**Instituto Tecnológico de Costa Rica**

Área de Ingeniería en Computadores

CE-4101 Especificación y Diseño de Software



Investigación del funcionamiento de

patrones de diseño

Elaborado por:

* Arturo Chinchilla 2013009344
* Fabián Solano Rodríguez 2013116291
* Juan Navarro Camacho 201236227
* Malcolm Davis Steele 201271325
* Santiago Gamboa Ramírez 2014092362

Profesor:

Ing. Daniel Madriz Huertas

Cartago

Segundo Semestre 2017

# Patrón *Observer*

## Descripción

Establece una relación de uno a muchos, de manera que cuando un objeto cambia de estado, los demás serán notificados de dicho cambio localizado y responderán actualizando sus estados y/o ejecutando funciones.

Dentro el contexto de este patrón se presenta por un lado considerar reducir el acoplamiento de los elementos constituyentes del sistema, para así evitar comprometer de manera negativa la reutilización de código. De lo anterior se puede entender que en este patrón requiere de asegurar el particionamiento efectivo de los elementos y por otro lado responder a la demanda de interoperabilidad de los constituyentes del sistema.

## La estrategia está relacionada con uno/varios objetos que son objeto de observación, *publishers*, que notifican a todos aquellos que requieran conocer de sus cambios de estado de forma transparente, es decir la cantidad de objetos suscritos, para ser notificados producto de los cambios de un objeto determinado, podrá variar.

Cabe destacar que el patrón observer es aplicable cuando no se tiene definido la cantidad, puede variar, de objetos que dependen del cambio de uno o más objetos. Junto con lo anterior el patrón observer aparece cuando se desea asegurar la transparencia a la hora de notificar a los objetos que requieren información del cambio de otros.

## Estructura

Imagen: detalle del diagrama de UML de la aplicación contextual del patrón observer.

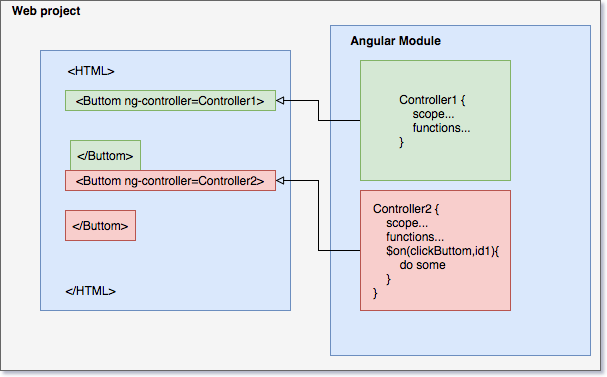
La dinámica de este patrón se realiza de la siguiente manera:

* La clase Subject define un objeto que es observado, establece funciones para agregar y eliminar a todos los objetos que desean observar a este tipo de objeto.
* La clase Observer abstrae, a través de herencia, a todos los objetos que desean estar al tanto de los cambio de estado de un/unos objetos que heredan de Subject.
* La estrategia es hacer transparente quiénes son los objetos observados y los observadores, pues utilizando herencia, permite que los objetos concretos observadores mediante el método Update() actualicen sus respectivos estados de forma personalizada, pues su implementación es en la clase hija y no en la clase Observer.
* Es importante que los objetos Observadores son actualizados desde una llamada por medio del Objeto Subject. Esta estrategia hace que el **Concrete Observer** no esté pendiente y realizando constantes consultas al objeto **Concrete Subject** para saber si este ha cambiado.

## Ejemplo

Las secciones de HTML, llámese etiquetas HTML, pueden ser controladas por diferentes controladores, Angular.Controllers, por lo que eventualmente cada espacio de ejecuci[on de funcionalidades, scope, se verán aislados de la interacción de scopes hermanos controlando otras secciones de HTML.

La función $on(eventname, ...) permite detectar desde cualquier controlador, la ejecución de funcionalidades y toma de decisiones producto del lanzamiento de una señal que indica que un evento se ha ejecutado, por ejemplo un controlador ajeno a la interacción y control de un elemento web se entere que ha sido seleccionado al igual que el controlador destinado a ser el que gobierne dicho elemento HTML.

