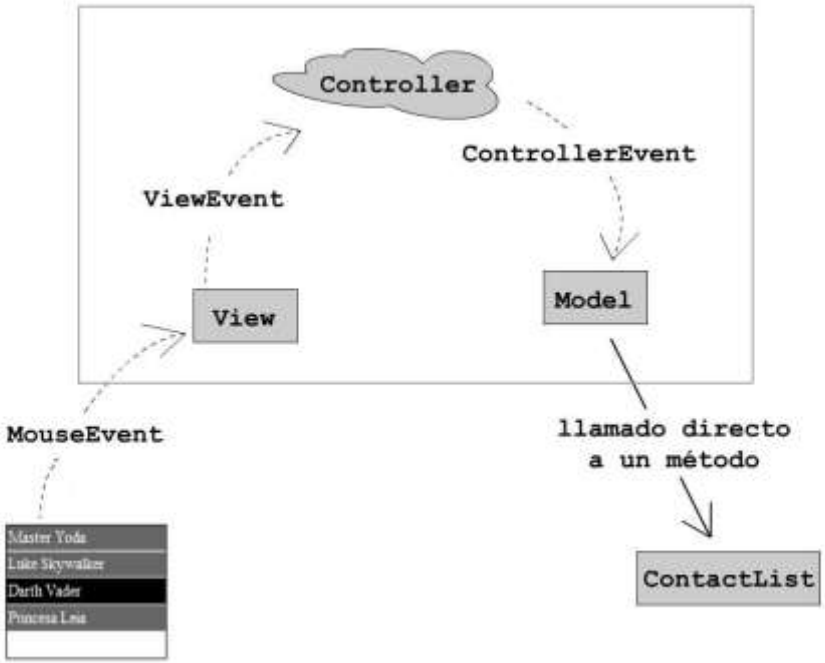

Fig. 13

Valiéndonos del mecanismo de eventos de ActionScript 3.0, no será que el `View`, se suscriba a "cada botón". El `View`, suscribirá un escucha a cada panel. Al recibir el evento, el `View` de acuerdo con el `target` que traiga el evento, traducirá el evento de `MouseEvent` a un evento `ViewEvent`, y lo despachará. Y hasta allí llegarán las "preocupaciones" del `View`. Entonces, ahora nos concentraremos en el `ViewEvent`.

Implementado el ViewEvent

Esta clase tiene tres tipos de eventos: ASK_PREV_INDEX (pedido de índice anterior),

ASK_NEXT_INDEX (pedido de índice siguiente), ASK_INDEX (pedido de un índice determinado). La única propiedad del objeto `ViewEvent` es `index`. Vale decir que para nuestra aplicación y para las historias de usuario que tenemos hasta ahora, cualquier evento que dispare un botón se traducirá un evento `ViewEvent`.

La clase quedaría así:

```
package com.alfaomegaeditor.events
{
  import com.alfaomegaeditor.utils.cloneObject;

  import flash.events.Event;

  public class ViewEvent extends Event
  {
    public static const ASK_PREV_INDEX:String = "askPrevChange";
    public static const ASK_NEXT_INDEX:String = "askNextIndex";
    public static const ASK_INDEX:String = "askIndex";

    private var _index:uint;

    public function ViewEvent(type:String, index:uint=0)
    {
      super(type);
      _index = index;
    }

    public function get index():uint
    {
      return _index;
    }

    override public function clone():Event
    {
      var event:ViewEvent = ViewEvent(cloneObject(this));
      return event;
    }
  }
}
```

Y ahora implementaremos los manejadores de evento (“traductores”) en `View`.

