Objective C: Análisis de los métodos de comunicación de eventos entre objetos

Objective C: Object event communication methods analysis

Óscar Víquez-Acuña[1]
Luis Alonso Vega-Brenes[2]

Fecha de recepción: 19 de mayo del 2014
Fecha de aprobación: 27 de julio del 2014

Víquez-Acuña, O; Vega-Brenes, L. A. Objective C: Análisis de los métodos de comunicación de eventos entre objetos. *Tecnología en Marcha*. Edición Especial Movilidad Estudiantil 2014. Pág 5-13

1 Profesor. Carrera de Ingeniería en Computación, sede San Carlos. Instituto Tecnológico de Costa Rica. Correo electrónico: oviquez@itcr.ac.cr

2 Estudiante. Carrera de Ingeniería en Computación, sede San Carlos. Instituto Tecnológico de Costa Rica. Correo electrónico: lavb91@gmail.com

Palabras clave

Objective C; Desarrollo iOS; Modelo Vista Controlador; Delegados; Notificaciones; Bloques de finalización; Observación; Comunicación de controladores.

Resumen

Este artículo muestra una serie de patrones de programación utilizados para la comunicación de eventos en el lenguaje Objective C. Cada uno de estos patrones es explicado según su funcionalidad, se da un ejemplo conciso de su uso en el desarrollo de software cotidiano y se presenta un listado de ventajas y desventajas con respecto a las características propias del lenguaje.

Al iniciar se retoman algunos conceptos de la programación orientada a objetos, para no perder de vista el tema con el que se está lidiando. Iniciando por algunos de los pilares de la orientación a objetos, se recuerda que se busca tener en mayor o menor medida en el software para que este sea considerado de calidad. Luego se analiza el patrón de modelo vista controlador, muy popular en los últimos años y el cual permite aplicar, como se verá después, los otros patrones de comunicación de eventos.

Los patrones tratados son: delegados, notificaciones, observación y bloques de finalización, por lo que a lo largo de este artículo se discutirá parte de sus usos cotidianos así como sus puntos a favor y contras. El objetivo de este análisis es brindar a cualquier desarrollador un punto de discusión al seleccionar alguno de estos métodos para el desarrollo de una tarea específica.

Keywords

Objective C; iOS Development; Model View Controller, Delegates; Notifications; Completion Blocks; Observation; Controller communication.

Abstract

This article shows a series of programming patterns used in event communication in Objective C. Each one of these patterns is explained according to its functionality, a concise example of its day to day use is given and, optionally, a listing of its advantages and disadvantages regarding its own features is presented.

In the beginning, some object oriented programming concepts are retaken, so that we don't lose sight of the topic we're dealing with. Starting with some of the object orientation pillars, we remember what we seek to have in a greater or minor extent in the software for it to be considered of good quality. Then the model view controller pattern is analyzed, which has been very popular in the last years, and which allows to apply, as it will be shown later, the other event communication patterns.

Mentioned patterns are: delegates, notifications, observation and completion blocks. Each one of these communication methods have its own pros and cons. Throughout this article their everyday uses will be discussed. The objective of this analysis is to offer any developer a discussion point when selecting one of these methods for a specific task.

Introducción

En la programación orientada a objetos, dos de los grandes pilares son la abstracción y el encapsulamiento. Ambos conceptos se aplican para reducir un concepto a una clase definida y discreta, que a la vez muestra únicamente una parte de su composición al resto del sistema. Este comportamiento garantiza una mayor estabilidad y seguridad interna del objeto, pues las interacciones con el mundo externo se mantienen bajo control propio.

Por otro lado, pero en el mismo tema, se encuentra el concepto de cohesión y de acoplamiento. Se considera que ambos deberían tener un nivel muy bajo en todo sistema. Como referencia, la métrica de cohesión está representada con la fórmula relacional siguiente:

Donde H es el nivel de cohesión, dado R como el número de relaciones internas al ensamblado y N el número de tipos o clases del mismo (Smacchia, Metrics Definition, 2004).