Se adjunta imagen de ayuda al ejemplo anterior.

Se observa que la etiqueta verde es controlada con el **Controlador1** sin embargo **Angular** y **HTML 5** permite que se lanze un evento al hacer click en la etiqueta **Verde**, la cual permitirá ejecutar código **JS** en el **Controlador2**, controlador que observaba el comportamiento del botón verde.

Repository

Descripción

Este patrón se utiliza para la lógica de acceso a datos de servicios como una base de datos, un servicio web o una lista de puntos de compartimiento. El mismo trata de evitar problemas que se presentan cuando hay una comunicación directa para el acceso a datos, problemas como: código duplicado, alta probabilidad a errores de programación, una débil tipificación de los datos del negocio, una dificultad para centralizar las políticas relacionadas con los datos como el almacenamiento en caché además que queda inhabilitado para realizar fácilmente pruebas de lógica de negocio, aislado de dependencias externas.

Por medio del patrón se pueden lograr los siguientes objetivos:

* Aislar la capa de datos para respaldar las pruebas de la unidad además para maximizar la cantidad de código que puede ser probado automáticamente.
* Se accede a los datos desde muchos distintos puntos .
* Se desean aplicar reglas de acceso a muchas ubicaciones de acceso y administrar centralizadamente para mantener consistencia en el sistema.
* Desea implementar y centralizar una estrategia de almacenamiento en caché para la fuente de datos.
* Desea mejorar la capacidad de mantenimiento y la legibilidad del código al separar la lógica empresarial de la lógica de acceso a datos o servicios.
* Desea utilizar entidades comerciales fuertemente tipadas para que pueda identificar problemas en tiempo de compilación en lugar de en tiempo de ejecución.
* Desea asociar un comportamiento con los datos relacionados.
* Desea aplicar un modelo de dominio para simplificar la lógica empresarial compleja.

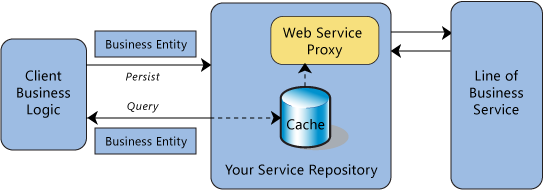
## Estructura:

## Imagen: detalle de interacciones con el repositorio.

Como se muestra en la imagen correspondiente, los repositorios son puentes entre dos dominios diferentes. Normalmente se hace un mapeo de datos entre dos dominios con diferentes tipos de datos, para esto se utiliza en algunos casos el patrón Data Mapper.

Además los repositorios eliminan las dependencias de las llamadas de los clientes. Al hacer el mapeo de datos, se logra eso.

## Ejemplo



## Imagen: Ejemplo de un repositorio con un servicio web.

Un ejemplo común de almacenar datos es un servicio de negocios que es expuesto por una aplicación de línea de negocios. Normalmente estos servicios están en un nivel de abstracción más alto que el de la semántica normal CRUD de una base de datos. Esto hace que acceder a estos servicios pueda ser complejo y pueda generar errores. Este repositorio centraliza el acceso lógico a un servicio y provee un punto de substitución para las pruebas unitarias. Además tiene una caché para optimizar los accesos.

Unit of Work (Unidad de Trabajo)

Descripción

*“Maintains a list of objects that are affected by a business transaction and coordinates the writing out of changes and resolution of concurrency problems”.*

Al realizar movimientos de objetos en una base de datos, es importante mantener un rastreo de los cambios realizados, ya que de otra manera, no se podrá tener de regreso la información.

Al realizar cambios en la base de datos relacionados al modelo, esto puede generar muchas pequeñas llamadas a la base, provocando que todo vaya muy lento. Además esto requerirá de tener una transacción abierta para todas estas interacciones, lo cual resulta poco práctico si se tiene una transacción comercial abarcando múltiples solicitudes. Y la situación se vuelve aún peor si se necesita dar seguimiento a los objetos leídos en el proceso de evitar lecturas inconsistentes.