Haremos un método que se encargue de agregar los escuchas del `View` a los `MouseEvent` de los paneles. Veremos cómo ve el método, y luego mostraremos la clase entera:

```
/**
 * Se configuran los escuchas a los paneles.
 */
private function setEventListeners():void
{
  var useCapture:Boolean = false;
  var priority:uint = 0;
  var useWeakReference:Boolean = true;

  //Tomamos los eventos de botón de, contacListPanel
  _contactListPanel.addEventListener(MouseEvent.CLICK,
                                     onClickContactListPanel,
                                     useCapture,
                                     priority,
                                     useWeakReference);

  _navegationPanel.addEventListener(MouseEvent.CLICK,
                                    onClickNavegationPanel,
                                    useCapture,
                                    priority,
                                    useWeakReference);
}
```

Haremos un método que cree y despache eventos del tipo: `EventView`.

```
/**
 * Crea y envía eventos del tipo: ViewEvent
 * @param type: nombre del evento.
 * @param index: un índice determinado, por defecto es el 0.
 */
```

```
private function createAndDispatchEvent(type:String, in-
dex:uint=0):void
{
  var event:ViewEvent = new ViewEvent(type, index);
  dispatchEvent(event);
}
```

Los dos manejadores de evento son: `onClickContactListPanel` y `onClickNavegationPanel`

Veamos `onClickContactListPanel`, simplemente obtenemos el índice del botón haciéndole clic, y lo usamos para solicitar el cambio a ese mismo índice.

```
/**
 * Manejador del evento: MouseEvent.CLICK del _contactListPanel.
 * @param event: evento que envía ActionScript.
 */
 private function onClickContactListPanel(event:MouseEvent):void
 {
   //Obtenemos el índice del botón que corresponde al índice del
contacto.
   var index:uint =
event.target.parent.getChildIndex(event.target);
   //despachaos el evento de pedido de cambio de índice.
   createAndDispatchEvent(ViewEvent.ASK_INDEX, index);
 }
```

Deberemos agregar el llamado a este método en: `initialize`.

Una técnica muy similar a la detallada a continuación, se utiliza en videojuegos triple A para los manejos de evento de los botones. Entonces, para el caso de los eventos del `NavegationPanel`, identificamos al botón por el nombre, y dependiendo de cuál sea, llamamos a uno u otro evento. Éstos son pasados a un método que identifica de qué botón se trata, y luego se tiene un `switch`, donde se ejecutan métodos de acuerdo con el caso.

Una manera más avanzada de manejarlo, podría ser usando un objeto que tenga como claves los nombres de los botones y como valor un Comando o método a ejecutar. En la práctica, con muchas personas de diferentes niveles de conocimiento tocando el código, resulta más legible este modo. Es una alternativa más de

tantas y para este caso puede resultar más claro ver en un mismo método las opciones, dependiendo del botón al que se le haga clic.

```
/**
 * Manejador del evento: MouseEvent.CLICK del _navegationPanel.
 * @param event: evento que envía ActionScript.
 */
private function onClickNavegationPanel(event:MouseEvent):void
{
  //Basados en el nombre del botón despachamos uno u otro evento.
  var buttonName:String = event.target.name;
  switch(buttonName)
  {
    case NavegationPanel.PREV:
      createAndDispatchEvent(ViewEvent.ASK_PREV_INDEX);
      break;
    case NavegationPanel.NEXT:
      createAndDispatchEvent(ViewEvent.ASK_NEXT_INDEX);
      break;
    default:
      //En caso de recibir un evento no esperado, detenemos la
aplicación.
      var errorString:String = "No existe evento para el botón: " +
buttonName;
      throwError("View", "onClickNextPrevButton", errorString);
  }
}
```

Ahora bien, debemos resaltar que hemos agregado una excepción de tal manera que se detenga la aplicación si el `switch` recibe un dato no válido. Utilizamos una función que está en el paquete `utils`. Lo hacemos de esta manera para tener la seguridad de que detectaremos un uso indebido.

La implementación de la función es la siguiente:

```
package com.alfaomegaeditor.utils
{
  import flash.display.DisplayObject;
```

```
  import flash.utils.getQualifiedClassName;


  /**
   * Esta función detiene la aplicación mostrando un mensaje de
error.
   * @param sourceName: nombre completo de la clase u objeto.
   * @param methodName: el nombre del método en el cual se produce
el error.
   * @param errorDescription: descripción del error.
   */
  public function throwError(sourceName:String,
                             methodName:String,
                             errorDescription:String):void
  {
    throw new Error(sourceName + " ::> " + methodName + ": " + er-
rorDescription);
  }
}
```

View: recepción de eventos

Habrá un solo evento que recibirá el `View` y es el cambio de índice. Este evento mantendrá consistente la capa Vista con el índice selecto. Separaremos del método `displayData`, lo concerniente a mostrar el contacto seleccionado.