El patrón Modelo Vista Controlador

En el lenguaje Objective C, y sobre todo en el desarrollo de aplicaciones móviles, el patrón de diseño Modelo Vista Controlador es utilizado de forma casi obligatoria, según las normas que dicta Apple sobre sus herramientas y kit de desarrollo (Apple, 2012). Las aplicaciones móviles se componen de una o varias vistas, las cuales realizan interacciones con el usuario, ya sea para obtener datos o para mostrarlos.

El conjunto de clases diseñadas únicamente para contener o procesar datos es conocido como el Modelo. Las clases que procesan las señales de entrada provenientes del usuario, tales como un toque en la pantalla o el movimiento del dispositivo, y a la vez otros eventos del modelo de datos, son conocidas como Controladores. Finalmente, en el caso especial del desarrollo en iOS, la Vista se conforma de archivos especiales conocidos con Nibs (extensión .xib) o Storyboards, básicamente guiones gráficos que describen cada vista y su relación con otras. Dichos archivos contienen una estructura XML especial para definir interfaces gráficas conformadas por los controles nativos de la plataforma y la especificación de cada relación, incluyendo enlaces a otra vista, salidas o conexiones de controles y acciones hacia el controlador.

Comunicación entre objetos

En el ambiente de desarrollo de iOS, se pueden encontrar algunos patrones de comunicación entre objetos que tienen como responsabilidad enviar o responder a señales, mensajes o interacciones (Ganem, 2013). El objetivo principal del presente artículo es brindar un análisis sobre cada uno de estos métodos, para obtener una idea más clara de los pros y contras de cada uno, así como sugerencias acerca de en qué contextos deberían ser preferiblemente utilizados cada uno (Hocking, 2011).

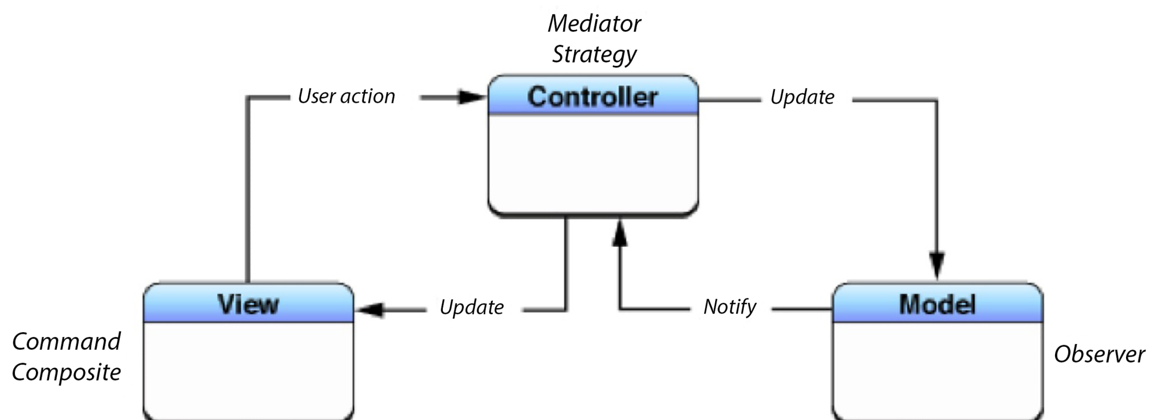

Figura 1 Diagrama Modelo-Vista-Controlador. Fuente: Apple, 2012.