La Unidad de Trabajo ayuda a realizar el seguimiento de todo lo que se hace durante una transacción comercial, que puede o no tener ramificaciones dentro de la base de datos.

Los cambios en la base de datos son la principal causa de usar la Unidad de Trabajo, inserciones, actualizaciones y borrado de datos, esta se encarga de llevar el seguimiento de todas estas transacciones. La unidad de Trabajo debería ser creada cada vez que se hace algo que afecte la base de datos, para que comience a hacer el seguimiento de datos, de esta manera, cada vez que se da un cambio, la Unidad de Trabajo será informada. Además se puede informar sobre los datos leídos, y así la Unidad de Trabajo puede verificar lecturas inconsistentes y verificar que ninguna modificó la base de datos durante la transacción.

El punto clave de este patrón es que al llegar el momento de tomar una decisión, éste decide qué hacer. Abre una transacción o realiza una comprobación de concurrencia, esto se puede hacer de dos modos, usando un Bloqueo Pesimista sin conexión o un Bloqueo Optimista sin conexión y escribe los cambios en la base de datos.

Comúnmente en el desarrollo de aplicaciones, los programadores no llaman explícitamente a métodos para actualizar campos en la base de datos, y de esta forma no tienen que hacer un seguimiento de lo que ha cambiado, ni existe la preocupación sobre integridad referencial ni cómo afecta ésta el orden en el que se deben hacer las transacciones.

## Estructura

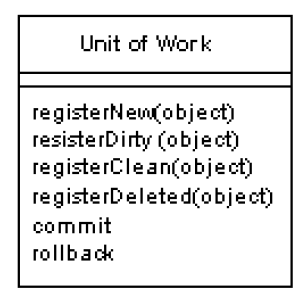


Imagen: Estructura de la Unidad de Trabajo.

Cada vez que el programador detecta que alguna acción va a modificar un sistema al que se le deba dar control de datos, se debe crear una unidad de trabajo para que dé el seguimiento necesario.

## Ejemplo

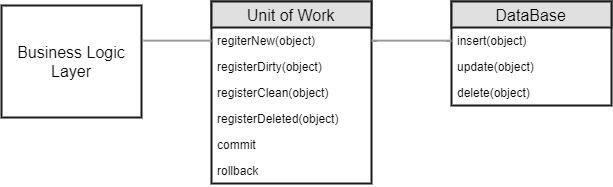


Imagen: Unidad de Trabajo aplicada a una base de datos.

Consiste en la introducción de una nueva capa, ubicada entre la capa de negocios y la base de datos, esto para que sea la encargada de dar el seguimiento de todos los cambios realizados a los datos. Además de tener un control sobre las lecturas inconsistentes y los cambios realizados por estas.

Conclusiones

* El patrón Observer se puede utilizar para notificar cambios de estado a los demás objetos. Estos objetos individualmente se actualizarán de acuerdo al cambio de estado. Este patrón es adecuado cuando se necesita disminuir el acoplamiento del sistema. Este patrón es ideal para cuando los objetos son dinámicos.
* El patrón Repository se puede utilizar cuando se necesita acceder a múltiples datos de servicios. Este patrón se utiliza cuando se quiere reducir problemas como duplicidad de código, alta probabilidad de errores y problemas relacionados al almacenamiento en caché.
* El patrón Unit of Work se utiliza para dar un seguimiento adecuado a los objetos leídos en el proceso de una transacción comercial. De esta forma se evita tener que crear llamadas innecesarias a la base de datos que va a provocar latencia en las consultas. El punto más alto de este patrón es que él es el encargado de tomar la decisión idónea.

Bibliografía

Fowler, M. (2003). Patterns of enterprise application architecture. Boston: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2016). *Design patterns*. Boston, Mass.: Addison-Wesley.

*AngularJS*. (2017). *Docs.angularjs.org*. Retrieved 22 November 2017, from <https://docs.angularjs.org/api>