El método `displayData` recibirá el parámetro `arrayData` del tipo `Array`, sobre la base del cual se desplegarán los botones. Y por otro lado, tendremos el manejador del evento `onChangIndex`, que actualizará los paneles cada vez que cambie un índice.

Refinando la clase View

Importamos las clases necesarias e implementamos los ajustes de `displayData` y del manejador de eventos `onChangeIndex`. También removemos el llamado a `displayData` y el uso del objeto `Dummy`. Dejamos la clase preparada para ser integrada a la aplicación.

```
package com.alfaomegaeditor.view
{
  /**
   * La responsabilidad de esta clase es: ser el Facade de la capa:
Vista.
   * Crea y destruye los objetos. Es el punto de comunicación entre
esta capa
   * y otras entidades.
   */

  import com.alfaomegaeditor.utils.throwError;
  import com.alfaomegaeditor.events.ViewEvent;
  import com.alfaomegaeditor.events.ControllerEvent;

  import flash.display.Sprite
  import flash.events.MouseEvent;

  public class View extends Sprite
  {
    internal static const GAP:Number = 5;
    internal static const COLOR_BLACK:Number = 0;
    internal static const COLOR_WITHE:Number = 0xFFFFFF;
    internal static const COLOR_LIGHT_GRAY:Number = 0x999999;
    internal static const COLOR_DARK_GRAY:Number = 0x666666;

    private var _contactListPanel:ContactListPanel;
    private var _contactPanel:ContactPanel;
    private var _navegationPanel:NavegationPanel;
    private var _contactListPanelWidth:Number = 130;
    private var _contactListPanelHeight:Number = 108;
    private var _contactPanelWidth:Number = 235;
    private var _contactPanelHeight:Number = 108;

    /**
     * Constructor de la clase.
     */
    public function View()
```

```
    {
      initialize();
    }

    /**
     * Método para mostrar los datos.
     */
    public function displayData(arrayData:Array):void
    {
      //Desplegamos los botones en la lista de contactos basados en
arrayData.
      _contactListPanel.displayData(arrayData);
    }

    /**
     * Manejador del evento: ControllerEvent.UPDATE_INDEX
     * Una vez recibido el evento muestra los datos correspondien-
tes
     * en cada panel.
     * @param event: el evento que enviará el documento de la cla-
se.
     */
    public function onChangeIndex(event:ControllerEvent):void
    {
      //Mostramos la selección de botón.
      var contactIndex:uint = event.contactIndex;
      _contactListPanel.selectByIndex(contactIndex);
      //Mostramos los datos en el panel de contacto.
      var fullName:String = event.fullName;
      var phoneNumber:String = event.phoneNumber;
      var commentary:String = event.commentary;
      _contactPanel.displayData(fullName, phoneNumber, commentary);
      //Mostramos los datos necesarios en el panel de navegación.
       navegationPanel.displayData(contactIndex,
event.totalContacts);
    }

    /**
```

```
     * Inicializa el objeto.
     */
    private function initialize():void
    {
      buildPanels();
      setEventListeners();
    }

    /**
     * Arma los paneles.
     */
    private function buildPanels():void
    {
      //Creamos los paneles.
       contactListPanel = new ContactList-
Panel(_contactListPanelWidth,

_contactListPanelHeight);
      _contactPanel = new ContactPanel(_contactPanelWidth,
                                       _contactPanelHeight);
      _navegationPanel = new NavegationPanel();
      //Configuramos algunas de sus propiedades.