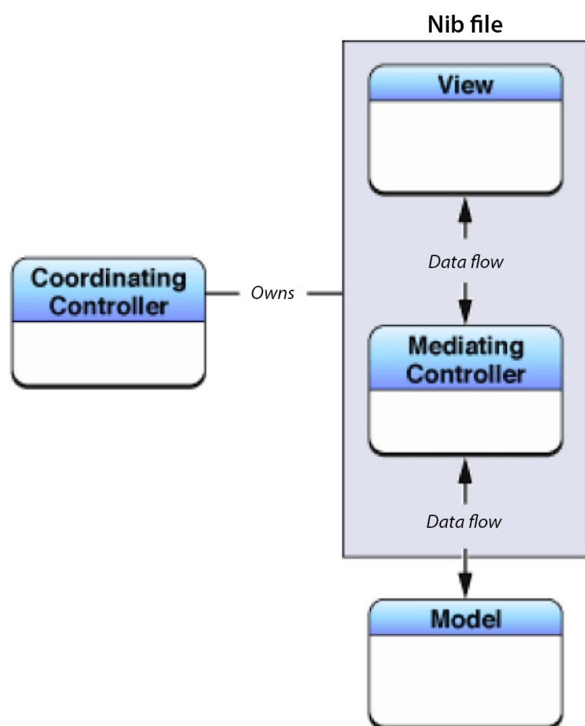

Figura 2 Coordinación de controlador como dueño de un xib. Fuente: Apple, 2012.

Los métodos principales son los siguientes:

- Delegados
- Notificaciones
- Observación
- Bloques de finalización

Delegados

Para el tema de comunicación, delegados y protocolos van de la mano. Los protocolos son lo que en otros lenguajes de programación se entenderían como interfaces. Poseen una declaración de métodos o mensajes que deben ser implementados ya sea obligatoria u opcionalmente por un objeto. La facilidad que proporcionan los delegados es simple: para un objeto poder comunicar algo a otro, no debe conocer la clase del primero, solo le basta saber que es un delegado de un determinado protocolo. Esto le asegura al emisor que el otro objeto sabe cómo reaccionar ante dicho mensaje.

Dentro del mismo marco de desarrollo de iOS, se observa que este patrón de programación es muy utilizado, sobre todo en objetos gráficos como controles, en los que resulta útil permitir al desarrollador que seleccione qué controlador u objeto desea que reciba los mensajes de eventos en dicho control. Por ejemplo, un caso típico es usar un control de vista de tabla. En esta situación es normal que sea necesario saber cuándo se selecciona una de las filas de la tabla, y generalmente es el mismo controlador de la vista el que asume la responsabilidad de decidir qué ocurre (navegar a otra vista, mostrar un menú, etc.).

Las fuentes de datos son patrones similares, pero la función principal de estas es indicar al emisor cierta información. Siguiendo el ejemplo de la vista de tabla, para los casos en que estas no son estáticas, es necesario declarar un controlador que determine sus datos. Nuevamente, es común que dicha fuente de datos sea el mismo controlador de la vista, sin embargo puede ser cualquier otro objeto que cumpla con el protocolo correspondiente (determinar la cantidad de secciones, la cantidad de filas por sección, etc.).

Entre algunas de las ventajas de este patrón se encuentran:

- El delegado puede ser de cualquier tipo, siempre y cuando cumpla con lo especificado por el protocolo.
- Sintaxis de programación estricta. Las llamadas son mensajes comunes, claramente definidas en el protocolo, por lo que los parámetros tienen un tipo fuertemente definido.
- Errores o advertencias en tiempo de compilación si algún método no está implementado de la manera correcta.
- El flujo de la comunicación es fácil de rastrear pues las llamadas son sincrónicas.
- Un mismo controlador puede implementar varios protocolos, con distintos delegados cada uno.

La comunicación puede ser bilateral, pues una llamada a un método del protocolo puede opcionalmente devolver un valor (como ocurre con las fuentes de datos).

