

Cycleframework

DESCRIPCIÓN CycleFramework permite a los desarrolladores implementar interfaces gráficas para aplicaciones de forma rápida y sencilla usando tecnología AS3. Dentro de estas se muestra especialmente útil en el subconjunto de aplicaciones de gestión, donde la segmentación de estados de vista es clara.

CycleFramework es un conjunto de implementaciones de patrones de diseño que proveen de una metodología de trabajo en el desarrollo, que incrementa la productividad y sobre todo facilita y aclara la relación entre los componentes visuales que conforman las aplicaciones.

De todos los patrones que se aplican, quizás los más importantes sean los que están relacionados con las buenas prácticas en POO y que no requieren de implementación, tales como:

Patrón experto en información

Asigne una responsabilidad al experto en información. La clase que tiene la información necesaria para llevar a cabo la responsabilidad es la que debe realizar la acción.

Patrón no hables con extraños (Ley de Demeter)

Para un método **m** de una clase **o** sólo deberían invocarse métodos de estos tipos de objetos:

- del propio objeto **o**
- de los parámetros que recibe el propio método **m**
- de cualquier objeto que instancie el propio método **m**
- de cualquier atributo de **o**

Patrón cadena de responsabilidades

Evite el acoplamiento del emisor del requerimiento y el receptor, al obtener más de un objeto con posibilidad de tratar el requerimiento. Es la cadena de objetos que reciben y pasan la petición a lo largo de la cadena hasta que el objeto experto en el requerimiento la manipule.

Con lo que juntando los tres patrones y una estructura clásica de aplicación se dispone, en cada estado de vista, de lo siguiente:

- **Parte fija:** Componentes visuales que no cambian dentro de un bloque conformado por varias vistas. Ejemplo de esto sería el menú principal de una aplicación con interfaz gráfica.
- **Parte variable:** Zona dentro de un componente visual que puede cambiar en el tiempo.

Navegación basada en CycleFramework

El siguiente árbol representa los diferentes estados que conforma una aplicación cualquiera.

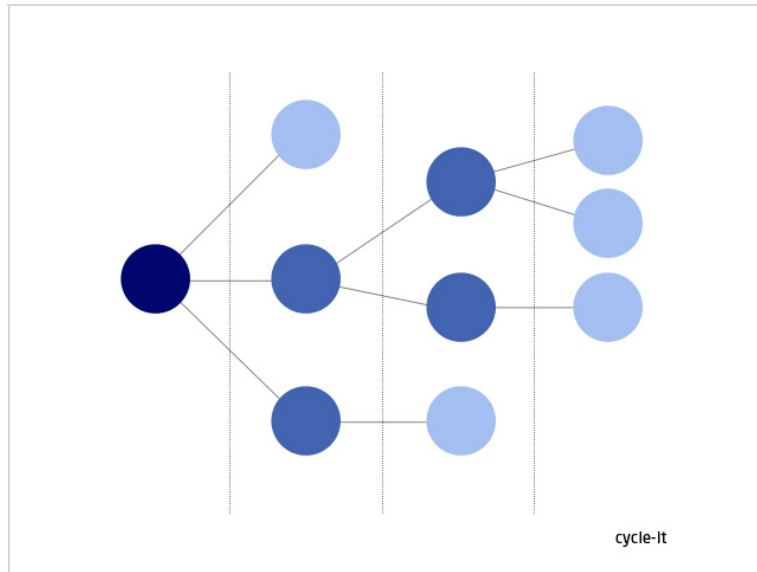


Figura 1.1

El primer nodo de la aplicación, mostrado en color azul oscuro, representa al *gestor de vistas* de la aplicación. Este nodo se implementará con una clase creada específicamente con ese propósito, siendo posible hacer que `Application` implemente una interfaz determinada (interfaz que se tratará más adelante en este documento).

Los nodos con un tono azul medio representan ramas dentro del árbol de navegación. Estas ramas deberán implementar también el interfaz mencionado anteriormente.

Las *hojas* representadas en un color azul claro indican vistas finales. Estas vistas no poseen parte variable y por lo tanto no será necesario implementar la interfaz mencionada anteriormente, necesaria para que CycleFramework gestione todos los posibles cambios de vista en la aplicación.