      _contactPanel.x = _contactListPanel.width + GAP;
      _navegationPanel.y = _contactPanel.y + _contactPanel.height;
      _navegationPanel.x = _contactPanel.x +
          (_contactPanel.width - _navegationPanel.width) / 2;
      //Los agregamos.
      addChild(_contactListPanel);
      addChild(_contactPanel);
      addChild(_navegationPanel);
    }

    /**
     * Se configuran los escuchas a los paneles.
     */
    private function setEventListeners():void
    {
```

```
      var useCapture:Boolean = false;
      var priority:uint = 0;
      var useWeakReference:Boolean = true;

      //Tomamos los eventos de botón de, contacListPanel
      _contactListPanel.addEventListener(MouseEvent.CLICK,
                                         onClickContactListPanel,
                                         useCapture,
                                         priority,
                                         useWeakReference);

      _navegationPanel.addEventListener(MouseEvent.CLICK,
                                        onClickNavegationPanel,
                                        useCapture,
                                        priority,
                                        useWeakReference);
    }

    /**
     * Manejador del evento: MouseEvent.CLICK del
_contactListPanel.
     * @param event: evento que envía ActionScript.
     */
    private function onClickContactListPanel(event:MouseEvent):void
    {
      //Obtenemos el índice del botón que corresponde al índice del
contacto.
      var index:uint =
event.target.parent.getChildIndex(event.target);
      //despachaos el evento de pedido de cambio de índice.
      createAndDispatchEvent(ViewEvent.ASK_INDEX, index);
    }

    /**
     * Manejador del evento: MouseEvent.CLICK del _navegationPanel.
     * @param event: evento que envía ActionScript.
     */
    private function onClickNavegationPanel(event:MouseEvent):void
```

```
    {
      //Basados en el nombre del botón despachamos uno u otro even-
to.
      var buttonName:String = event.target.name;

      switch(buttonName)
      {
        case NavegationPanel.PREV:
          createAndDispatchEvent(ViewEvent.ASK_PREV_INDEX);
          break;
        case NavegationPanel.NEXT:
          createAndDispatchEvent(ViewEvent.ASK_NEXT_INDEX);
          break;
        default:
          //En caso de recibir un evento no esperado, detenemos la
aplicación.
          var errorString:String = "No existe evento para el botón:
" + buttonName;
          throwError("View", "onClickNextPrevButton", errorString);
      }
    }

    /**
     * Crea y envía eventos del tipo: ViewEvent
     * @param type: nombre del evento.
     * @param index: un índice determinado, por defecto es el 0.
     */
    private function createAndDispatchEvent(type:String, in-
dex:uint=0):void
    {
      var event:ViewEvent = new ViewEvent(type, index);
      dispatchEvent(event);
    }

    /**
     * Destruye todos los objetos complejos.
     */
    public function destroyObjects():void
```

```
    {
      //Destruimos:
      //El panel con la lista de contactos.
      _contactListPanel.destroyObjects();
      removeChild(_contactListPanel);
      _contactListPanel = null;

      //El panel. de contacto.
      _contactPanel.destroyObjects();
      removeChild(_contactPanel);
      _contactPanel = null;

      //El panel de navegación.
      _navegationPanel.destroyObjects();
      removeChild(_navegationPanel);
      _navegationPanel = null;
    }
  }
}
```