Por otro lado, algunas de las desventajas que presentan los delegados son:

- Su declaración puede ser algo extensa a nivel de líneas de código, pues requieren la declaración

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {

    return [elements count];
}


- (UITableViewCell *)tableView:(UITableView *)tableView
            cellForRowAtIndexPath:(NSIndexPath *)indexPath
    {
        static NSString *MyIdentifier = @"MyIdentifier";

        UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:MyIdentifier];

        if (cell == nil)
```

Figura 3 Uso de delegado en vista de tabla

del protocolo. Es recomendado que esta declaración se haga en un archivo independiente. También requiere declarar la propiedad del delegado, y la implementación del protocolo en el delegado mismo.

- Requiere establecer el delegado en nulo al remover de memoria el controlador. No hacer esto puede causar un error grave de memoria en tiempo de ejecución debido a llamadas a un objeto no asignado.
- Tener varios delegados para un mismo evento requiere de una implementación difícil de mantener, pues básicamente necesitaría hacer la llamada a cada uno. Este patrón está diseñado para tener únicamente un receptor.

Notificaciones

En iOS existe un concepto de centro de notificaciones. Básicamente es una variable "singleton" que se encarga de recibir publicaciones de mensajes (eventos) y registrar observadores para estos mensajes. Entre las características de este patrón se encuentran el hecho de requerir un nivel de acoplamiento muy bajo, el uso de llaves de notificación, las cuales son cadenas de caracteres, y la capacidad de que cualquier objeto pueda recibir las notificaciones.

Un ejemplo del uso de este patrón es al crear aplicaciones que hacen constantemente llamadas a un API en un sitio web. Es posible que varios componentes requieran conocer el momento en que se recibieron nuevos datos. Para esto, cada uno de esos componentes debe registrarse como observador de la notificación, y el controlador encargado de recibir los datos debe publicar la notificación utilizando la llave indicada.

Otra característica de este tipo de comunicación es que el único mensaje que se puede enviar a los observadores es un objeto de clase diccionario, por lo que tanto emisor como receptores deben conocer la estructura exacta del contenido de ese objeto.

Algunas ventajas de las notificaciones son las que siguen:

- Fácil de implementar, no requiere muchas líneas de código.
- Pueden haber varios objetos receptores de una misma notificación.
- Se puede enviar datos a los receptores utilizando un objeto diccionario.

Por otro lado, algunas desventajas son:

- En tiempo de compilación es imposible asegurarse de que las notificaciones sean manejadas correctamente por los observadores.

```
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(receiveTestNotification:)
        name:@"TestNotification"
        object:nil];