Teniendo en cuenta una posible navegación (indicado en naranja) hacia un estado de vista en particular, el árbol de navegación quedaría de la siguiente forma:

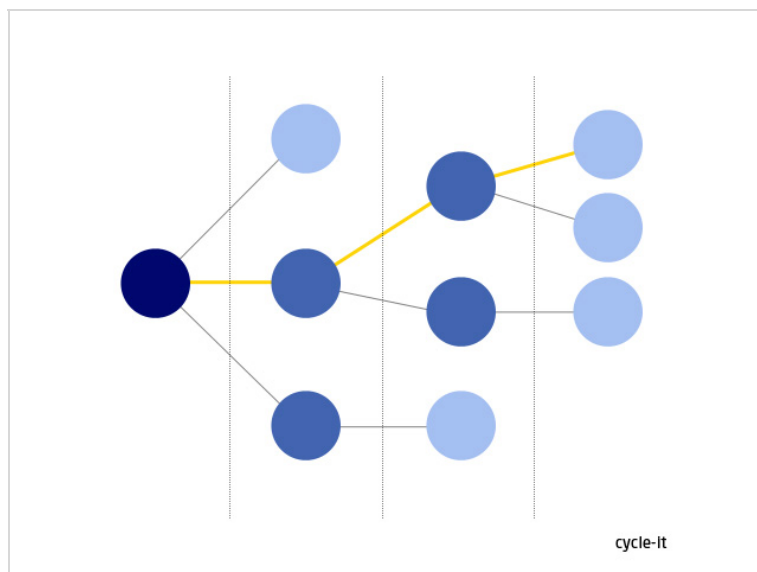


Figura 1.2

En la siguiente imagen se muestra la anterior navegación aplicada sobre la interfaz gráfica:

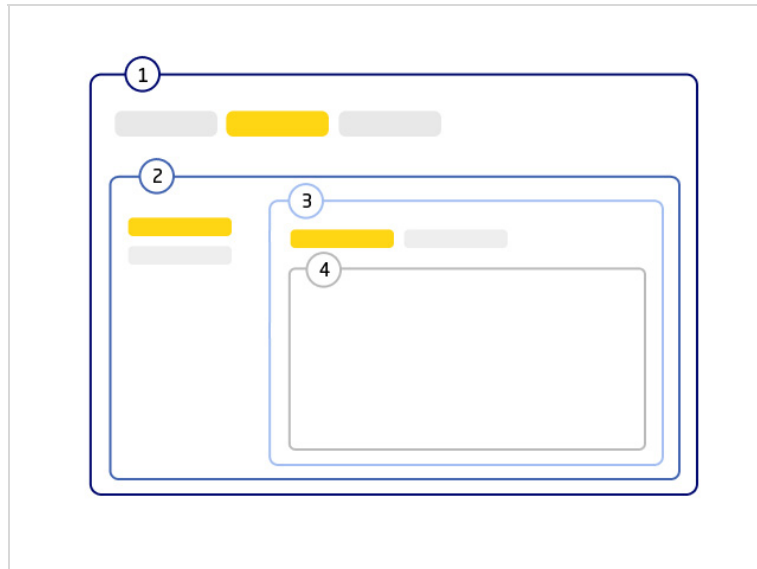


Figura 1.3

En la figura 1.3 aparece representada la aplicación después de haber realizado una navegación; la **Zona 1** conforma el menú de la aplicación que será visible durante todo el ciclo de vida de esta, siendo la parte fija del primer nivel de vista. A su vez, la **Zona 2** representa la parte variable de la Zona 1 que contiene igualmente un menú, que constituye su parte fija. Así sucesivamente se irá conformando el estado actual de la aplicación representada.

Un ejemplo descriptivo de este caso sería una gestión de usuarios clásica en la que se visualiza una foto del usuario con un menú para consultar sus datos, modificarlos o eliminarlos. Este menú constituirá la parte fija y los distintos formularios, representados en la figura como las **Zonas 3 y 4**, serán la parte variable.

Teniendo en cuenta los patrones de diseño anteriores, estos se aplicarán para que cada vista sepa y sea responsable de pintar y gestionar su parte fija. Así mismo, al conocer sus partes variables, cada vista será responsable de crear y añadir las distintas vistas dentro de su región de trabajo o zona variable. Sucesivamente, cada una de ellas será responsable de pintar su parte fija, y saber qué tiene que pintar en su zona variable dependiendo del estado de la aplicación.

Existe un caso especial en los componentes visuales que no tienen parte variable; que son hojas en el árbol de navegación en cuyo interior no cambiará la información a lo largo del ciclo de vida del componente.