Nuestras clases y paquetes quedarían como se muestra a continuación:

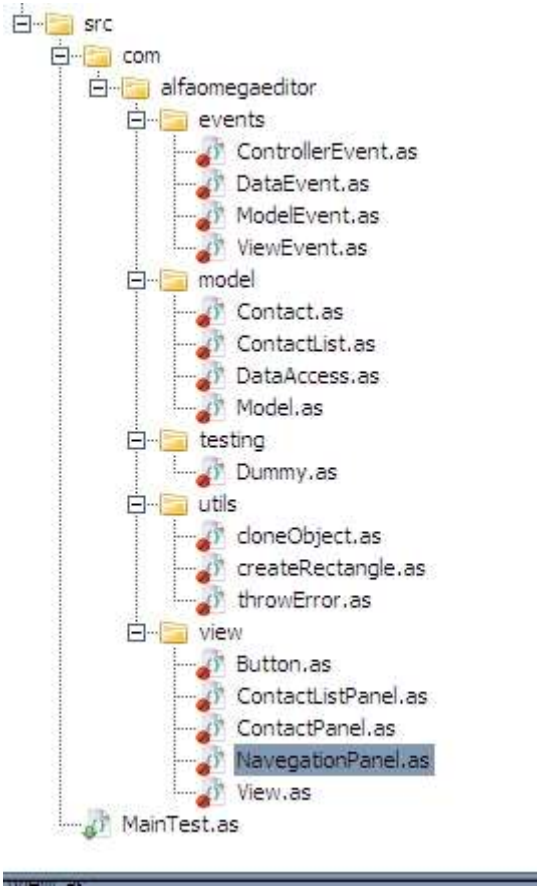

Fig. 14

Lo que aquí vemos es que más allá de tener las clases desacopladas, esta forma de trabajo permite que fraccionemos toda una aplicación en partes más pequeñas y controlables. Trabajamos en pequeñas fracciones de la aplicación sin temer por los errores que se propaguen por toda la aplicación. Vemos que en el caso en que haya alguna falla, la estructura de la aplicación permite trabajar en forma acotada. Y una falla se mantendrá también en un área acotada de la aplicación, y su arreglo no traería riesgos de propagación de errores o equivocaciones colaterales.

Implementando la capa del Controlador

Esta capa es la que define la manera en la que se comunicarán el Modelo y la Vista. En nuestro caso, como estamos realizando una aplicación sencilla que no implica mayor complejidad en el manejo de las acciones del usuario, utilizaremos una sola clase denominada: `Controller`. Eventualmente, se podría usar una clase `Controller` que funcione como un manejador de otros controladores. Su lógica podría implicar el cambio de estados o propagar eventos que provoquen reconfiguraciones de la aplicación, cambios de pantalla o estados, que a su vez podrían disparar un evento para que el controlador sea cambiado, o reaccionen distintos controladores de acuerdo con el caso.

La clase Controller

Esta clase es la que actuará como el `EventAggregator`. Esta clase recibirá eventos del tipo: `ModelEvent` y `ViewEvent`, y los traducirá en eventos del tipo `ControllerEvent` sin utilizar una lógica intermedia. Eventualmente, se podría implementar otra clase, manejando cierta lógica del controlador, sin necesidad de cambiar el `Model` o la clase `View`.

Controller: envío de eventos.

La clase `Controller` traducirá los eventos que reciba a un evento del tipo `ControllerEvent` y lo despachará. El método encargado de crear el evento y despacharlo es: `createAndDispatchEvent`. Este método recibe como parámetro el nombre del evento y el índice recibido.

Controller: recepción de eventos.

Los manejadores de eventos que los reciben, llamarán al método `createAndDispatchEvent`, pasándole el nombre del evento a despachar y el índice que hayan recibido.

Eventos de la clase: `ModelEvent`

El método encargado de manejar este tipo de evento es `onChangeIndex`. El nombre del evento que despachará es: `ControllerEvent.UPDATE_INDEX`

Eventos de la clase: `ViewEvent`

Tres manejadores de evento recibirán los pedidos de cambio para índice anterior, índice siguiente, y un índice determinado. A continuación, veremos qué nombre de evento pasará cada manejador de evento:

`onAskPrevIndex` pasará: `ControllerEvent.ASK_PREV_INDEX`
`onAskNextIndex` pasará: `ControllerEvent.ASK_NEXT_INDEX`
`onAskIndex` pasará: `ControllerEvent.ASK_INDEX`