    return self;
}

- (void) receiveTestNotification:(NSNotification *) notification
{

    if ([[notification name] isEqualToString:@"TestNotification"])
        NSLog (@"Successfully received the test notification!");
}
```

Figura 4 Ejemplo de uso de notificaciones

- Se necesita eliminar el observador al desasignar el objeto, de otra forma podrían ocurrir errores de memoria.
- No es muy rastreable. La ejecución de los observadores es más compleja y depurar este tipo de mensajes requiere más trabajo.
- El nombre de la notificación (llave) y del diccionario enviado debe ser conocido por ambas partes.
- No hay forma de que quien publica la notificación reciba algún dato de vuelta. Y de cualquier manera sería muy complejo procesar alguna posible respuesta debido a que puede haber más de un observador que atiende el evento.

Observación

Por sus siglas en inglés, KVO, la observación de llave-valor es un patrón que consiste en observar el valor de la propiedad de algún objeto para saber cuándo esta cambia. A diferencia de los dos patrones anteriores, delegación y notificaciones, la observación es más útil aplicada en objetos más que en controladores (Thompson, 2013). Solamente es utilizado en propiedades y no puede ser usado en métodos.

Algunas ventajas de este patrón involucran:

- Permite una forma sencilla de proveer sincronía entre dos objetos. Un caso práctico de esto podría ser la relación entre un modelo y una vista.
- Provee una forma de responder a cambios de estados de objetos que no fueron creados por uno mismo, y cuya implementación no puede ser modificada (por ejemplo, en el SDK).
- Puede mostrar el valor anterior y el nuevo de la propiedad que se está observando.
- Se pueden indicar propiedades anidadas.
- Abstracción completa sobre el objeto observado, pues no se necesita agregar ningún código extra.

También se tienen algunas desventajas, como las siguientes:

- Las propiedades observadas se definen utilizando cadenas de caracteres, lo que podría no permite recibir advertencias o errores en tiempo de compilación. Sin embargo hay formas de arreglar esta situación con un poco de código extra, utilizando un selector de método y convirtiéndolo a cadena (Thompson, 2013).
- La refactorización de código puede producir que este patrón deje de funcionar (al alterar los nombres de propiedades).
- Debido a que la observación se realiza utilizando un solo método, el código puede resultar en una gran cantidad de condicionales anidados en caso de observar varias propiedades.
- Se necesita eliminar el observador al desasignar el objeto.

Bloques de finalización

Este último patrón consiste en bloques enviados en línea como parámetro de un método. Los bloques de código son una característica que fue agregada a C, Objective C y C++. Estos son lo que en otros lenguajes de programación se conocerían como lambdas, y representan código ejecutable que puede ser guardado en variables o pasado como parámetro, y que a la vez puede retornar datos.

Este tipo de patrón se puede ver a menudo en eventos que pueden tomar algún tiempo, o en llamadas asíncronas. Un ejemplo puede ser las animaciones en iOS, las cuales por lo general poseen un parámetro donde se indican los cambios en una vista (la cual se animará), como se muestra en la figura 5, o incluso un bloque de finalización que se ejecuta cuando la animación ha terminado.

Como segundo ejemplo se puede mencionar las llamadas asíncronas a servicios web, que pueden ejecutar un bloque de finalización en caso de recibir correctamente datos, y otro en caso de haber algún error. Este ejemplo muestra cómo este patrón podría sustituir a un delegado.

Algunas ventajas de este patrón son:

- Permiten indicar una manera flexible de establecer una respuesta ante un evento.
- Este patrón permite enviar parámetros y obtener datos de vuelta con clases estrictamente definidas.
- No requieren de código extra para ser definidos, únicamente de que un parámetro sea de tipo bloque.

```
UIView* view = [self.view viewWithTag:100];
[UIView animateWithDuration:0.5
                              delay:0.1
                            options: UIViewAnimationCurveEaseOut
                         animations:^
        {
            CGRect frame = view.frame;
            frame.origin.y = 0;
            frame.origin.x = (-100);
            view.frame = frame;
        }
                         completion:^(BOOL finished)
        {
            NSLog(@"Completed");