¿Cómo se consigue esto? Para que CycleFramework gestione las distintas vistas de una aplicación, estas deben implementar la interfaz ***INavegableView*** en el caso de ser ramas en el árbol de navegación (en caso de ser hoja del árbol no se requerirá implementación de dicha interfaz).

```
package es.cycle.interfaces {
    import es.cycle.vo.NavigationView;
    import mx.core.Container;
    import mx.events.PropertyChangeEvent;

    public interface INavegableView {
        function navigateTo(nextState:String):void;
        function get mainContainer():Container;
        function get navigationView():NavigationView;
    }
}
```

Como se puede observar, los requisitos no son muchos. Sólo habrá que implementar tres métodos:

navigateTo(nextState:String)

Como parámetro se introducirá un identificador de estado de vista. Este método decidirá que vista (que a su vez implemente ***INavegableView***) se añadirá en función del valor de **nextState**. Una vez decidido se usará **mainContainer** para ubicar esta nueva vista.

get mainContainer()

Asegura que todo componente visual rama que participa en el framework, tiene un contenedor que delimita que zona dentro de él será la dedicada para almacenar el contenido variable que es capaz de generar. **mainContainer** no debería contener más de un hijo simultáneamente, aunque no es estrictamente obligatorio.

get navigationView()

Esta es la única dependencia real de los componentes visuales con el framework. Por composición usará un objeto **NavigationView**, el cual se encargará de mantener esta vista sincronizada con el motor de CycleFramework.

NavigationView

El objeto **NavigationView** tiene una variable pública llamada **myIndex** que indica el nivel dentro del árbol de navegación del componente visual asociado a dicho objeto.

El constructor recibe como argumento una referencia a la instancia de ***INavegableView*** que lo usa. Opcionalmente puede recibir el parámetro **eraseable** – el cual por defecto tiene valor igual a **false** – que indica si se desea mantener referencia al componente visual, para no tener que instanciarlo cada vez que se invoque el estado de vista correspondiente dentro de **navigateTo**. Si por el contrario se quiere instanciar un componente visual nuevo con cada cambio de estado, el parámetro **eraseable** deberá tener un valor igual a **true**.

A continuación se muestra un ejemplo de una vista que implementa esta interfaz:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml" width="100%" height="100%"
  backgroundColor="#dddddd"
  implements="es.cycle.interfaces.INavigableView">
  <mx:Script>
    <![CDATA[
      import mx.events.FlexEvent;
      import es.cycle.utils.changecontainer.ChangeContainerManager;
      import mx.collections.ArrayCollection;
      import es.cycle.event.ViewStateChangeEvent;
      import mx.core.UIComponent;
      import mx.core.Container;
      import es.cycle.vo.NavigationView;

      private var alta:AltaUsuario;
      private var baja:BajaUsuario;

      private var _navigationView:NavigationView = new NavigationView(this as INavigableView);

      public function get navigationView():NavigationView {
        return _navigationView;
      }

      public function get mainContainer():Container {
        return _mainContainer;
      }

      public function navigateTo(nextState:String):void {
        var aux:UIComponent = null;
        switch (nextState){
          case "alta":
            if (!alta){
              alta = new AltaUsuario();
            }
            aux = alta;
            break;

          case "baja":
            if (!baja){
              baja = new BajaUsuario();
            }
            aux = baja;
            break;

          default:
            mainContainer.removeAllChildren();
            break;
        }
        if (aux != null) {
          mainContainer.removeAllChildren();
          mainContainer.addChild(aux);
        }
      }
    ]]>
  </mx:Script>

  <mx:VBox height="100%" width="120" left="0" verticalAlign="middle" backgroundColor="#cccccc">
    <mx:Button label="Alta" width="100"
      click="new ViewStateChangeEvent(ViewStateChangeEvent.MODIFY_VIEW,new ArrayCollection(['alta']),
        navigationView.myIndex+1).dispatch()" />
    <mx:Button label="Edición" width="100"/>
    <mx:Button label="Baja" width="100"
      click="new ViewStateChangeEvent(ViewStateChangeEvent.MODIFY_VIEW,new ArrayCollection(['baja']),
        navigationView.myIndex+1).dispatch()" />
    <mx:Button label="Consulta" width="100"/>
  </mx:VBox>
  <mx:Canvas id="_mainContainer" width="100%" height="100%" left="120"
    verticalScrollPolicy="off" horizontalScrollPolicy="off"/>
</mx:Canvas>
```

En este componente, que implementa *INavigableView*, se puede observar que la parte fija está constituida por una botonera con acciones de alta y baja. Se utilizará *mainContainer* para añadir bajo demanda, los formularios correspondientes a las acciones anteriores.

El método *navigateTo* añadirá hijos en su *mainContainer* (la zona que puede cambiar en el tiempo dentro de un componente visual), en función del identificador de estado que se reciba como parámetro. Como recomendación, previamente se debe borrar todo el contenido antiguo que ya no se necesite, pintándose así únicamente el nuevo.

Por último, al declarar la variable *_navigationView*, esta se inicializa con una nueva instancia de tal forma que se integra con el motor del framework, pasando como argumento de entrada al constructor la instancia del objeto *INavigationView* que lo utiliza.

El siguiente paso consiste en conocer cómo navegar entre los diferentes estados de vista que conforman el árbol de navegación de la aplicación; usando el evento **ViewStateChangeEvent**. Tomando como ejemplo el código anterior correspondiente a la vista de gestión de usuario, existen dos formas posibles de emplear este evento:

```
<mx:Button label="Usuarios" width="100" left="10" top="10"
click="new ViewStateChangeEvent(ViewStateChangeEvent.MODIFY_VIEW,
new ArrayCollection(['vm','gUsuarios','alta'])).dispatch()"/>
```

Esta primera versión recibe toda la ruta de componentes hacia donde se quiere navegar como secuencia de identificadores de estado (vm → gUsuarios → alta):

- **vm**: pantalla inicial que lo contiene todo o **ViewManager**,
- **gUsuarios**: gestión de usuarios,
- **alta**: formulario de alta de usuario.

```
<mx:Button label="Alta" width="100" left="10" top="10"
click="new ViewStateChangeEvent(ViewStateChangeEvent.MODIFY_VIEW,
new ArrayCollection(['alta']),navigationView.myIndex+1).dispatch()"/>
```

Esta segunda versión recibe una secuencia de identificadores de estado de vista y además un parámetro adicional, indicando el nivel dentro del árbol de navegación a partir del cual quiere comenzarse a modificar el estado de nuestra aplicación:

- **alta**: formulario de alta,
- **myIndex+1**: siendo **myIndex** el nivel en el cual se encuentra **gUsuarios** dentro del árbol de navegación, al sumarle 1 se accederá al siguiente nivel donde se encuentra la vista asociada al estado “alta”.

Al despachar estos eventos, automáticamente el motor de CycleFramework invocará el método **navigateTo** de las correspondientes vistas involucradas en la navegación. Esta navegación viene representada por la secuencia de identificadores fijada como parámetro al evento.

Para facilitar y organizar el desarrollo de las aplicaciones aparecen las siguientes clases e interfaces:

CycleController

Implementa el patrón de diseño **Front Controller**. Se encargará de gestionar todas las invocaciones de comandos que permiten realizar modificaciones en el modelo, sin necesidad de terminar en llamadas a lógica de negocio. Para añadir comandos a gestionar se utilizará el método **addCommand(eventType,ICycleCommand)**. Nótese que este controlador permite más de un comando por tipo de evento.

ICycleCommand

Interfaz que deben implementar todos los comandos que ejecutan lógica para hacer modificaciones en el modelo sin la necesidad de invocar llamadas a lógica de negocio. Para invocarlos es necesario crear un evento que herede de **CycleEvent** y añadirlos al controlador **CycleController**.

ViewController

Controlador de vista que permite observar variables del modelo de la aplicación para asociar los cambios de estas variables con cambios en el estado de la aplicación. Para observar estos cambios de estado **ViewController** provee de un método de uso sencillo, cuya notación es **addWatcher(host,property,navigationList,observerHandler=null)**

- **host**: variable en el modelo que contiene la propiedad a observar,

- **property**: propiedad a observar,
- **navigationList**: ArrayCollection con la secuencia de estados de vista en la que debe quedar posicionada nuestra aplicación dentro del árbol de navegación tras modificarse la propiedad anterior,
- **observerHandler**: función manejadora asociada al evento `PropertyChangeEvent` provocado por el cambio en la propiedad anterior y que se ejecutará en sustitución a la navegación asociada a `navigationList`.

Cómo ejemplo, observar cambios en una variable `usuarioLogado` permite obtener de forma automática una navegación hacia el estado inicial de la aplicación. Opcionalmente, si el mecanismo genérico no es suficiente, se podrá establecer un valor nulo a `navigationList`, creando un manejador a medida para el argumento `observerHandler`, permitiéndose realizar las operaciones pertinentes asociadas al cambio de la propiedad. Por ejemplo, dependiendo del rol del usuario que se ha logado, se abrirá el estado inicial o un formulario de consulta predefinido.

ServiceController

Su filosofía es similar a la del `ViewController`, sólo que en este caso, asociado al cambio de propiedades, se invocará la ejecución de funciones que solicitan datos susceptibles de haber sido modificados de forma externa a la aplicación. Esto facilita una tarea común al desarrollo de toda aplicación: el refresco de listados. El ejemplo clásico se puede encontrar en el alta de un nuevo usuario; una vez dado de alta el mismo es necesario refrescar un listado existente de usuarios. Para ello bastará con añadir un observador a la variable que contiene la información, asociando una función que maneje el refresco del listado.