Implementación de la clase Controller

La clase `Controller` quedaría como se muestra a continuación. Es interesante observar que no tiene el método `destroyObjects`, porque de momento no tiene objetos complejos. Los objetos complejos que utiliza son variables temporales.

```
package com.alfaomegaeditor.controller
{

  import com.alfaomegaeditor.events.DataEvent;
  import com.alfaomegaeditor.events.ModelEvent;
  import com.alfaomegaeditor.events.ViewEvent
  import com.alfaomegaeditor.events.ControllerEvent;
  import com.alfaomegaeditor.model.Contact;

  import flash.events.MouseEvent;
  import flash.events.EventDispatcher;

  import flash.utils.Dictionary;
  import flash.display.Sprite;

  public class Controller extends Sprite
  {
    /**
     * Constructor de la clase
     */
    public function Controller()
```

```
    {
    }

    /**
     * Manejador de evento al cambiar el index de contactList.
     */
    public function onChangeIndex(event:ModelEvent):void
    {
      var type:String = ControllerEvent.UPDATE_INDEX;
      var contactIndex:uint = event.contactIndex;
      var totalContacts:uint = event.totalContacts;
      var contact:Contact = event.contact;
      var fullName:String = contact.name + " " + contact.lastName;
      var phoneNumber:String = contact.phoneNumber;
      var commentary:String = contact.commentary;

      createAndDispatchEvent(type,
                             contactIndex,
                             totalContacts,
                             fullName,
                             phoneNumber,
                             commentary);
    }

    /**
     * Manejador de evento de los botones del view.
     */
    public function onAskPrevIndex(event:ViewEvent):void
    {
      createAndDispatchEvent(ControllerEvent.ASK PREV INDEX,
event.index);
    }
    public function onAskNextIndex(event:ViewEvent):void
    {
      createAndDispatchEvent(ControllerEvent.ASK NEXT INDEX,
event.index);
    }
```

```
    public function onAskIndex(event:ViewEvent):void
    {
      createAndDispatchEvent(ControllerEvent.ASK INDEX,
event.index);
    }

    /**
     * Despacha el evento MediatorEvent
     */
    private function createAndDispatchEvent(type:String,
                                            contactIndex:uint=0,
                                            totalContacts:uint=0,
                                            fullName:String="",
                                            phoneNumber:String="",
                                            commen-
tary:String=""):void
    {
      var event:ControllerEvent = new ControllerEvent(type,
                                                      contactIndex,
                                                      totalCon-
tacts,
                                                      fullName,
                                                      phoneNumber,
                                                      commentary);
      dispatchEvent(event);
    }
  }
}
```

Integración del Model, View y Controller

Tenemos las tres capas funcionando independientemente. Las tres saben cómo reaccionar a los eventos. Ahora, lo que haremos es interconectar a las clases entre sí. ¿Debería funcionar todo, entonces? Por supuesto que sí. Pero, no debemos olvi-

dar ninguna relación, es decir, no debemos olvidar agregar los escuchas correspondientes. Entonces, en la clase del documento, hace lo siguiente:

1. Instanciar y destruir los objetos.
2. Configuraremos los escuchas.
3. Cargar los datos.
4. Agregar a la instancia de `View` al `display list`.

Con esto realizado, la aplicación estará en funcionamiento.

La clase del documento instancia los tres representantes de cada capa (`Model`, `View` y `Controller`), y los interrelaciona entre sí. El máximo acoplamiento se da en esta clase, ya que conoce a las otras tres. Sin embargo, ninguna de las tres clases se conoce entre sí. Y dentro de las tres clases capa (`Model`, `View` y `Controller`), el máximo acoplamiento está dado en la clase `Controller` con las clases de los eventos.

Separaremos en tres métodos el agregado de escuchas.