        }];
```

Figura 5 Ejemplo de bloque de completición en animaciones de vistas

```
NSURLRequest *request = [NSURLRequest requestWithURL:URL];
NSURLSessionDataTask *dataTask = [manager dataTaskWithRequest:request
    completionHandler:^(NSURLResponse *response, id responseObject, NSError *error) {
    if (error) {
        NSLog(@"error!");
    } else {
        NSLog(@"task successful!");
    }
}];
[dataTask resume];
```

Figura 6 Ejemplo de bloque de finalización en llamada asíncrona

Por otro lado, algunas desventajas son las siguientes:

- Pueden crean una alta cohesión al ser usados.
- No tienen un alcance global.

Experiencias de desarrollo

Durante el tiempo la practican se estuvo siempre en el ámbito de desarrollo para iOS, y desde entonces también se ha seguido dedicando mayormente a esta plataforma de desarrollo.

En todo este tiempo se lograró aprender, evaluar y utilizar distintos modelos o patrones de desarrollo, ya sea a gran escala o en extractos específicos de una tarea. Para dar algunos ejemplos:

El uso de delegación es ampliamente visto en cualquier proyecto, ya sea al utilizar componentes propios de la plataforma, y al crear uno mismo sus propias clases y objetos. Este patrón permite definir una estructura, pero sobre todo, un comportamiento. Las notificaciones funcionan muy bien para dar a conocer eventos en un rango más grande de emisión y con más posibles receptores, como al necesitar dar a conocer a varios componentes de que hay nuevos datos por mostrar, procesar, etc. Los bloques de finalización funcionan muy bien para dar opción al desarrollador de ejecutar algo luego de una llamada asíncrona, y la observación es perfecta para estar al tanto del estado de un objeto y responder ante cualquier cambio de este.

En uno de los proyectos que se ha tenido que trabajar, se utilizó hasta tres de estos patrones juntos. La descripción del funcionamiento es la siguiente:

Una aplicación móvil requiere hacer llamadas a un servicio web constantemente para revisar si existen nuevos datos disponibles y procesarlos, o para ejecutar una acción como publicar un mensaje. La llamada al servicio se hace utilizando un método propio del marco de desarrollo de iOS, el cual acepta como parámetro un bloque de finalización. Se garantiza que este código será ejecutado únicamente cuando los resultados hayan llegado. Estos datos se procesan para ver si la llamada se hizo correctamente o si hubo algún error. Dependiendo del resultado de este proceso, se indica al delegado de la clase que se encargó de traer los datos. Este último procesará nuevamente el resultado para realizar los cambios correspondientes en un nivel más alto de abstracción y al final emitirá una notificación para que cualquier controlador que requiera dichos datos sepa que hay nueva información disponible, que alguna acción se ejecutó correctamente o que hubo un fallo al ejecutarla. A continuación, los controles se encargan independientemente de mostrar los nuevos datos o el resultado de la acción a nivel de vista.

Conclusiones

En el caso de los patrones de programación generalmente no existe un favorito, y la decisión sobre qué metodología usar proviene siempre del contexto y de las necesidades específicas presentes en el diseño de la aplicación. También es importante rescatar que estos patrones pueden ser usados en conjunto, y no necesariamente hay que elegir uno.

Otra consideración que se debe tener en la producción de software es seguir la guía de estándares y buenas prácticas. En el caso de estos patrones, es fácil localizar dicha información en el portal de desarrolladores de Apple.

Referencias

Apple. (13 de Diciembre de 2012). *Coordinating Efforts Between View Controllers*. Obtenido de iOS Developer Library: https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/ManagingDataFlowBetweenViewControllers/ManagingDataFlowBetweenViewControllers.html

Apple. (17 de Julio de 2012). *KVO Compliance*. Obtenido de iOS Developer Library: https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/KeyValueObserving/Articles/KVOCompliance.html

Apple. (9 de Enero de 2012). *Model-View-Controller*. Obtenido de iOS Developer Library: https://developer.apple.com/library/ios/documentation/general/conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html

Apple. (13 de Diciembre de 2012). *Working with Blocks*. Obtenido de iOS Developer Library: https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithBlocks/WorkingwithBlocks.html

Ganem, E. (4 de Setiembre de 2013). *iOS Design Patterns*. Obtenido de RayWenderlich: http://www.raywenderlich.com/46988/ios-design-patterns

Hocking, C. (14 de Junio de 2011). *When to use Delegation, Notification, or Observation in iOS*. Obtenido de Shine Technologies: http://blog.shinetech.com/2011/06/14/delegation-notification-and-observation/

Smacchia, P. (2004). *Metrics Definition.* Retrieved from ndepend: http://www.ndepend.com/Metrics.aspx#RelationalCohesion

Smacchia, P. (15 de Febrero de 2008). *Code metrics on Coupling, Dead Code, Design flaws and Re-engineering.* Obtenido de CodeBetter: http://codebetter.com/patricks-macchia/2008/02/15/code-metrics-on-coupling-dead-code-design-flaws-and-re-engineering/

Thompson, M. (7 de Octubre de 2013). *Key-Value Observing.* Obtenido de NSHipster: http://nshipster.com/key-value-observing/