La clase `MainTest` quedará como sigue:

```
package
{
  /**
   * Punto de entrada de la aplicación.
   * Su responsabilidad es manejar los objetos del tipo:
   * Model, View y Controller.
   * Esto implica, crearlos, eventualmente inicializarlos y destruirlos.
   */
  import com.alfaomegaeditor.events.DataEvent;
  import com.alfaomegaeditor.events.ModelEvent;
  import com.alfaomegaeditor.events.ViewEvent;
  import com.alfaomegaeditor.events.ControllerEvent;

  import com.alfaomegaeditor.controller.Controller;
  import com.alfaomegaeditor.model.Model;
  import com.alfaomegaeditor.view.View;

  import flash.display.Sprite;
```

```
public class MainTest extends Sprite
{
  private var _model:Model;
  private var _view:View;
  private var _controller:Controller;

  /**
   * Constructor de la clase
   */
  public function MainTest()
  {
    initialize();
  }

  /**
   * Inicializa la clase.
   */
  private function initialize():void
  {
    //Crea los objetos.
    createObjects();
    //Configura los escuchas.
    setEventListeners();
    //Carga los datos.
    _model.loadData();
    //Agrega los objetos visuales.
    addChild(_view);
  }

  /**
   * Crea los objetos necesarios.
   */
  private function createObjects():void
  {
    _model = new Model();
    _view = new View();
    _controller = new Controller();
```

```
    }

    /**
     * Configura todos los escuchas del Model, View y Controller.
     */
    private function setEventListeners():void
    {
      addModelEventListeners();
      addViewEventListeners();
      addControllerEventListeners();
    }

    /**
     * Configura los escuchas del Model.
     */
    private function addModelEventListeners():void
    {
      //Agregamos un escucha de:
      //El Controller ante el cambio de índice.
      _model.addEventListener(ControllerEvent.UPDATE_INDEX,
                              _controller.onChangeIndex);
      //Clase del documento para cuando los datos estén disponi-
bles.
      _model.addEventListener(DataEvent.DATA_READY, onDataReady);
    }

    /**
     * Configura los escuchas del View.
     */
    private function addViewEventListeners():void
    {
      var useCapture:Boolean = false;
      var priority:uint = 0;
      var useWeakReference:Boolean = true;

      //Se agregan escuchas del Controller para cuando:
      //Se pide el índice siguiente.
```

```
      _view.addEventListener(ViewEvent.ASK_PREV_INDEX,
                             _controller.onAskPrevIndex,
                             useCapture,
                             priority,
                             useWeakReference);
      //Se pide el índice anterior.
     _view.addEventListener(ViewEvent.ASK_NEXT_INDEX,
                             _controller.onAskNextIndex,
                             useCapture,
                             priority,
                             useWeakReference);
     //Se pide un índice determinado.
      _view.addEventListener(ViewEvent.ASK_INDEX,
                             _controller.onAskIndex,
                             useCapture,
                             priority,
                             useWeakReference);
    }

    /**
     * Configura los escuchas del Controller.
     */
    private function addControllerEventListeners():void
    {
      var useCapture:Boolean = false;
      var priority:uint = 0;
      var useWeakReference:Boolean = true;

      //Escuchas del Model:
      //Pedido de cambio de índice.
      _controller.addEventListener(ControllerEvent.ASK_INDEX,
                                   _model.onAskChangeIndex,
                                   useCapture,
                                   priority,
                                   useWeakReference);
      //Pedido del índice previo.
      _controller.addEventListener(ControllerEvent.ASK_PREV_INDEX,
```

```
                                _model.onAskPrevtIndex,
                                useCapture,
                                priority,
                                useWeakReference);
    //Pedido del índice próximo.
    _controller.addEventListener(ControllerEvent.ASK_NEXT_INDEX,
                                _model.onAskNextIndex,
                                useCapture,
                                priority,
                                useWeakReference);

    //Escucha del View:
    //Cambio de índice.
    _controller.addEventListener(ControllerEvent.UPDATE_INDEX,
                                _view.onChangeIndex,
                                useCapture,
                                priority,
                                useWeakReference);
  }

  /**
   * Cuando se cargan los datos, despliega los botones.
   *
   * @param event
   */
  private function onDataReady(event:DataEvent):void
  {
    _view.displayData(event.arrayData);
  }

  /**
   * Destruye todos los objetos complejos.
   */
  public function destroyObjects():void
  {
    _model.destroyObjects();
```

```
            _model = null;

            _view.destroyObjects();
            _view = null;

            _controller = null;
        }
    }
}
```

Compilamos, probamos nuestra aplicación y constatamos que hemos cubierto todas las historias de usuario:

1. La aplicación cargará los datos de los contactos de una fuente determinada.
2. Al cargarse los datos:
    a. El sistema desplegará en el "Panel lista de contactos", botones con el nombre y apellido de contacto.
    b. Se seleccionará el primer contacto en forma automática.
3. Se selecciona el contacto desde el "Panel lista de Contactos". Al hacer clic sobre algunos de los contactos, se seleccionará el contacto respectivo.
4. Se selecciona el contacto desde "Panel de navegación". Al hacer clic sobre algunos de los botones: "anterior" o "siguiente", se seleccionará el contacto respectivo.
5. Al seleccionar un contacto:
    a. En el "Panel lista de contactos", se marcará el contacto seleccionado.
    b. En el "Panel de Contacto", se mostrará el nombre y apellido, número de teléfono y comentario correspondiente al contacto seleccionado.
    c. En el "Panel de navegación", estando en el primer contacto seleccionado, se deshabilitará el botón con el texto "anterior".
    d. En el "Panel de navegación", estando el último contacto seleccionado, se deshabilitará el botón con el texto "siguiente".
    e. En el "Panel de navegación", se mostrará el índice actual y la cantidad total de contactos.